Chapter

13

함수가 호출되기까지

☑ 스택 세그먼트와 스택 프레임은 어떻게 다른가요?

스택 세그먼트(Stack Segment)란 프로그램에서 사용하기 편하도록 메모리를 몇 가지 용도로 나눈 것 중 하나를 의미합니다. 즉, 프로그램의 코드가 들어 있는 코드 세그먼트, 전역 변수를 저장하는 데이터 세그먼트 등과 함께 용도에 따른 메모리 구분 방식 중 하나죠. 한편 스택 프레임(Stack Frame)이란 함수가 호출될 때마다 그 함수 호출을 위해 할당 받는 메모리 덩어리를 일컫습니다. 특히, 이 스택 프레임은 위에서 말한 몇 가지 구분된 메모리 중 스택 세그먼트에 잡히죠.

☑ 스택 프레임을 사용해서 얻는 이점은 어떤 것이 있나요?

C 프로그램은 함수의 호출 과정이라고 해도 과언이 아닙니다. main이라는 함수에서 시작해서 차례대로 다른 함수를 불러가며 프로그램이 진행되죠. 이런 함수 내부에서는 또다시 다른 함수를 호출하여, 이전 함수가 리턴 하기 전에 다른 함수가 불리는 중첩 호출이 일어납니다. 이처럼 함수 호출 시 가장 중요한 이슈가 복귀 주소와 인자의 전달인데, 문제는 이 때 레지스터나 메모리의 특정 절대 주소를 사용하면 중첩 호출이 불가능해지죠. 반면 스택 프레임을 사용하면 매번 함수가 호출될 때마다 스택 프레임이 구성되므로 인자 전달 및 복귀 주소를 스택 프레임을 통해 전달하고, 함수가 중첩되어 호출되더라도 매 호출 때마다 별도의 스택 프레임을 구성하므로 스택 크기만 충분하다면 얼마든지 중첩 호출이 가능합니다.

Section 01

벽돌과도 같은 함수 – 구조화된 프로그래밍

C에서 가장 중요한 요소 두 가지를 꼽으라면 여러분은 어떤 것을 들겠습니까? 필자는 주저 없이 변수와 함수를 들 것입니다. 변수가 없는 C는 그야말로 속 없는 만두나 마찬가지고, 함수가 없다면 만두피 없는 만두 속이랑 같을 것입니다.

앞에서도 언급하였지만 변수가 없는 C로는 만들 수 있는 프로그램이 거의 없습니다. 그런 중요성 때문에 우리는 이미 앞 장에서 변수에 대해 한 장 전체를 할애해서 살펴보았죠.

한편 함수가 없는 C로는 그나마 꽤 유용한 프로그램을 만들 수 있습니다. 단지 효율적이고 구조화된 프로그램을 만들 수 없다는 문제가 있긴 하지만요. 변수를 만두 속에 비유하고 만두피를 함수에 비유한 것은 그러한 이유 때문입니다. 만두의 주된 맛은 속에서 우러나오지만, 겉 피가 없는 만두는 맛은 어느 정도 낼 수 있을지 몰라도 더 이상 만두라고 부를 수 없죠.

이미 여러분은 C에서 함수가 무엇인지, 그리고 이 함수로 어떤 일을 할 수 있는지, 또는 함수를 어떻게 활용하면 프로그램을 효율적으로 작성할 수 있는지 경험적으로 터득하고 있으리라 믿습니다.

C에서 함수라는 것은 사실 일반적으로 구조화된(Structured) 프로그래밍 언어에서 '프로시저(Procedure)' 및 '함수(Function)'라고 부르는 두 가지 개념을 모두 포함하는 것입니다.

프로시저는 집을 지을 때 흙을 한줌한줌 쌓아 올리는 것보다 미리 벽돌이라는 것을 많이 만들어 두고 이 벽돌을 활용해 담장도 쌓고, 벽도 쌓고 하는 것과 비슷한 개념입니다. 즉, 프로그램 안에서 자주 쓰이는 기능을 프로시저라는 별도의 코드로 작성해 놓고 필요한 곳에서 이 프로시저를 부르는 것이죠. C에서는 이런 프로시저라는 것을 별도로 두지 않고 함수의 리턴값이 없는 void를 허용함으로써 대신하고 있습니다. 만일 이런 프로시저 혹은 함수라는 것이 없다면 여러분은 printf를 부르는 대신 글자를 화면에 뿌리기 위해 매번 printf 코드 전체를 글자를 출력하고자 하는 지점에 삽입해야 할 것입니다. 만일 printf 안에서 또 다른 함수를 부르고 있다면 여러분은 그 함수 코드 역시 매번 그 자리에 삽입해야 할 것입니다.

한편 함수라는 것은 프로시저와 닮아 있긴 하지만 호출의 결과값이 있다는 점에서 다릅니다. 함수는 마치 수학에서 y = f(x)처럼 인자 값 x를 넘기면 이 값을 잘 가공해서 y를 만드는 것이랑 비슷합니다.

실제로 파스칼 같은 언어에서는 함수랑 프로시저를 구분해서 따로 제공하고 있지만 리턴값이 있고 없고의 차이만 있을 뿐 사용 용도가 거의 비슷하기 때문에 C에서는 하나로 합쳐서 함수라는

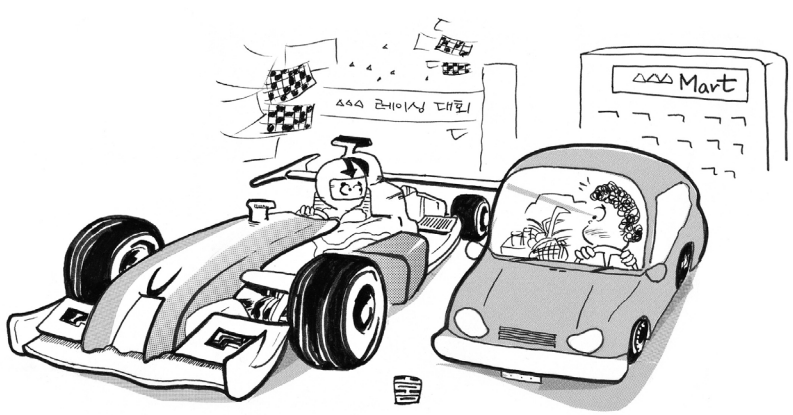
개념 한가지만 제공하고 있습니다.

그리고 함수를 잘 활용하면 자주 반복되는 코드를 묶어서 코드의 효율성을 높일 수 있습니다. 뿐만 아니라, 복잡한 기능을 일단 함수로 구현하고 나면 그 내부 구조는 잊어버려도 그 함수의 논리적 기능, 즉 입력과 출력 관계에 대해서만 기억함으로써 더 복잡한 프로그램을 만들 수 있습니다.

여기까지는 여러분도 비교적 잘 아는 내용일 것입니다. 뿐만 아니라 함수라는 것이 인자를 넘겨받고 결과값을 리턴(물론 void 형으로 선언하면 리턴값이 없겠지만)할 수 있다는 것도 잘 알고 있을 것입니다.

사실 이미 여러분은 함수라는 것을 아주 유용하게 잘 사용하고 있으며 그 사용법이나 효용성에 대해 굳이 얘기할 필요조차 없을 것입니다. 대신 여기서는 함수라는 것이 C에서 구체적으로 어떻게 구현되는지 자세히 살펴보도록 하겠습니다.

왜 굳이 그런 걸 자세히 알아야 하는지에 대해서는 아마 여러분이 더 잘 알고 있을겁니다. 장을 보러 차를 몰고 5분 거리에 있는 마트에 운전을 해 갈 때는 엑셀과 브레이크, 그리고 핸들 조작법에 대해서만 알면 아무 문제 없이 잘 다녀올 수 있지만, 레이서가 서킷에 들어가 경주를 한다든가 좀 더 다이나믹하게 차를 몰려면 차의 기본적인 원리에서 시작해 엔진의 구조라든가 특성 등에 대해서도 파악하고 있어야 하죠. 마찬가지로 여러분은 이 책을 선택한 순간 이미 단순히 어떻게든 돌아가기만 하는 프로그램을 작성할 줄 아는 '기능인'이 아니라 효율적인 프로그램을 작성하는 프로가 되려고 하는 사람들이니까요.



앞서 언급했듯이 여러분은 이미 함수가 무엇인지, 그리고 어떻게 호출하는지도 잘 알고 있습니다. 함수가 효율적인 것은 자주 쓰이는 루틴을 함수라는 이름으로 추상화해 두고(즉, 입출력 관계만 기억하고 그 내부 코드는 잊어버리는 거죠) 필요한 곳에서 그 함수를 '호출'해서 사용할 수 있기 때문입니다.

여기서 호출이라는 단어를 강조한 것은 말 그대로 함수를 부르는 지점에서는 함수의 코드가 삽입되는 것이 아니라 함수의 코드가 있는 곳으로 점프하기 때문입니다. 혹시 예전에 베이직 언어를 배운 적이 있다면 서브루틴이라는 단어를 들어보았을 것입니다. 바로 서브루틴과 같은 개념이죠.

☑ 서브루틴

따로 사용되지 않고 메인루틴과 결합하여 기능을 수행하며 특정 기능을 반복 수행해야 할 때 주로 이용합니다.

이를 좀 더 CPU 입장에서 본다면 jmp 인스트럭션을 통해 제어권이 함수의 코드가 있는 메모리 주소로 넘어간다는 것을 의미합니다(Part1을 상기해보세요). 단순히 이렇게 점프하는 자체는 사실 아주 간단합니다. 하지만 우리가 사용하는 함수의 기능을 생각해보면 단순히 점프하는 것만으로는 부족합니다.

우선 함수의 첫 번째 중요한 요소는 함수 루틴으로 점프하여 수행이 끝난 후, 다시 함수를 호출한 지점, 즉 점프 인스트럭션이 호출된 지점으로 복귀되어야 한다는 점입니다.

다음으로 함수를 호출할 때, 그리고 함수가 리턴할 때 함수와 호출된 지점 간에 데이터 교환이 있어야 한다는 점입니다. 즉, 함수를 호출할 때는 인자 값을 함수에게 넘겨 함수가 그 인자에 대해 적절한 동작을 할 수 있도록 해야 하며, 이 동작의 결과 발생한 결과값을 다시 호출한 코드에서 활용할 수 있도록 어떤 식으로든 다시 넘겨 받아야 합니다.

마지막으로 함수의 호출은 중첩 가능해야 합니다. 이 부분이 선뜻 이해가 안 가는 독자들이 많을 텐데, 이미 여러분이 당연한 듯이 쓰고 있는 기능입니다. 바로 함수 안에서 다시 함수를 부를 수 있어야 한다는 의미입니다. 그리고 이렇게 함수 안에서 함수를 호출할 때, 호출되는 함수가 호출하는 함수랑 동일할 수도 있어야 합니다. 이를 앞서 설명했듯 재귀호출이라 하여 경우에 따라 매우 유용하고 효율적인 코드를 작성할 수 있게 해 줍니다.

다시 정리하자면, 우리가 지금까지 너무나 당연하게 여기며 사용해왔던 함수는 크게 아래와 같이 세 가지 기능을 지원합니다.

1. **호출 지점으로 복귀 가능해야 한다.**
2. **호출하는 코드(이를 콜러-Caller라 부른다) 호출되는 함수(불린다는 의미에서 콜리-Callee라고 한다)간에 데이터 교환을 할 수 있어야 한다. 즉, 인자를 넘기고 리턴값을 받을 수 있어야 한다.**
3. **함수 안에서 또다시 자기 자신을 포함해 어떤 함수든지 부를 수 있도록 함수 호출은 중첩적이어야 합니다.**

그러면 과연 이러한 함수의 세 가지 기본 기능을 지원하기 위해 C는 어떤 알고리즘이나 자료구조를 사용할까요? 정답은 바로 우리가 바로 앞 장에서 배웠던 스택(Stack)이라는 자료구조입니다.

Section 02

함수호출의 기본 원리 – 스택 프레임

만일 여러분이 C를 사용하지 않고 어셈블리로 인스트럭션 레벨에서 직접 프로그래밍한다고 가정해봅시다. 실제로도 이런 어셈블리 프로그래밍은 예전뿐 아니라 지금도 많이 사용되고 있습니다. 특히, 리모컨이라든지, 세탁기와 같은 비교적 소규모의 임베디드 시스템에서 많이 사용되는데 PC 같은 대규모 시스템에서조차 디바이스 드라이버를 개발하는 것처럼 특별히 하드웨어를 제어해야 한다거나 효율성이 높은 코드를 작성하고자 할 때는 애용합니다.

그렇다면 여러분이 바로 이런 어셈블리로 프로그래밍을 할 경우 서브루틴을 어떻게 구현할 건가요? 여러분이 메모리를 복사하는 루틴을 만들고 이를 자주 활용해야 하는 경우, 복사할 시작 주소와 타겟 주소, 그리고 복사할 양을 서브루틴에 전달할 것입니다. 그리고 이 루틴이 끝난 후에 다시 호출된 지점으로 돌아와 하던 작업을 계속 해야겠죠. 결국은 함수의 기본 기능 중 적어도 첫 번째와 두 번째는 만족시켜야 하는 것입니다.

또한 십중팔구는 아마 레지스터를 활용할 것입니다. 즉, CPU가 가지고 있는 범용 레지스터 중 일부를 인자 전달, 혹은 서브루틴으로부터 리턴값을 받는 용도로 할애할 것입니다.

이는 분명 훌륭한 생각입니다. 실제로도 이 방식은 많이 활용되죠. 그런데 문제는 이런 식으로 레지스터만을 활용해 인자 전달을 하다 보면, 어지간히 많은 레지스터를 가지지 않고서는 곧 레지스터 부족에 시달리게 될 수 있습니다.

가령 방금 예를 든 메모리 복사 루틴이야 인자가 몇 개 필요 없지만, 전달해야 하는 인자가 100개 정도 되는 큰 규모의 서브루틴이 있다면 어떻게 할 건가요? CPU가 이보다 큰 110개의 범용 레지스터를 가지고 있다고 치더라도 문제입니다. 왜냐하면 100개의 레지스터는 이미 인자 전달용으로 사용되고 있기 때문에 정작 서브루틴 안에서는 나머지 10개의 레지스터만으로 작업해야 하므로 또 다시 심각한 레지스터 부족에 시달리게 됩니다.

이 뿐이 아닙니다. 호출하는 쪽에서도 레지스터를 사용하지 않는다는 보장이 없습니다. 즉, 110개의 레지스터 중 5개는 호출한 곳에서 사용 중이었고, 100개는 인자 전달용으로 사용되었다면, 서브루틴 안에선 나머지 5개의 레지스터만으로 모든 작업을 끝마쳐야 합니다.

게다가 더 최악인 것은 서브루틴은 자기를 호출한 시점에서 어떤 레지스터가 사용 중인지를 알 방법이 없다는 것입니다. 서브루틴이란 아무 곳에서나 부를 수 있게 마련이고 그 서브루틴을 부른 시점에서 방금 얘기한 것처럼 5개의 레지스터만을 사용한다는 보장이 없기 때문이죠

결과적으로 뭔가 특단의 대책이 없는 한 이런 방식의 서브루틴 호출(즉, C에서의 함수 호출)은 극

히 프로그래머를 피곤하게 만들 뿐 전혀 효율적이지 못한 것이죠. 더더군다나 함수의 세 가지 요건 중 세 번째 중첩 요건은 꿈도 꾸지 못하는 상황이 되죠.

하지만 정말이지 머리 좋은 사람들이 생각해낸 모든 것을 한방에 해결할 수 있는 획기적인 방법이 있습니다. 바로 스택 프레임(Stack Frame)이라는 것이죠.

여러분은 바로 이전 장에서 스택 세그먼트(Stack Segment)라는 것에 대해 공부하였습니다. 메모리의 용도를 크게 4가지 정도로 구분해서 사용하였고 그 중 하나가 바로 이 스택 세그먼트였습니다. 사실 스택이라는 용어는 메모리의 구분에 국한되는 것이 아니라, 각종 알고리즘에서 사용되는 대표적인 자료구조(Data Structure) 중 하나입니다. 데이터가 발생하면 접시를 쌓듯 가장 위쪽에서 차곡차곡 쌓아가고, 필요할 때는 가장 위쪽에 쌓인 데이터부터 다시 차근차근 꺼내 갈 수 있는 구조를 스택이라고 부릅니다.

우리는 이미 스택 세그먼트에 대해 살펴보았고, 함수의 호출과 함수 내의 지역 변수들이 이 스택 내에 생긴다는 것을 알고 있습니다. 그렇다면 스택 세그먼트와 스택 프레임이 어떻게 다른 것인지 살펴보고 좀 더 자세히 함수 호출 과정을 살펴보도록 하겠습니다.

우선 스택 세그먼트(Stack Segment)란 프로그램에서 사용하기 편하도록 메모리를 몇 가지 용도로 나눈 것 중 하나를 의미합니다. 즉, 프로그램의 코드가 들어 있는 코드 세그먼트, 전역 변수를 저장하는 데이터 세그먼트 등과 함께 용도에 따른 메모리 구분 방식 중 하나죠.

한편 스택 프레임(Stack Frame)이란 함수가 호출될 때마다 그 함수 호출을 위해 할당받는 메모리 덩어리를 일컫습니다. 특히 이 스택 프레임은 위에서 말한 몇 가지 구분된 메모리 중 스택 세그먼트에 잡히게 됩니다.

그렇다면 함수가 호출될 때 필요한 메모리란 어떤 것일까요?

우선 함수의 기본 요건 중 첫 번째가 함수의 수행이 끝난 후 함수를 호출한 곳으로 되돌아오는 것이었죠. 앞에서 함수를 호출한다는 것이 결국 jmp 인스트럭션 등을 써서 함수의 코드가 있는 곳으로 점프를 하는 거라고 말했습니다. 마찬가지로 함수가 끝나고 다시 호출된 코드가 있는 곳으로 돌아오는 것 역시 이런 식으로 점프 인스트럭션 등을 써서 구현합니다.

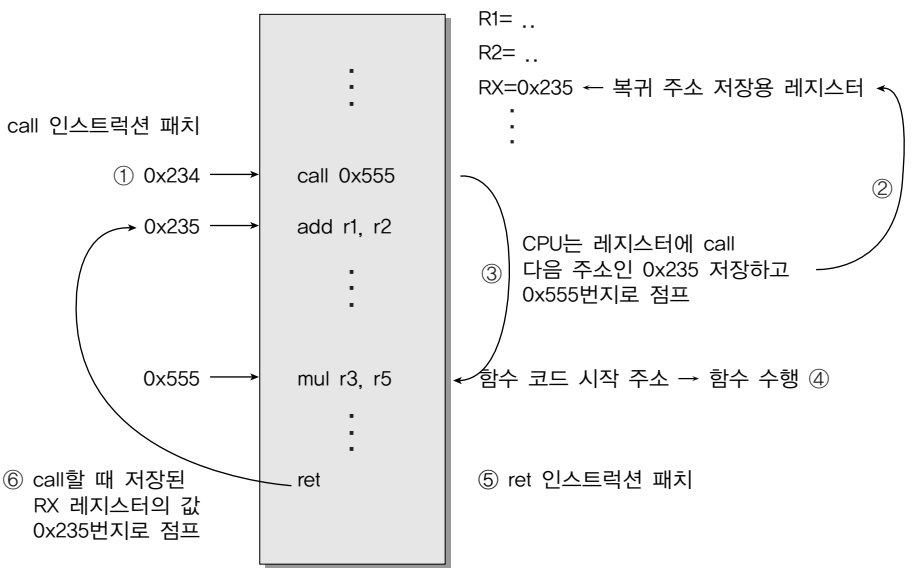
문제는 함수의 코드 자체는 고정된 메모리 주소에 존재하기 때문에 컴파일러가 함수 호출이 발생할 때마다 그 주소를 계산해서 만들어 줄 수 있지만 함수에서 호출 코드로 되돌아 갈 때는 이 주소가 고정되어 있지 않다는 점입니다. 왜냐하면 함수라는 것은 아무 곳에서나 불리기 마련이고 함수 입장에서는 이 '아무 곳'이라는 주소를 알 수가 없으므로 결국 jmp 인스트럭션으로 되돌아갈 수가 없는 것이죠.

따라서 함수를 호출하기 위해서 일반적으로 몇 가지 방식이 함께 쓰입니다. 우선 CPU에서 함수

호출을 지원해주기 위해 특별한 레지스터를 마련하고 이 레지스터에 리턴 주소를 보관하는 방식입니다. 그리고 CPU는 단순히 주어진 주소값으로만 점프하는 jmp 인스트럭션과 별도의 인스트럭션을 제공합니다. CPU마다 이 인스트럭션을 지칭하는 어셈블리 심볼은 틀리지만, 여기서는 call이라고 하겠습니다.

CPU는 인스트럭션을 수행할 때 이 call 인스트럭션을 만나면 jmp와 마찬가지로 call과 함께 주어진 주소로 점프합니다. 단, 이 때 그 call 인스트럭션 바로 다음 주소를 특정한 레지스터에 저장하게 됩니다.

그리고 일반적으로 CPU에서는 ret라는 함수 복귀 인스트럭션을 별도로 제공합니다. 이 ret 인스트럭션은 call 때 저장한 특정한 레지스터 값으로 점프하는 것입니다.



[그림 13-1] 레지스터 파일을 통한 범용 레지스터 엑세스

이런 방식은 굳이 CPU가 call이나 ret 인스트럭션을 별도로 지원하지 않아도 불가능하지는 않습니다. 단지 PC 레지스터만 읽을 수 있다면 그냥 원하는 레지스터에 PC 레지스터 값을 저장하고 jmp 인스트럭션으로 원하는 함수 코드로 점프하고 함수 마지막에는 복귀 주소를 저장한 레지스터 값으로 다시 점프하기만 하면 되기 때문이죠. 하지만 함수 호출은 실제로는 프로그램에 있어서는 매우 중요하고 또 빈번히 일어나는 작업이기 때문에 대다수 CPU는 함수 호출(정확히는 서브루틴 호출)을 위한 별도의 인스트럭션을 제공합니다. 이는 프로그래머가 직접 PC값을 저장하고 다시 jmp 인스트럭션을 사용하는 두 가지 작업을 하드웨어적으로 한 인스트럭션에 끝낼 수 있게 해주기 때문에 성능 향상에도 도움을 미칩니다.

그렇다면 위와 같이 특정한 레지스터를 통해 함수(서브루틴) 호출을 하게 되면 어떤 문제가 발생할까요?

함수가 갖추어야 할 두 번째 요소인 인자나 리턴값 전달은 별도로 얘기하고, 세 번째 요소인 중첩 호출이 불가능해집니다. 즉, 함수(서브루틴) 안에서 다시 또 다른 함수(서브루틴)를 호출할 수 없는 것입니다. 왜냐하면 call 인스트럭션을 통해 다른 함수를 호출하면 복귀 주소를 다시 RX 레지스터에 저장하게 될 것인데, 문제는 이미 RX에는 현재 수행 중인 함수가 복귀해야 할 주소가 보관되어 있다는 점입니다. 따라서 이런 경우 함수 안에서 함수 호출을 할 때는 결국 현재 RX 레지스터의 값을 따로 다른 레지스터나 메모리에 보관하고 호출해야 하죠. 그리고 호출한 함수가 리턴하면 다시 RX 레지스터에 원래 값을 복구해 놓아야 합니다.

만일 함수가 한 번만 중첩되는 것이 아니라면 문제는 더 심각해집니다. 호출된 함수는 자신이 몇 번째 중첩된 호출인지 알 수 없고 따라서 어떤 레지스터를 건드리면 안 되는지 알 수 없습니다. 왜냐하면 이전 호출에서 계속해서 다른 레지스터에 RX 레지스터 값을 백업할 것이기 때문이죠. 물론 메모리에 저장할 수도 있지만, 결국 메모리에 저장하는 경우도 절대 주소를 사용하면 레지스터와 다를 바가 없어집니다. 즉, 중첩 호출되면 같은 메모리 주소에 값을 쓰게 되고, 이전 값이 사라지는 것이죠.

그래서 고안된 방식이 바로 스택 프레임(Stack Frame)입니다.

함수호출의 중요한 특징 중 하나가 호출하는 함수(Caller)는 호출된 함수(Callee)가 리턴하기 전까지는 절대 리턴하지 않는다는 점입니다. 이러한 사실이 왜 중요한 지에 대해 다음과 같이 생각해 봅시다.

예를 들어 여러분이 영어 소설 책을 펼쳐 들었습니다. 이 책을 책상 위에 펼쳐 놓고 한 줄씩 읽어 나가고 있죠. 그리고 여러분 옆에는 사전이 한 권이 아니라 셀 수 없이 많이 있다고 가정합시다.

그런데 잘 읽어가던 도중 막히는 단어가 나왔습니다. 하지만 걱정할 필요는 없습니다. 사전이 한 권도 아니고 골라 잡을 정도로 많으니까요. 여러분은 읽고 있던 소설책에서 지금까지 읽은 곳을 표시해 놓고 사전을 이 소설 책 위에 겹쳐서 펼쳐 놓습니다. 그리고 모르는 단어를 찾고, 이 단어를 좀 더 철저히 알기 위해 예문까지 읽었습니다. 이런, 그런데 이번엔 이 예문에서 또 모르는 단어가 나왔네요. 하지만 여전히 걱정할 필요없습니다. 여러분에겐 사전이 한 권이 아니니까요. 다시 읽던 예문에 표시해 두고 또 다른 사전을 펼쳐 앞의 사전 위에 겹쳐 놓습니다. 그리고 방금 전 예문에서 몰랐던 단어를 찾습니다. 다행히 이번에 찾은 단어의 예문에는 모르는 단어가 없습니다. 따라서 차곡차곡 쌓인 세 권 중 제일 위의 사전을 들어 냅니다. 그리고 아까 읽던 첫 번째 단어의 예문을 중단했던 지점부터 다시 읽어갑니다. 그리고 이 과정이 끝나면 이 첫 번째 사전도 치우고

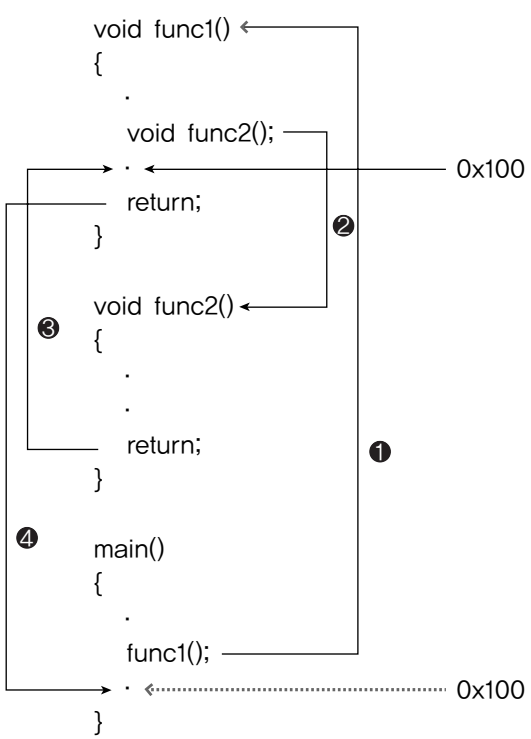
처음 읽던 소설책을 멈췄던 지점부터 다시 읽어가기 시작할 것입니다.

바로 이 과정이 여러분의 C 프로그램이 수행되는 것과 똑같습니다. 소설책은 main 함수이고 사전이 다른 함수라 생각할 수 있습니다. 중요한 것은 방금 든 예처럼 책 세 권(소설책 + 사전 두 권)뿐 아니라 더 많은 사전을 그 위에 겹쳐 놓더라도, 언젠가는 이 과정이 끝나고 다시 앞으로 돌아갈 때 갑자기 소설 책이나 중간에 쌓인 사전으로 돌아가는 것이 아니라, 가장 위에 펼쳐진 것부터 다시 차례대로 읽고, 치우는 것을 반복하며 마지막의 소설책까지 내려간다는 것입니다. 따라서 메모리에서 데이터를 저장할 때 이렇게 책을 쌓듯 차례대로 쌓아 올리고, 값을 해제할 때는 제일 위에서부터 차례대로 해제하는 방식을 '쌓아올린다'는 뜻의 스택(Stack)이라고 합니다.

다시 C 얘기로 돌아와서, 함수가 호출될 때 다른 건 둘째치고라도 당장 복귀 주소를 어딘가 기록해 둬야 함수가 리턴할 수 있을텐데, 이 복귀 주소를 앞에서 든 예처럼 특정 레지스터나, 특정 메모리 주소에 저장하는 대신 스택 공간 안에 쌓아간다고 생각해 봅시다.

그리고 이렇게 복귀 주소를 쌓아둔 메모리 중 가장 윗 부분 주소를 특정 레지스터에 저장해두도록 약속합니다. 우리는 이 레지스터를 스택 포인터(Stack Pointer), 줄여서 SP라고 부르도록 하겠습니다.

이제 다음 그림과 같은 상황을 생각해 봅시다.



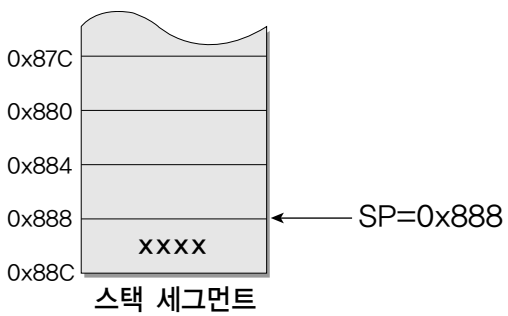
[그림 13-2] 스택 프레임 설명을 위한 함수 호출 예제

위 그림에서 나와 있는 소스 코드를 살펴보면 main 함수를 포함해 총 3개의 함수가 있고 main이 func1을 부르고 다시 func1 안에서 func2를 부르는 중첩 호출 구조로 되어 있습니다. 그리고 이 때 main이 func1을 호출한 바로 다음 주소를 0x400번지, func1에서 func2를 호출한 다음 주소가 0x100번지라 가정하겠습니다.

한편, 방금 앞에서 설명했던 대로 함수를 호출할 때 복귀 주소를 스택 구조로 차곡차곡 쌓아가며 저장한다고 했을 때 이렇게 저장하기 위한 스택 세그먼트의 시작 주소를 0x88C번지, 그리고 이 스택 주소는 항상 SP라는 스택 포인터 레지스터에 저장되어 있다고 가정하겠습니다.

이제 그림의 1~4번까지 각 번호가 실행되었을 때의 스택 세그먼트의 구성을 살펴봅시다.

우선 1번 화살표, 즉 main에서 func1이 호출되기 전까지의 스택 세그먼트의 구성입니다.



[그림 13-3] 1번 화살표가 실행되기 이전의 스택 세그먼트 상태

다시 한 번 앞에서 한 얘기를 상기해 보면, 함수가 호출될 때마다 복귀 주소를 스택에 차곡차곡 쌓아간다고 하였습니다. 그리고 이 때 가장 위의 주소를 SP 레지스터에 저장한다고 하였죠. 따라서 아직 아무 함수도 호출되지 않은 상태에서 SP 레지스터는 스택 세그먼트의 시작 주소인 0x88C번지를 가리키고 있어야 할 텐데, 의외로 0x888이라는 값을 가리키고 여기에 xxxx라는 값이 하나 채워져 있습니다.

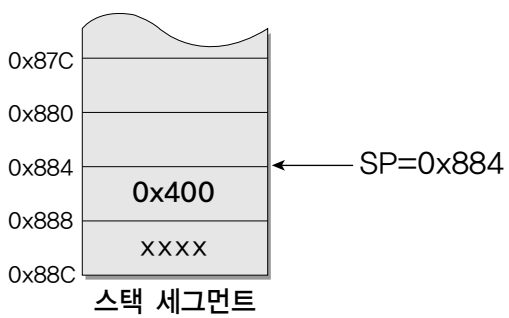
이는 바로 여러분이 프로그램의 시작점으로 생각하는 main 함수도 실은 시작지점(Entry Point)이 아니라 다른 함수들과 같은 일반 함수로 취급되기 때문입니다. 즉, main 함수도 다른 함수처럼 어디선가 호출이 된 함수이고 xxxx는 이 main 함수가 수행이 끝났을 때 복귀해야 할 주소인 것입니다.

이를 자세히 설명하자면, 프로그램이 시작되고 또 종료될 때 (사실 여러분은 신경 쓰지 않아도 되지만) 여러 작업이 일어납니다. 가령 앞 장에서 배운 것처럼 프로그램에서 사용할 힙(Heap)이나 스택(Stack)에 대한 할당 및 초기화라든지, 프로그램이 시작될 때 전달받은 인자들을 main 함

수로 전달하기 위한 작업이라든지, 또는 C 함수 자체적으로 필요한 초기화 작업 등, 여러 가지 일을 먼저 수행한 다음에 비로서 main 함수가 불리는 것입니다.

또한 main 함수가 리턴한 후에도 프로그램을 정상적으로 종료시키기 위한 마무리 작업 역시 필요합니다. 따라서 실제 프로그램이 시작되는 지점은 우리가 만드는 프로그램 소스 코드상엔 없지만 컴파일러가 별도로 만들어 넣고 이 부분에서 main 함수를 일반 함수처럼 호출하는 것입니다. 이 역시 하나의 일반 함수와 같이 생각할 수 있고 main 함수가 끝나고 난 후의 복귀 주소가 스택에 저장되어야 하는 것입니다. 결국 위 그림의 xxxx는 이러한 main 함수의 복귀 주소를 나타내는 것이며 이 크기인 4바이트만큼 스택 포인터 SP 역시 조정되어 0x888를 가리키게 되는 것입니다.

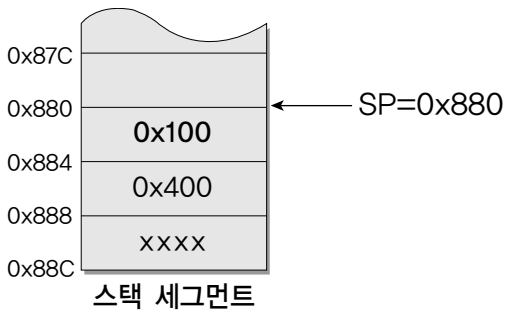
그렇다면 이제 1번 화살표, 즉 func1 함수가 불리고 난 후의 스택은 어떤 모습일까요?



[그림 13-4] 1번 화살표(func1 호출)가 실행된 이후의 스택 세그먼트 상태

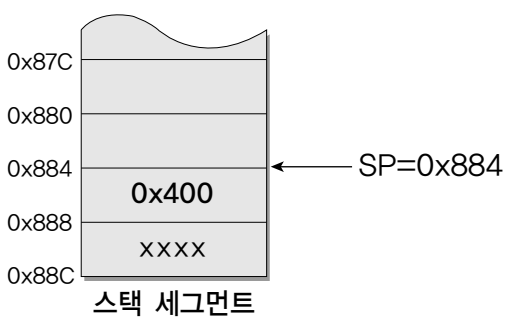
벌써 예상한 분도 있겠지만, main 함수 내에서 func1을 호출하는 바로 다음 지점 주소인 0x400이라는 값이 main 함수의 복귀 주소 바로 위에 저장되었습니다. 또한 이번에도 그 크기인 4바이트만큼 SP가 조정되어 0x884를 가리키겠죠.

마찬가지로 2번 화살표, 즉 func2가 호출되면 다음 그림과 같이 복귀 주소인 0x100번지가 저장되고 SP 역시 아래 그림과 같이 조정될 것입니다.



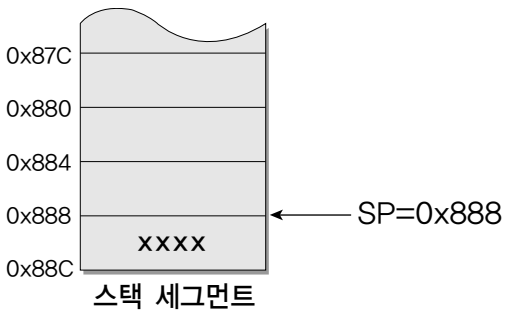
[그림 13-5] 2번 화살표(func2 호출)가 실행된 이후의 스택 세그먼트 상태

그렇다면 이제 func2 함수가 수행을 마치고 리턴하면 어떻게 될까요? func2 함수를 호출했던 바로 다음 주소인 0x100번지부터 프로그램이 수행되어야 할 것이고, 이미 이 복귀 주소는 스택의 제일 위에 저장되어 있습니다. 또한 이 값이 저장된 스택의 주소는 스택 포인터 레지스터 SP가 가리키는 메모리에서 읽어 쉽게 얻을 수 있습니다. 따라서 func2는 현재 SP가 가리키는 0x880번지에서 읽어온 0x400번지로 점프하게 되고, 동시에 마지막 읽어온 데이터 크기 4바이트만큼 SP 값을 조정해줘야 합니다. 그럼 결국 스택 상태는 다음 그림과 같이 다시 func2가 호출되기 이전의 상태로 되돌아가고 func1은 아무 일도 없었다는 듯 하던 일을 계속 할 수 있는 것입니다.



[그림 13-6] 3번 화살표(func2가 리턴)가 실행된 이후의 스택 세그먼트 상태 – func2를 호출하기 이전과 같다

그리고 예상할 수 있듯이 func1이 리턴하고 난 후의 모습 역시 다음 그림처럼 func1이 호출되기 이전과 마찬가지인 것입니다.



[그림 13-7] 4번 화살표(func1이 리턴)가 실행되고 난 이후의 스택 세그먼트 상태 – func1을 호출하기 이전과 같다.

이런 식이라면 아무리 함수가 중첩되어 많이 호출되더라도 혹은 심지어 함수가 다시 자신을 호출하는 재귀호출(Recursive Call)이 되더라도 각 호출 때마다의 복귀 주소가 항상 스택의 제일 꼭대기에 저장되고 SP 레지스터로 이 값을 엑세스할 수 있으므로 문제없이 호출된 지점으로 복귀할 수 있습니다.

이제 다시 원점으로 돌아와, 그렇다면 스택 프레임이라는 용어가 나타내는 것은 무엇일까요? 지금까지 설명한 내용에서 스택 프레임이라는 것은 바로 함수의 복귀 주소를 나타냅니다. 처음 스택 프레임을 얘기할 때 스택 프레임이란 함수를 호출할 때 그 함수 호출을 위해 필요한 데이터를 저장하는 메모리 덩어리라고 설명하였습니다. 따라서 지금까지 살펴본 바로는 함수를 호출할 때 꼭 필요한 데이터가 바로 복귀 주소였고 위 그림에서 4바이트 단위로 이루어진 복귀 주소 하나하나가 각각의 함수 호출에 대한 스택 프레임이 되는 것입니다.

그렇다면 실제 스택 프레임도 이렇게 복귀 주소 하나만으로 구성되는 것일까요?

Section 03 스택 프레임을 통한 함수간 값 전달

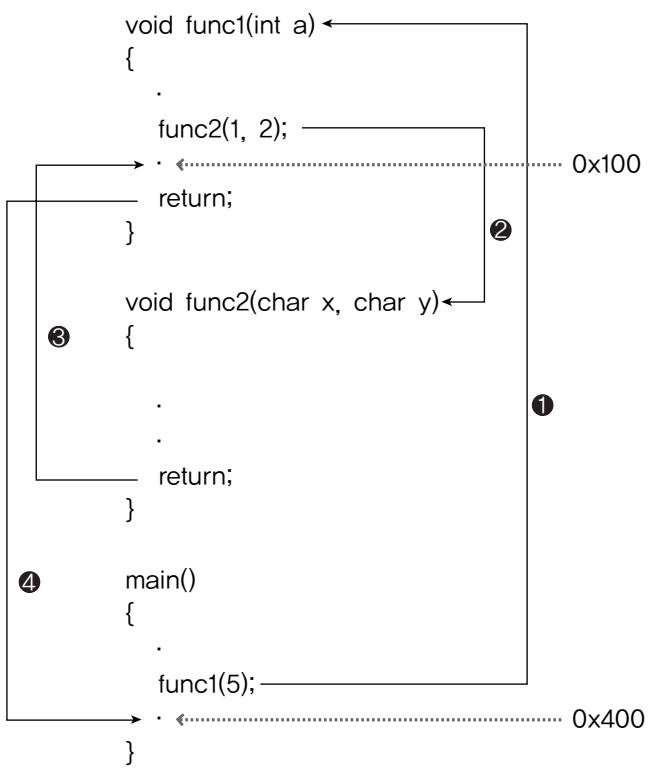
함수를 호출할 때 가장 필수적인 것이 복귀 주소이긴 하지만, 여러 다른 요소도 필요합니다. 앞에서 함수의 요건 중 두 번째가 바로 호출하는 측(Caller)과, 호출 당하는 함수(Callee)간의 데이터 교환이 가능해야 한다는 점이었습니다.

여러분이 함수를 호출할 때 인자없이 호출하는 경우도 있지만, 일반적으로 함수라는 것은 인자로 어떤 값을 전달하고 이 값에 대한 처리를 하여 그 결과를 다시 리턴해 주는 경우가 대부분입

니다. 이럴 때 이 인자를 전달하고 다시 그 결과값을 리턴받기 위한 용도로도 역시 스택 프레임이 활용됩니다.

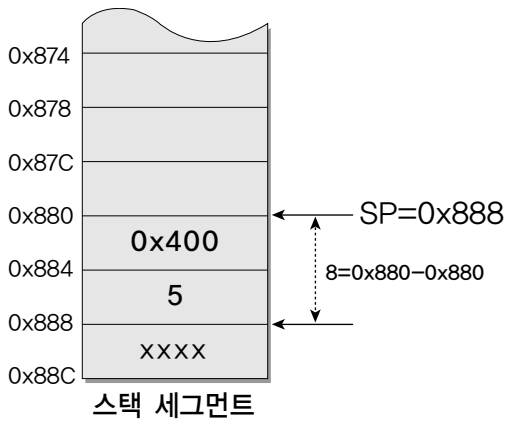
예를 들면 함수를 호출할 때 복귀 주소를 저장하고 그 주소 크기만큼 스택 포인터 SP를 조정하였죠. 만일 32비트 주소 체계를 가지는 CPU라면 이 값은 4바이트가 될 것입니다. 그런데 스택 프레임을 사용해서 인자 전달까지 하는 경우 함수 호출 시 이 인자들을 복귀 주소 바로 다음에 차곡차곡 저장하고 복귀 주소와 저장된 인자 크기만큼 SP를 조정해주고 호출받은 함수 측에서는 다시 이 SP를 사용하여 해당 값을 엑세스해 인자 교환이 일어나는 것입니다.

이를 살펴보기 위해 앞에서 사용한 함수 호출 코드를 살짝 수정해보겠습니다.



[그림 13-8] 인자가 있는 함수의 호출 예제

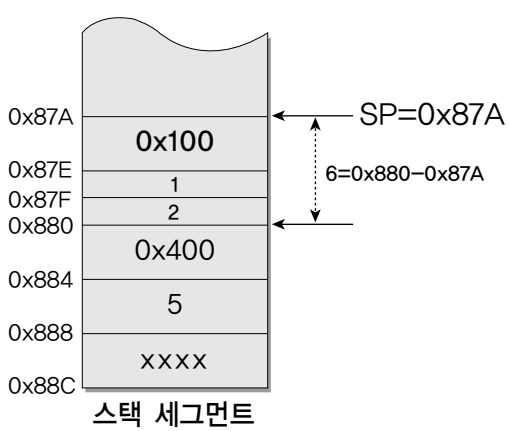
1번 선, 즉 func1이 호출되기 전까지는 앞에서와 마찬가지로 스택에는 main 함수의 복귀 주소만 저장되어 있습니다. 하지만 func1 호출이 일어난 후의 스택 모습은 앞서와 조금 달라집니다.



[그림 13-9] func1 호출 후의 스택

위 그림에서처럼 복귀 주소인 0x400 이외에도 func1에 전달하는 인자값 5도 스택에 같이 저장되었습니다. 그리고 마찬가지로 32비트 CPU라고 가정하면 이 때 주소값 크기 4바이트와 int 형 인자 크기인 4바이트까지 더해 총 8바이트만큼 SP 값이 조정되었습니다(0x884 – 8 = 0x87C. 왜 SP 값이 증가하지 않고 감소하는지는 뒤에 이어지는 '비타민 퀴즈 – 스택의 증가 방향'편을 참고하세요).

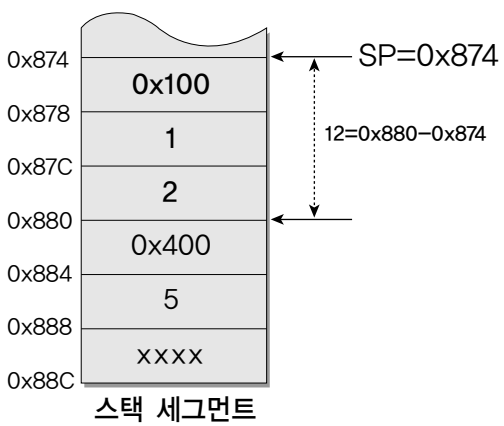
여기서 다시 func2가 호출되면(2번 선), 이번에는 func2가 복귀할 주소와 인자값 1, 2가 스택에 저장되고 다시 그만큼 SP 값이 조정될 것입니다. 그런데 여기서 잠깐 주의 깊게 살펴보면, func1의 인자는 int 형이므로 4바이트를 차지하여 복귀 주소까지 총 8바이트만큼 SP를 조정하고, func2를 호출할 때 인자는 모두 char이므로 2바이트만 차지하기 때문에 복귀주소까지 총 6바이트를 조정해줘야 할 것입니다.



[그림 13-10] func2 호출 후의 스택 –스택 Alignment가 되지 않은 경우

그런데 이렇게 데이터 형에 따라 1바이트, 혹은 2바이트처럼 스택 포인터 SP를 조절할 수 있으려면 CPU에서 이를 지원해줘야 합니다. 스택 포인터 레지스터는 일반적으로 그 값을 직접 조정할 수 없고 push나 pop으로 대표되는 인스트럭션을 통해 스택에 데이터를 저장하거나 빼오면서 자동으로 조절됩니다. 그런데 이런 인스트럭션은 CPU의 처리 단위, 즉 32비트 CPU라면 32비트(4바이트) 단위로 스택에 데이터를 저장하기 때문에 별도의 1바이트나 2바이트 단위로 push, pop을 하는 인스트럭션이 지원되지 않으면 위와 같이 데이터 형에 맞추어 스택 사이즈가 조절되는 경우는 없습니다. 또한 비록 CPU에서 이를 허용한다 하더라도 성능적인 면에서 딱 정해진 크기만큼씩 할당하고 해제하는 것(이를 정렬(Alignment)이라고 합니다)이 훨씬 효율적인 경우가 많습니다. 따라서 컴파일러가 의도적으로 코드를 생성할 때 위와 같은 경우도 int와 동일한 크기로 char 변수를 생성하기도 합니다.

가령 앞에서처럼 4바이트 단위로 정렬(Alignment)되었을 경우 다음과 같이 스택이 구성될 것입니다.



[그림 13-11] func2 호출 후의 스택 – 4바이트 정렬된 경우

이제 남은 것은 과연 저렇게 스택에 저장한 인자를 호출 된 함수(Callee)에서 어떻게 엑세스할 것인가 하는 부분입니다.

이 역시 복귀 주소 때와 마찬가지로 스택 포인터 SP를 이용해서 간단히 읽어 올 수 있습니다. 가령 바로 위 그림에서 func2가 전달받은 1과 2라는 인자값을 엑세스하고 싶으면, 현재 각각 SP + 4 및 SP + 8한 곳의 값을 읽어오면 되는 것입니다. 또한 func2가 수행을 마치고 리턴할 때는 바로 SP가 가리키는 0x100번지로 점프하면 되죠. 즉, 스택 포인터 레지스터는 항상 스택의 제일 끝을 가리키고 있으며, 이 끝 주소를 기준으로 앞 쪽에 차곡차곡 복귀 주소와 함수의 인자 값이 저장되어 있다는 것을 알고 있으므로 얼마든지 이들을 엑세스할 수 있는 것입니다.

한편, 앞에서 든 예제 코드에서의 함수들은 모두 리턴값이 없는 void 형이었습니다. 그런데 함수라는 것이 많은 경우 전달받은 값으로 특정한 작업을 하고, 그 결과를 호출한 측에 넘겨줘야 합니다. 이럴 경우 그러면 함수는 어떻게 리턴값을 넘겨 주게 될까요? 이 때도 스택 프레임을 활용할까요?

이 부분 역시 CPU나 컴파일러에 따라 다를 수 있지만 그래도 대부분의 경우는 레지스터를 이용합니다. 함수 호출 때 전달하는 인자와는 달리 리턴값은 하나로 고정되어 있기 때문에 주로 특정 레지스터를 통해 전달합니다. 가령 우리가 사용하는 PC에서는 주로 EAX라는 레지스터를 통해 값을 넘기죠(정수가 아닌 실수 값을 리턴하는 경우는 실수 연산용 별도 레지스터에 저장합니다). 호출받은 함수가 리턴하기 바로 직전에 EAX 레지스터에 리턴하고자 하는 값을 저장하고, 호출한 측은 함수호출이 끝나면 EAX 레지스터에 리턴값이 있는 걸로 간주하고 다음 처리를 해 나갑니다.

가령 int sum(int a,int b) 같은 함수가 있어서, 이 함수의 리턴값이 전달받은 a와 b의 합이라면, sum 함수 내부에서는 가장 마지막에 a + b 값을 eax 레지스터에 저장하고, sum 함수를 호출한 곳에서는 sum 함수의 호출 바로 다음부터 eax 레지스터의 값으로 원하는 작업을 하는 것이죠.

지금까지 대략 어떤 방식으로 함수 호출이 이루어지고 그 과정에서 인자전달이나 리턴값의 반환이 어떻게 이루어지는지 감은 잡았을 것입니다. 그런데 구체적으로, 예를 들어 스택 포인터 레지스터의 값을 조정한다고 했는데, 이런 작업은 호출하는 곳에서 하는 건지 호출받은 함수에서 하는 건지 등 자세한 사항에 대해 궁금할 수도 있겠죠. 그럼 백문불여일견(百聞不如一見)이라고 실제 코드를 살펴보며 이런 부분에 대해 의문을 해소해 보겠습니다.

```
#include <stdio.h>

int sum(int a,int b)
{
   return a + b;
}

void printresult(int c)
{
   printf("10 + 20 = %d",c);
}
int main(int argc, char* argv[])
{
   printresult(sum(10,20));

   return 1;
}
```

너무나도 간단한, 10 + 20의 결과를 출력하는 프로그램입니다. 사실, 일부러 함수 호출 과정을 설명하기 위해 작성한 것이라 그렇지 실제로는 저렇게 작성할 필요도 없이 간단한 프로그램이죠.

우선 실행 과정을 살펴보면, main 함수가 '호출'됩니다. 앞에서 말했다시피 main 함수 역시 실제로 프로그램이 출발하는 지점(Entry Point)이 실제 소스상에 나와 있지 않은 시작점이 있어서 그 곳에서 main 함수를 다른 함수처럼 호출하는 것입니다. 한편 main 함수 내에서는 리턴을 빼면 딱 한 줄의 코드 밖에 없습니다. 이 코드 안에서는 우선 10과 20이라는 인자를 넘기면서 sum이라는 함수를 수행하게 됩니다. 그리고 이 함수의 결과값을 그대로 다시 printresult라는 함수의 인자로 넘기면서 printresult 함수를 호출하게 되죠.

이제 컴파일러가 main 함수를 어떤 형태로 번역했는지 살펴보겠습니다.

```
12:   int main(int argc, char* argv[])
13:   {
00401021   push        ebp
00401022   mov         ebp,esp
14:       printresult(sum(10,20));
00401024   push        14h
00401026   push        0Ah
00401028   call        _sum (00401000)
0040102D   add         esp,8
00401030   push        eax
00401031   call        _printresult (0040100b)
00401036   add         esp,4
15:
16:       return 1;
00401039   mov         eax,1
17:   }
```

[그림 13-12] 컴파일된 main 함수의 어셈블리 코드

가장 위에 보면 printresult(sum(10,20));의 코드가 나타나기 전에 push ebp와 mov ebp, esp라는 코드 두 줄이 보입니다. 이 부분에 대해서는 잠시 후에 설명하도록 하고 우선 main 함수 내의 유일한 코드인 printresult(sum(10,20));가 어떻게 컴파일되었는지 살펴봅시다.

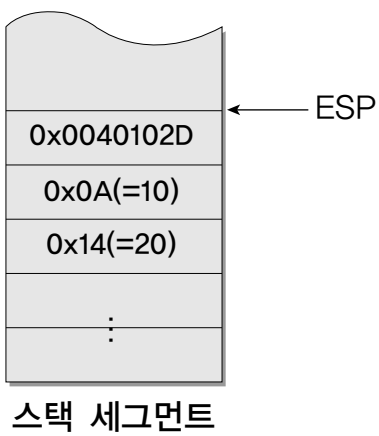
우선 제일 먼저 해야 할 일이 sum 함수를 호출하는 것입니다. 그리고 이 때 인자값으로 10과 20 두 개의 int 값을 전달해야 합니다. 앞에서 얘기한대로 이 두 인자는 함수가 호출되기 전에 스택에 저장해야 호출된 함수에서 스택 포인터 레지스터를 통해 엑세스할 수 있습니다. 그래서 컴파일러가 번역한 어셈블리 코드에서도 push라는 인스트럭션을 통해 각각 14h(=20)과 0Ah(=10)을 스택에 저장하고 있습니다.

이 push 인스트럭션은 주어진 값을 스택의 제일 끝(스택 포인터 레지스터가 가리키는 곳)에 저장하고 스택 포인터 레지스터 값을 4바이트(32비트 CPU 기준)만큼 감소시킵니다(여기서는 스택이

큰 주소에서부터 시작해서 점점 작은 주소 쪽으로 쌓여가도록 되어 있습니다. 따라서 스택에 값이 쌓일 때마다 주소는 작은 쪽으로 이동합니다).

그리고 call이라는 인스트럭션을 통해 sum 함수를 호출합니다. call 인스트럭션은 뒤에 주어진 주소대로 점프를 하되 스택에 현재 PC 레지스터의 값, 즉 바로 다음 클럭에 수행할 인스트럭션의 주소를 저장하죠. 물론 push와 마찬가지로 스택에 값을 저장하므로 스택 포인터 레지스터를 4바이트만큼 조정(감소)합니다.

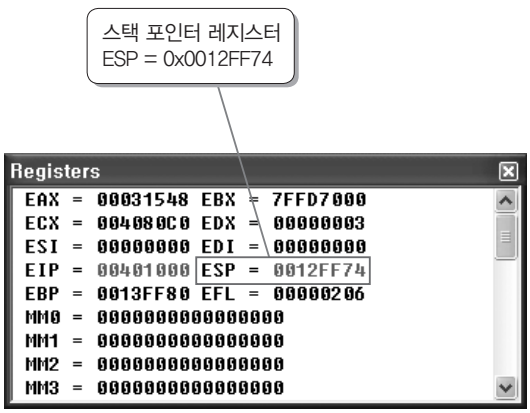
한편 우리가 사용하는 x86 CPU에서 스택 포인터로 ESP라는 레지스터가 사용됩니다. 따라서 call 인스트럭션까지 수행되고 난 후의 스택의 모습을 그림으로 나타내면 다음과 같습니다.



[그림 13-13] 첫 번째 call까지 수행된 직후의 스택 모습

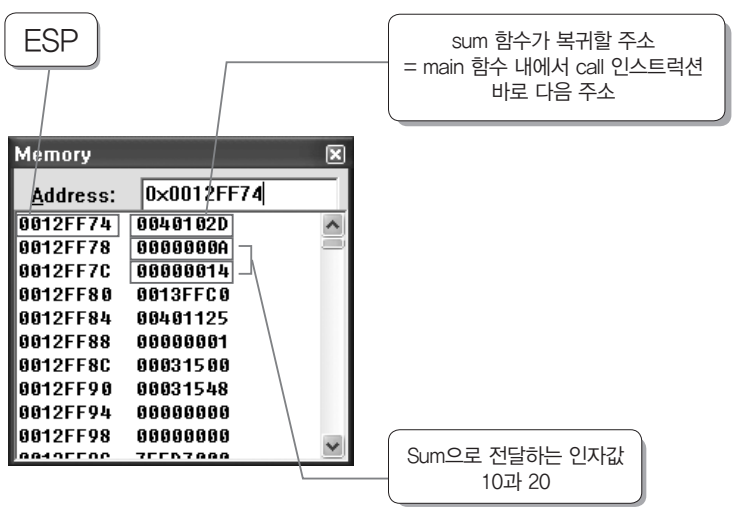
이제 이를 Visual C++의 디버깅 기능을 이용하여 실제로 확인해 보도록 하겠습니다.

우선 프로그램을 call 인스트럭션 직전까지 수행한 후 F11을 눌러 call 인스트럭션을 수행합니다. 그리고 Watch 창에 ESP를 적어 값을 직접 보거나, Alt + 5를 눌러 레지스터 창을 띄워 스택 포인터 레지스터 ESP의 값을 확인합니다. 여러분과 예제 PC의 환경이 틀려 다소 다를 수 있겠지만, 예제 PC에서 캡쳐한 화면은 다음과 같습니다.



[그림 13-14] 첫 번째 call까지 수행된 직후의 스택 포인터 레지스터 값

그림에서 보듯이 스택 포인터 레지스터인 ESP 레지스터의 값이 0x0012FF74입니다. 따라서 실제 스택 값이 어떻게 구성되어 있는지 살펴 보기 위해 Alt + 6을 눌러 메모리 창을 띄운 후 0x0012FF74번지를 살펴 보면 다음과 같습니다.



[그림 13-15] 첫 번째 call까지 수행된 직후의 스택 구성

앞에서 제가 도식적으로 설명한 그림 내용 그대로 현재 ESP가 가리키는 곳, 즉 스택의 제일 끝에는 sum 함수가 복귀할 주소값이 기록되어 있고 그 뒤로 인자값 10과 20이 각각 16진수 0x0A와 0x14로 나타나 있는 것을 알 수 있습니다.

이제 다시 앞의 상황으로 돌아가 call 인스트럭션의 수행이 끝났다고 가정하겠습니다. 다음으로 필요한 일이 어떤 것일까요? 그냥 바로 다음 할 일을 하면 되는 것일까요? 아닙니다. 아직 sum

함수를 호출하는 과정이 끝나지 않았죠. sum 함수를 호출할 때 인자와 복귀 주소를 스택에 저장합니다. 이 말은 스택 포인터 레지스터 ESP가 바뀌었다는 얘기입니다. 하지만 sum 함수의 수행이 완료되고 그 이후 코드를 수행하는 시점에서는 이 스택 포인터는 sum 함수를 호출하기 이전과 동일해야 합니다. 즉, 함수를 호출하기 이전의 상태로 스택을 돌려놓아야 한다는 것입니다. 그래야만 아무 일 없었다는 듯이 다음 코드를 수행할 수 있으니까요.

따라서 call 인스트럭션 바로 다음에 보면 아래와 같은 코드가 삽입되어 있습니다.

```
add  esp, 8
```

이는 여러분도 쉽게 알 수 있듯이 스택 포인터 레지스터 esp를 8바이트만큼 증가시키는 코드입니다. 그리고 여기서는 스택이 증가하는 방향이 큰 주소에서 작은 주소쪽이므로 esp를 증가시킨다는 것은 할당된 스택을 끝에서 8바이트만큼 줄인다는 의미입니다.

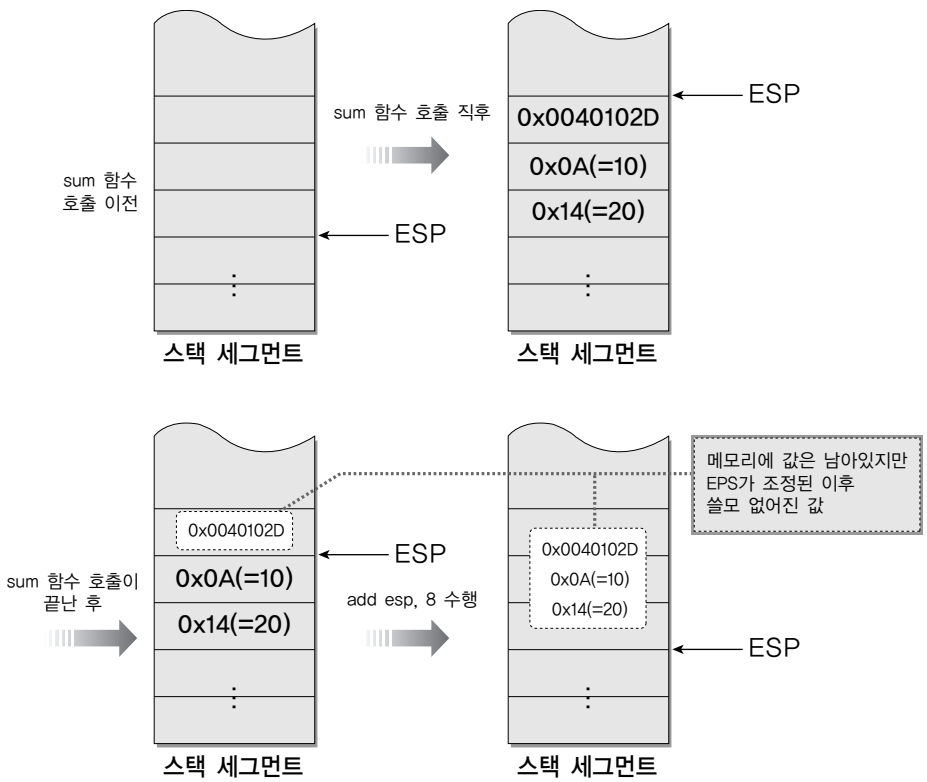
그런데 이상한 점은 sum 함수를 호출하기 이전에 두 개의 int 인자가 8바이트, 그리고 call 인스트럭션이 자동으로 스택에 저장하게 되는 복귀 주소가 4바이트 즉, 총 12바이트만큼 스택을 사용하였으므로 함수 호출이 끝나고 다시 없애줘야 하는 크기도 12바이트여야 할 것 같은데, 여기서는 8바이트만 조정하고 있습니다. 이는 sum 함수 내에서 사실 복귀할 때 미리 복귀 주소 4바이트만큼을 스택에서 해제, 즉 스택 포인터 레지스터 ESP를 4바이트만큼 증가시켜두기 때문입니다.

다음 그림은 sum 함수가 호출되기 이전부터 호출이 끝나고 마무리될 때까지의 스택의 모습을 나타낸 것입니다. 결국 첫 번째와 마지막 스택의 모습을 보면 ESP가 같은 곳을 가리키고 있으므로 호출 이전의 스택 모습을 그대로 유지하고 있습니다. 다만 ESP보다 위 쪽에 앞서 썼던 값이 남아 있긴 하지만 어차피 메모리는 어떤 값이라도 저장하고 있어야 하고, 저 값들은 어차피 더이상 참조되지 않는 값이므로 특별히 따로 지우거나 할 필요도 없는 쓰레기 값인 셈입니다.

따라서 최종적으로 sum 함수 호출 이후에 호출한 측에서는 복귀 주소 4바이트를 제외하고 전체 인자 크기 8바이트만큼 ESP를 조정해주면 되는 것입니다.

여기서 어떤 독자는 아마 이런 생각을 할 겁니다. '함수를 호출할 때 사용한 스택을 해지하는 작업을 왜, 일부는 호출된 함수 내부에서 하고 일부는 호출 이후에 호출한 측에서 처리할까?'라고 말이죠.

이는 바로 호출 규약(Calling Convention)이라고 부르는 표준의 문제입니다. 결국 장단이 있을 수는 있지만, 꼭 이렇게 해야 하는 절대적인 이유가 있는 것이 아니라, 정하기 나름의 문제인 것입니다. C에서는 함수 호출과 이 과정에서 스택을 처리하는 주체를 두고 몇 가지 방식으로 나뉘게 됩니다.



[그림 13-16] sum 함수 호출 이전에서 호출 이후까지 스택의 변화 모습

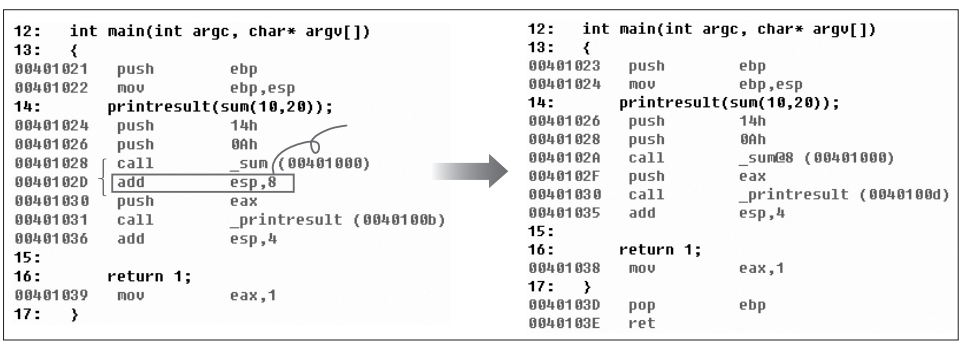
그 중 함수를 선언할 때 특별히 명시하지 않으면 __cdecl이라고 부르는 C의 기본 함수 호출 규약을 따르며 이 호출 방식에서는 기본적으로 인자가 차지하는 스택을 해지하는 책임이 호출한 측에 있습니다. 즉, 우리가 방금 살펴본 예처럼 sum 함수에 전달한 인자 값 8바이트를 스택에서 해지하는 것은 sum 함수가 리턴한 다음에 호출한 측에서 add esp,8을 통해 행하는 것입니다.

한편 __cdecl뿐 아니라, __stdcall이라고 하는 다른 호출 규약도 존재합니다. __stdcall에서는 인자에 대한 스택 해지까지 호출된 함수 내부에서 하며, 따라서 호출한 측에서는 함수 호출이 끝난 다음에도 별도의 작업을 할 필요가 없어 함수의 독립성이 뛰어나다는 장점이 있습니다. 또한 스택을 해지하는 코드, 가령 앞에서 add esp, 8 같은 코드가 매번 함수를 호출할 때마다 생성될 필요 없으므로 전체적인 프로그램의 코드 사이즈도 작아집니다.

이런 콜링 컨벤션은 함수를 선언할 때 함수의 리턴값과 이름 사이에 컨벤션 타입을 적어 결정할 수 있습니다. 가령 sum 함수가 __cdecl 타입이 아닌 __stdcall 타입의 콜링 컨벤션을 가지도록 하려면 sum 함수를 다음과 같이 수정하면 됩니다.

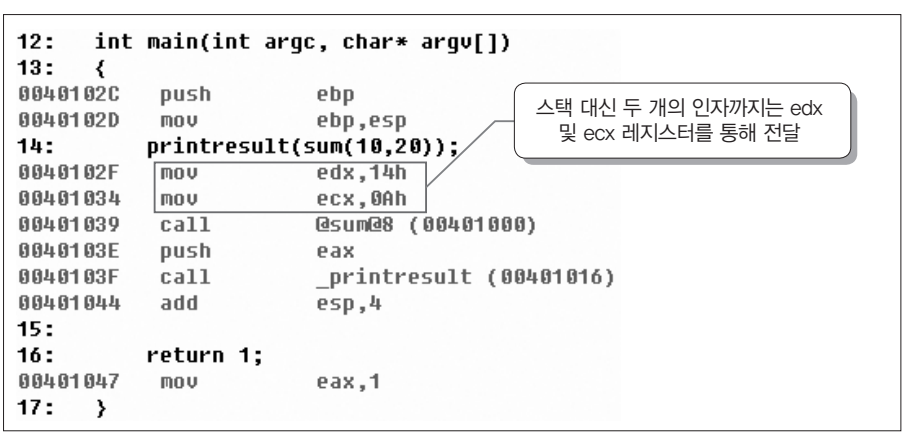
```
int __stdcall sum(int a,int b)
{
    return a + b;
}
```

int와 sum 사이에 __stdcall이라는 키워드를 삽입하고 재컴파일하면 이제 sum 함수의 호출은 __stdcall 규약을 따르도록 코드가 생성됩니다. 프로그램을 그냥 실행하는 입장에서는 아무런 차이가 없지만, 실제 내부 동작은 조금 달라지는 것이죠.



[그림 13-17] 콜링 컨벤션에 따른 코드 생성 차이 : __cdecl과 __stdcall의 차이점

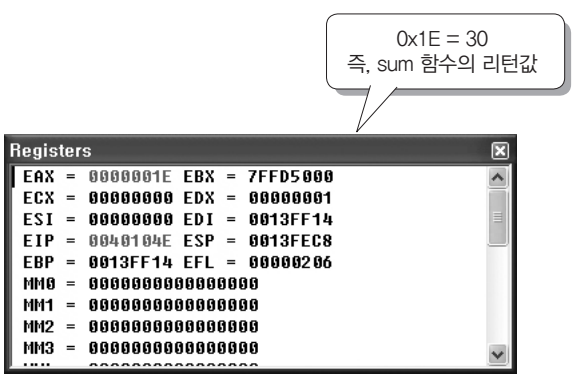
한편, 속도를 더 높이기 위해 두 개의 파라미터까지는 메모리를 사용하지 않고 레지스터를 사용해서 전달하는 __fastcall이라는 호출규약도 있습니다. 이 경우 함수 인자 두 개까지는 ecx와 edx라는 레지스터를 통해 전달하고 그 이상의 인자에 대해서는 __stdcall과 마찬가지 형태로 함수 내부에서 인자의 해지를 담당합니다. 아래 그림은 sum 함수를 __fastcall로 선언했을 때의 main 함수의 컴파일된 코드입니다.



[그림 13-18] 콜링 컨벤션에 따른 코드 생성 차이 : __fastcall 호출 규약을 사용시 레지스터를 통해 인자 전달

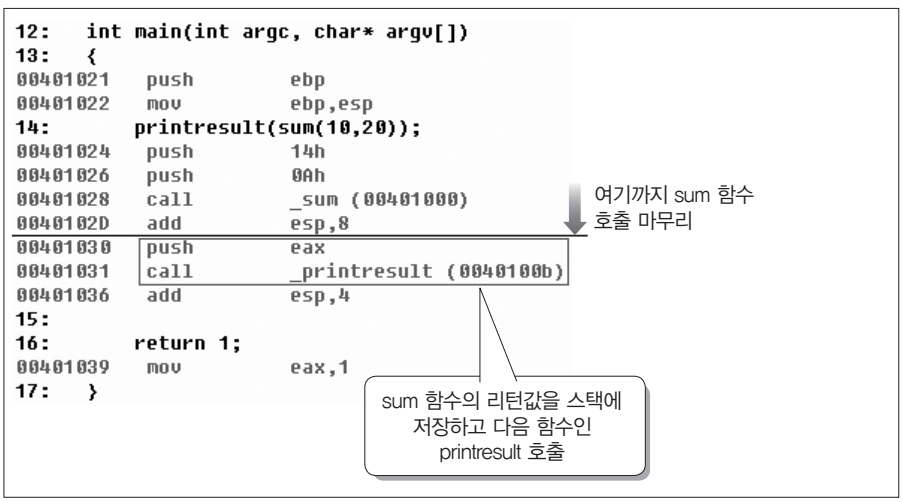
언급한대로 push 14h, push 0Ah가 사라지고, 대신 edx와 ecx 레지스터에 인자값을 넣고 있습니다. 그리고 sum 함수 내부에서는 이 두 레지스터를 통해 인자값을 얻는 것입니다. 만일 두 개보다 더 많은 인자가 있을 경우 3번째 인자부터는 앞에서와 마찬가지로 push를 통해 스택에 저장하는 것이죠. 그리고 이에 대한 해지는 __stdcall처럼 sum 함수 내부에서 하므로 main에는 add esp, … 같은 코드는 보이지 않는 것입니다.

이제 sum 함수의 호출이 끝나고 스택에 대한 해지도 끝났습니다. 남은 것은 sum 함수의 호출 결과를 얻어와 이를 다시 다음 함수인 printresult의 인자로 넘겨주는 것입니다. 그런데 앞에서 함수의 리턴값은 스택이 아닌 레지스터를 통해 전달된다고 하였죠. 그리고 그 레지스터가 바로 eax라는 레지스터입니다. 따라서 call sum 인스트럭션을 수행한 직후에 레지스터 윈도우를 띄워 eax 레지스터를 확인해보면 10 + 20한 결과값인 30이 들어가 있는 것을 확인할 수 있습니다.



[그림 13-19] sum 함수 리턴 직후의 레지스터 : eax에 리턴값 30이 들어 있다

이제 eax 레지스터의 값을 printresult의 인자로 전달해야 하는데, 어떻게 해야 할까요? 이미 여러분은 그 답을 알고 있을 것입니다. 앞에서 sum 함수를 호출할 때와 마찬가지로 스택에 저장해야겠죠. 따라서 sum 함수의 호출이 마무리 된 바로 직후, push eax라는 코드가 먼저 실행되고 이윽고 call 인스트럭션을 통해 printresult 함수를 호출하는 것을 볼 수 있습니다.



[그림 13-20] sum 함수의 리턴값이 저장된 eax 레지스터를 인자로 전달하기 위해 스택에 저장하고 printresult 함수 호출

그리고 역시 sum 함수 때와 마찬가지로 특별히 호출 규약을 지정하지 않았으므로 __cdecl 형태로 컴파일되어 인자에 대한 해지를 호출한 측에서 하고 있습니다(add esp, 4).

그리고 위 그림에서 가장 마지막 줄을 보면 main 함수가 return 1하는 코드가 아래와 같이 컴파일되어 있습니다.

```
mov  eax,1
```

즉, 앞서 여러번 얘기한 것처럼 main도 하나의 함수일 뿐이고 다른 함수들과 마찬가지로 리턴값을 eax 레지스터에 저장하는 것입니다.

이제 그럼 sum 함수 내부가 어떻게 번역되었는지 한번 살짝 들여다 봅시다.

```
3:    int sum(int a,int b)
4:    {
00401000   push        ebp
00401001   mov         ebp,esp
5:        return a + b;
00401003   mov         eax,dword ptr [ebp+8]
00401006   add         eax,dword ptr [ebp+0Ch]
6:    }
00401009   pop         ebp
0040100A   ret
```

[그림 13-21] sum 함수의 컴파일된 어셈블리 코드

sum 함수도 main에서와 마찬가지로 가장 먼저 나오는 것이 아래 두 줄입니다.

```
push ebp
mov  ebp,esp
```

앞에서 이 부분에 대한 설명은 뒤로 미뤘었는데, 이제 이 부분에 대한 얘기를 좀 해보겠습니다. 코드 자체의 의미는 단순합니다. ebp 레지스터를 스택에 저장하고 다시 이 ebp 레지스터에 esp 레지스터의 내용을 복사하는 것입니다.

그리고 끝에서 두 번째 줄에 보면 다음과 같은 코드가 있습니다.

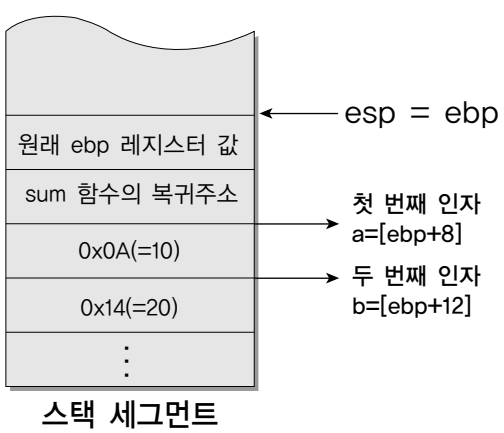
```
pop  ebp
```

이 pop이라는 인스트럭션은 push 인스트럭션과 정반대의 일을 하는데, 스택의 제일 끝에서 4바이트만큼 데이터를 꺼내 와서 지정된 레지스터에 저장하는 역할을 합니다. 그리고 push와 반대로 스택 포인터를 스택이 줄어드는 방향으로 4바이트만큼 조정합니다. 즉, 여기서는 ebp 레지스터에 스택 끝에 있는 값을 가져오는 셈입니다.

그런데 이 때 스택 끝에 있는 값이란 무엇일까요? 바로 다름 아닌, 앞에서 push ebp로 저장했던, 원래 ebp 레지스터에 있던 값이었습니다. 그리고 원래 ebp 레지스터에 있던 값이란 것은, 이 함수와는 상관 없이 함수 호출 이전에 결정되어 있던 값입니다. 왜냐하면 함수 가장 첫 머리에 ebp 레지스터를 저장하므로 함수 내부에서는 ebp 레지스터를 건드리기도 전에 이전의 값 그대로를 저장했다가 함수가 끝날 때 다시 돌려 놓는 셈입니다. 따라서 이 함수를 호출하는 입장에서는 호출 전이나 후에 ebp 레지스터에 어떠한 변화도 없는 셈이죠.

그리고 이 ebp 레지스터는 코드 중간에 보면 ebp + 8 같은 형태로 사용하고 있습니다. 이는 바로 호출측에서 스택에 저장한 인자값을 엑세스하기 위해서인데, 이미 ebp 레지스터엔 스택 포인터 레지스터 esp를 그대로 복사해 놓았기 때문에 esp 대신 ebp를 사용해서 엑세스할 수 있습니다.

push ebp 직후의 스택의 모습을 보면 다음과 같습니다.



[그림 13-22] sum 함수 내부에서 push ebp 직후의 스택 모습

sum 함수를 호출하기 이전에 20, 10, 복귀 주소 순으로 스택에 저장했고 sum 함수 내부에 들어와 제일 먼저 push ebp로 ebp 값을 저장했으므로 위와 같이 스택이 구성될 것입니다.

그리고 sum 함수의 주 목적인 두 인자의 합은 다음과 같은 코드로 구성되어 있습니다.

```
mov         eax,dword ptr [ebp+8]
add         eax,dword ptr [ebp+0Ch]
```

그림에서 보면 첫 번째 인자는 ebp + 8번지에 저장되어 있고, 두 번째 인자는 그 다음인 ebp + 12 = ebp + 0x0C에 저장되어 있으므로 위 어셈블리 코드에서도 ebp + 8번지의 값을 eax 레지스터로 옮겨와 이 값을 ebp + 12번지에 있는 값이랑 덧셈합니다. 그리고 이 덧셈 결과는 다시 eax 레지스터에 저장됩니다. 어차피 리턴값은 eax 레지스터에 저장되어야 하므로 결국 sum 함수는 이제 할 일을 다 한 셈입니다.

그리고 sum 함수는 ebp 레지스터를 복구하고(pop ebp) 함수에서 ret 인스트럭션으로 리턴하죠. 이 ret 인스트럭션은 스택의 제일 끝에 있는 값을 복귀 주소로 삼아 그 주소로 점프하고 동시에 스택 포인터 레지스터 esp 값을 4바이트만큼 스택을 줄이는 방향으로 조정합니다.

위 스택 그림에서 보면 복귀 주소는 원래 ebp 값 바로 다음에 있었는데, ret 바로 이전에 pop 인스트럭션으로 원래 ebp 값을 해지하였으므로 이제 스택 제일 끝에는 복귀 주소가 있는 셈입니다. 따라서 ret는 무사히 복귀 주소를 스택 끝에서 읽어와 그 주소로 점프할 수 있습니다.

결국 지금까지의 과정을 보면 ebp 레지스터는 스택 포인터 레지스터 esp 대신 인자값을 엑세스하기 위해 사용된 것밖에 없습니다. 이때 '왜 그럼 직접 esp 레지스터를 사용하지 않고 ebp를 사용할까?'하는 의문이 들 수 있을 것입니다. 이는 컴파일러의 편의성을 위해서라고 할 수 있습니다. 스택 포인터 레지스터 esp는 코드 중간에 push한다든지 pop한다든지 하면서 자동으로 수시로 변할 여지가 있습니다.

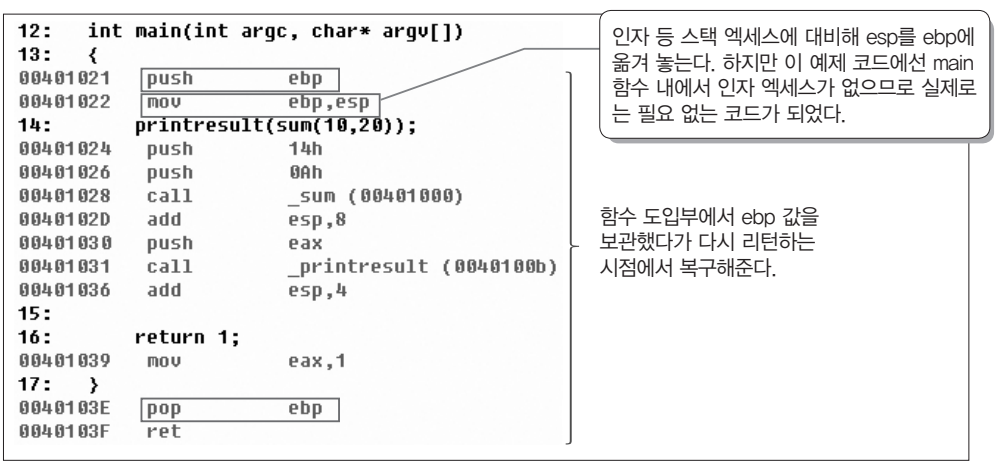
따라서 push라는 인스트럭션이 중간에 있으면 그 이전과 이후에 esp를 기준으로 엑세스할 때 상대적인 오프셋 값(위에서 ebp + 8과 같은 값)이 변한다는 문제가 있습니다. 사실 이 부분도 컴파일러에 따라서는 일일이 다 고려해서 오프셋 값을 생성하는 경우도 있지만, ebp 같은 별도의 레지스터에 초반 esp 값을 복사해놓고 이후 ebp 레지스터의 값을 기준으로 스택을 엑세스하면, esp가 아무리 변하더라도 고정된 오프셋 값으로 스택을 엑세스할 수 있습니다. 결국 코드를 생성하는 입장에서도, 또 개발자 입장에서도 훨씬 이해하기 쉬워지죠.

그래서 VC++ 컴파일러는 스택 프레임을 구성하면서 ebp 레지스터에 esp 값을 옮겨와 사용하되, 호출한 함수(caller)측에서도 똑같은 방식으로 ebp 레지스터를 사용하고 있으므로 호출받은 함수 내부에서 미리 ebp 레지스터 값을 스택에 보관했다가 함수가 리턴하기 바로 직전에 이를 복구합

니다. 때문에 호출한 측에서는 여전히 같은 ebp 값으로 작업을 계속할 수 있는 것이죠.

이는 메인 함수를 살펴봐도 역시 마찬가지입니다. 다음 그림을 보면 메인 함수도 도입부에서 ebp 레지스터를 스택에 보관하고 esp 값을 복사한 후 마지막에 다시 ebp를 복구하는 기계적인 작업을 그대로 하고 있습니다.

그런데 가만히 잘 살펴보면 main 함수 내에서는 ebp 레지스터를 통해 인자를 엑세스하는 부분이 없습니다. 따라서 방금 얘기한 작업은 아무 쓸데 없이 괜히 코드 낭비만 하는 셈이죠. 따라서 실제로 컴파일러는 최적화 과정을 통해 이런 불필요한 코드를 제거하기도 합니다. 여기서는 제가 여러분에게 소개하기 위해 일부러 최적화(Optimization) 옵션을 모두 끄고 컴파일을 했기 때문에 생성된 것입니다.



[그림 13-23] main 함수 내부에서 ebp 레지스터 관련 코드 : 여기서는 ebp 레지스터를 활용하지 않으므로 결국 쓸데없는 코드이다 → 컴파일러 최적화 옵션으로 해결 가능

Vitamin Quiz

스택의 증가 방향

스택 프레임을 설명하면서 눈치 빠른 독자들은 아마 이런 의문을 가졌을 것입니다. '스택에 값을 쌓아갈 때 왜 거꾸로 쌓는 것인가'하고 말이죠. 사실 거꾸로라는 표현도 정확한 건 아니지만, 일반적으로 메모리에 데이터를 저장할 때 작은 주소에서 큰 주소로 증가시켜가면서 저장하는 것이 일반적입니다. 그런데 스택 역시 어차피 메모리를 논리적인 사용 용도에 따라 나눈 것뿐인데, 왜 지금까지 든 예제에서는 모두 큰 주소에서 작은 주소로 감소해가며 저장했을까요?

Section 04

지역 변수

지금까지 여러분은 함수 호출에서 인자의 전달과 리턴값을 위해 스택 프레임이라는 것에 대해 살펴보았습니다.

그런데 이 스택 프레임이라는 것이 사실은 이 용도 외에도 아주 중요한 다른 용도로 사용되고 있습니다. 바로 함수 내부에서 사용하는 지역 변수(Local Variable)를 위해서입니다.

함수라는 것은 어디서 불릴지 모르는 것이고, 그 자체가 하나의 모듈로서 독립성을 가지고 동작해야 하는 경우가 대부분입니다. 따라서 함수 내부에서 전역 변수를 사용하게 되면 이런 독립성이 떨어지므로 좋은 방법이라 할 수 없습니다. 가령 함수는 자기 자신을 스스로 호출 하는 재귀호출(Recursive Call)뿐 아니라 여러 형태로 해당 함수가 끝나기 전에 다시 함수가 호출되는(이를 재진입이라고 합니다) 경우가 종종 발생할 수 있으므로 이에 대한 대비책이 필요하죠. 즉, 함수가 중첩되어 호출되더라도 앞 뒤 작업간의 독립성을 유지할 방법 말입니다.

이런 독립성을 유지하는 가장 전형적인 방법이 바로 매 함수 호출 때마다 함수 내부에서 사용하는 변수를 몽땅 메모리에 새로 할당받는 방법입니다. 이렇게 되면 함수가 아무리 중복되어 호출되더라도 각 함수마다 서로 다른 메모리에서 작업하므로 함수간의 독립성을 유지할 수 있죠. 문제는 '어떻게'입니다.

우리가 앞에서 배운 스택 프레임은 사실 인자의 전달뿐 아니라 이런 지역 변수의 할당을 위해서도 최적의 구조를 제공합니다. 또 다시 백문불여일견-예제 코드를 살펴보도록 하겠습니다.

```
#include <stdio.h>

int triple_sum(int a,int b)
{
   int nTripleA = a * 3;
   int nTripleB = b * 3;

   return nTripleA+ nTripleB;
}

void printresult(int c)
{
   printf("(10*3) + (20*3) = %d",c);
}
int main(int argc, char* argv[])
{
```

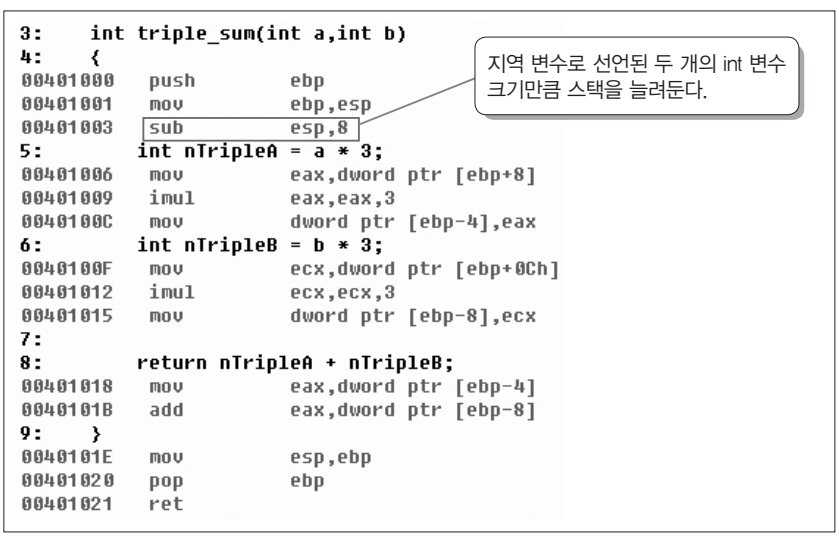
```
    printresult(triple_sum(10,20));

    return 1;
}
```

이번에는 설명을 위해 sum 함수 내부에 nTripleA와 nTripleB라는 두 개의 변수를 만들고 이 변수에 각각 넘겨 받은 인자 a, b에 3씩 곱해 넣었습니다.

그리고 이렇게 3이 곱해진 두 수의 합을 리턴하고 있습니다.

어차피 다른 부분은 그대로이므로 바뀐 triple_sum 함수만 어떻게 컴파일되었는지 살펴볼까요?




```
3:    int triple_sum(int a,int b)
4:    {
00401000   push        ebp
00401001   mov         ebp,esp
00401003   sub         esp,8
5:        int nTripleA = a * 3;
00401006   mov         eax,dword ptr [ebp+8]
00401009   imul        eax,eax,3
0040100C   mov         dword ptr [ebp-4],eax
6:        int nTripleB = b * 3;
0040100F   mov         ecx,dword ptr [ebp+0Ch]
00401012   imul        ecx,ecx,3
00401015   mov         dword ptr [ebp-8],ecx
7:
8:        return nTripleA + nTripleB;
00401018   mov         eax,dword ptr [ebp-4]
0040101B   add         eax,dword ptr [ebp-8]
9:    }
0040101E   mov         esp,ebp
00401020   pop         ebp
00401021   ret
```

[그림 13-24] 두 개의 지역 변수가 사용된 triple_sum 함수의 어셈블리 코드

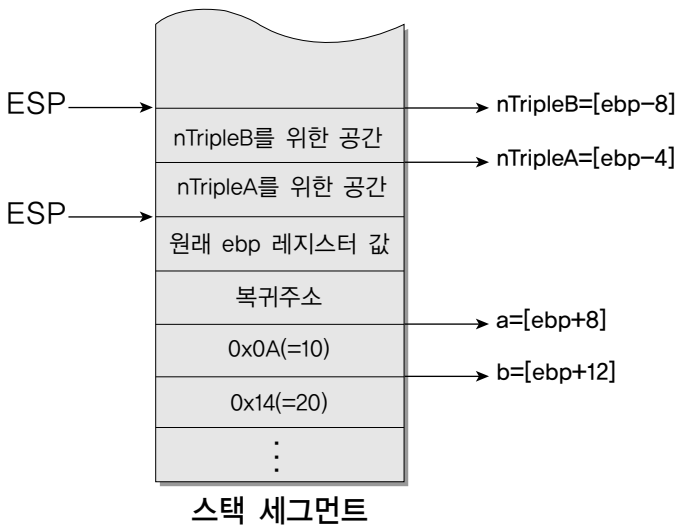
우선 제일 처음 ebp 레지스터를 스택에 저장하고 esp 레지스터를 복사하는 부분은 앞에서와 동일합니다. 그리고 가장 처음 눈에 띄는 다른 부분은 push나 pop 같은 스택 인스트럭션이 아니라, 직접 뺄셈을 하는 sub 인스트럭션으로 esp 레지스터 값을 8 줄이는 부분입니다.

다시 한번 상기해보면 우리가 지금까지 봐 왔던 예제, 즉 PC에서의 스택은 메모리의 뒤 쪽, 주소가 큰 부분에서 앞 쪽(주소가 작은 쪽)으로 커지는 모양이었습니다. 따라서 스택 포인터 레지스터 esp를 8바이트 만큼 줄인다는 것은 스택을 8바이트 만큼 키운다는 얘기와 같습니다.

그리고 벌써 눈치 챈 독자들도 있겠지만, 이 늘어난 8바이트의 스택 공간이 바로 triple_sum 함수 내에 선언된 두 개의 지역 변수를 위해 할당된 공간인 것입니다.

컴파일러는 이미 컴파일하면서 triple_sum 함수 내에서 두 개의 int 지역변수가 사용된다는 사실

을 알고 있고, 따라서 각각 4바이트씩 총 8바이트 만큼의 스택 공간을 이 두 변수에 할당하기 위해 스택 포인터 레지스터 esp를 강제로 조정한 것입니다. 따라서 sub esp, 8 코드가 수행되고 난 직후의 스택 모습은 다음과 같을 것입니다.



[그림 13-25] triple_sum 함수 내에서 sub esp, 8 실행 직후의 스택 모습

여기서 주의 깊게 살펴 볼 부분은 이번에는 이전 예제들과는 다르게 esp와 ebp 레지스터의 값이 서로 다르다는 점입니다. 왜냐하면 ebp 레지스터에 esp 레지스터를 복사하고 난 이후에 esp를 강제로 8바이트만큼 조정한 것이기 때문이죠.

앞에서 설명하였듯이 esp를 ebp에 복사하는 이유는 지금과 같이 스택 포인터가 변하게 되는 경우 그 때마다 엑세스하고자 하는 곳에 대한 오프셋 값이 변하기 때문이라고 했습니다. 그리고 esp는 push, pop 등 모든 스택 인스트럭션이 기준으로 삼는 스택 포인터 레지스터이므로 항상 스택의 끝을 가리키고 있어야 합니다. 표현이 좀 헷갈릴 수도 있는데, esp 레지스터가 스택의 끝을 가리키고 있는 것이 아니라 esp 레지스터가 가리키고 있는 곳이 스택의 끝이라고 하는 표현이 더 맞을 수도 있겠군요.

어쨌건, 이제 esp는 다른 필요한 공간을 스택에 다 할당한 후에 지역 변수를 위해 8바이트만큼 조정되었으므로 이후 triple_sum 함수가 리턴하기 전에 중간에 다른 함수가 호출된다든가 하더라도 항상 esp를 기준으로 그 위 쪽에 다시 스택 프레임이 구성되므로 지역 변수를 위한 공간은 침범받지 않는 것입니다. 그리고 이 공간은 ebp 레지스터를 기준으로 이번에는 마이너스 오프셋 값을 가지고 엑세스할 수 있습니다. 즉, nTripleA 변수는 ebp – 4 번지로 엑세스할 수 있고, nTripleB는 ebp – 8번지로 엑세스할 수 있는 것이죠.

실제로 코드를 살펴보면, nTripleA 변수에 인자로 넘겨 받은 a에 3을 곱해서 대입하는 부분이 다음과 같이 번역되어 있습니다.

```
mov         eax,dword ptr [ebp+8]
imul        eax,eax,3
mov         dword ptr [ebp-4],eax
```

그림에서 보듯이 인자 a가 위치한 주소는 ebp + 8번지이고 이 값을 eax 레지스터로 옮겨와(mov) 여기에 3을 곱해서(imul) 다시 eax에 대입하게 됩니다. 그리고 이 결과값을 다시 ebp - 4번지, 즉 nTripleA를 위해 확보한 공간에 옮깁니다. 결국 우리가 C에서 기술한대로 동작이 되는 것이죠.

다음 nTripleB에 대해서도 비슷한 작업을 하고 마지막에 두 변수 값을 더해 리턴하는 부분에서 역시 ebp - 4 번지와 ebp - 8 번지 값을 더해서 eax에 저장하도록 번역되어 있습니다. 왜냐하면 앞서 말했듯이 리턴값은 eax 레지스터에 저장되기 때문이죠.

다음으로 이전의 예제 코드와 다른 점이 눈에 띄는데, 바로 끝에서 세 번째 줄에 코드입니다.

```
mov         esp,ebp
```

위 코드는 esp에 다시 ebp 값을 넣어주는 코드인데, 바로 이 함수 내부에서 사용했던 지역 변수용 스택을 해지해 주는 부분입니다. 이를 통해 이전에 지역 변수가 없었던 예제의 함수와 같은 형태로 함수의 리턴 준비를 할 수 있는 것입니다. 바로 pop ebp로 오리지널 ebp 값을 복원시키고 ret 인스트럭션으로 이제 스택의 제일 윗 부분에 있게 된 복귀 주소를 참조해 호출 코드로 되돌아가는 것이죠. 그리고 호출측에서는 앞에서처럼 다시 esp 값을 조정해서 인자를 위한 스택 부분을 해지할 것입니다.

이와 같이 스택 프레임에 지역 변수를 위한 공간을 마련하면, 이 함수 내부에서 또 다시 자기 자신을 부른다 하더라도 결국 현재 스택 위에 또 다른 스택 프레임이 구성되고 그 안에 새로운 지역 변수 공간이 확보되기 때문에 이전 변수값에 대해선 아무런 영향을 미치지 않는 것입니다.

Vitamin Info

호출규약

앞에서 호출 규약(Calling Convention)에 대한 얘기를 잠깐 하고 넘어갔습니다. __cdecl이나 __stdcall 같은 것들이 있는데, 왜 하나로 통일하지 않고 이렇게 서로 다른 규약이 있는 것일까요? 그리고 서로 어떤 장단점이 있을까요?

호출 규약(Calling Convention)은 간략히 얘기해 함수를 호출하는 과정에서 인자를 전달하기 위해 스택 프레임이라는 구조를 사용하는데, 인자를 위해 확보한 스택 공간을 누가 해지해 주는가 하는 부분에 대한 정의입니다. 즉, 이를 호출한 쪽(Caller)에서 하느냐, 아니면 호출된 함수(Callee)가 책임지는가 하는 문제인 것이죠.

이러한 스택 공간의 해지는 add esp, 8 같이 스택 포인터 레지스터를 인자 크기만큼 강제로 조정하는 코드를 삽입함으로써 이루어집니다.

C에서는 기본적으로 특별한 구문없이 함수를 선언하게 되면, 모두 __cdecl형으로 함수가 선언되고 따라서 저런 해지 코드가 호출한 측, 즉 함수를 call하고 그 다음에 삽입합니다. 사실 별거 아닌 거 같지만, 프로그램은 함수 호출로 이루어져 있다고 해도 과언이 아닐 정도로 많은 함수를 호출하고 있습니다. 만일 이런 함수 호출이 100군데 있다면 add esp, 8이라는 코드가 100군데 모두 들어가야 하는 셈입니다. 그만큼 코드 크기도 커지는 것이고, 만일 여러분이 어셈블리로 직접 코딩한다면 서브루틴 호출(함수 호출) Eoakke 하나하나 전달 인자 크기를 계산하여 코드를 삽입해야 합니다.

반면 __stdcall 형으로 선언한 함수에서는 해지 코드가 함수 내부에 리턴하기 바로 직전에 들어있습니다. 따라서 이 함수를 100군데서 부르든 1000군데서 부르든 상관 없이 해지 코드는 단 하나로 족한 것이고 그만큼 코드 사이즈도 컴팩트해집니다. 그 뿐 아니라, 개념적으로도 함수에 관련된 부분이므로 함수 내부에서 처리하고 외부에서는 이에 대해 신경쓰지 않게 해 함수의 독립성을 높일 수 있다는 장점이 있습니다.

그렇다면 왜 이런 장점에도 불구하고 __cdecl 형이 존재하는 것일까요? 여기엔 두 가지 이유가 있습니다. 하나는 __stdcall이라는 것은 사실, 처음부터 C에 존재하는 형이 아니었습니다. 원래 이는 파스칼에서 사용되던 함수 호출 규약이었는데, 여러분이 사용하는 핸드폰도 회사에 따라 사용 주파수나 방식이 다르듯이 회사에 따라 서로 다른 규약을 사용했던 것입니다.

그런데 C를 만든 사람들이 __stdcall과 같은 형태가 더 좋은 걸 몰라서 그런 식으로 디자인하지 않았을까요? 아닙니다. C에서는 __stdcall 방식을 사용할 수 없는 결정적인 이유가 있습니다. 바로 가변 인자라고 하는 것입니다. 여러분이 아무 생각 없이 무심코 사용하는 printf라는 함수가 바로 이런 가변 인자를 사용하는 대표적인 함수입니다.

항상 여러분은 함수를 선언할 때 그 인자 수와 형(形)을 명시적으로 기록해야 했습니다. 그리고 그 포맷에 맞추어 함수를 호출해야 했죠. 그런데 printf는 이런 인자의 제약 없

이 뒤에 무한히 많은 인자를 계속 넣을 수 있습니다. printf의 프로토타입을 살펴보면 다음과 같이 되어 있습니다.

```
int __cdecl printf(const char *, ...);
```

이 중 마지막의 …으로 표시되는 인자가 바로 개수에 상관 없이 받아 들일 수 있는 가변 인자입니다.

하지만 이런 가변 인자가 있는 함수라고 해서 호출하는 데 있어 다른 함수와 별 다른 점이 있는 것은 아닙니다. 사실 함수의 프로토타입이라는 것은 컴파일러가 문법적으로 함수 호출에 대한 부분을 체크하기 위한 것이지, 어셈블리로 번역되고 나면 인자가 한 개든, 열 개든 중요한 것이 아닙니다. 어차피 함수 호출 시 기재된 개수만큼 push 인스트럭션을 통해 스택에 저장을 하고 함수를 호출하면 되는 것이니까요. 그런데 문제는 함수 수행이 끝나고 인자 부분을 스택에서 해지할 때 일어납니다. 만일 __cdecl 형으로 선언된 함수일 경우, 이 인자 부분에 대한 해지를 호출한 곳에서 매번 해주므로 printf 함수를 부를 때마다 인자 개수를 매번 다르게 주어도 아무 상관 없습니다. 왜냐하면 어차피 printf 함수를 호출한 곳마다 add esp, xx 같은 해지 코드가 들어가고 각 호출 때 사용된 인자 개수만큼 컴파일러가 알아서 xx 부분을 결정하면 되기 때문이죠.

반면 __stdcall로 선언된 경우는 얘기가 좀 다릅니다. 이 경우 인자 해지 코드가 함수 내부로 들어가야 하는데, printf 같이 가변 인자가 사용되는 경우 과연 몇 개의 인자가 전달되었는지 함수 내부에서는 알 방법이 없기 때문입니다. 따라서 획일적으로 add esp, 8처럼 정해진 상수값으로 스택 포인터 레지스터를 조정해 줄 수 없으므로 __stdcall로는 가변 인자를 사용할 수 없는 것이죠.

그렇다면 실제로 실험을 한번 해 볼까요? 함수를 __stdcall 형태로 선언하고 가변 인자를 사용하지 않았을 경우랑 가변인자를 사용하였을 경우 어떻게 컴파일되는지 살펴보겠습니다.

```
int __stdcall testfunc(int a,int b)
{
    return a + b;
}
```

위와 같이 선언된 함수를 컴파일 하고,이를 호출하는 측을 살펴보면 아래 그림과 같습니다.

```
19:   int main(int argc, char* argv[])
20:   {
0040103F   push        ebp
00401040   mov         ebp,esp
21:       testfunc(10,20);
00401042   push        14h
00401044   push        0Ah
00401046   call        _testfunc@8 (00401000)
22:
23:       return 1;
0040104B   mov         eax,1
24:   }
00401050   pop         ebp
00401051   ret
```

[그림 13-26] __stdcall 방식으로 함수를 호출하는 코드

위 그림에서 보듯이 call 이후에 add esp, 8 같은 코드가 없습니다. 반면 testfunc 함수 내부를 살펴보면 마지막에 ret 인스트럭션을 부를 때 8이라는 숫자와 함께 부릅니다.

```
3:    int __stdcall testfunc(int a,int b)
4:    {
00401000   push        ebp
00401001   mov         ebp,esp
5:        return a + b;
00401003   mov         eax,dword ptr [ebp+8]
00401006   add         eax,dword ptr [ebp+0Ch]
6:    }
00401009   pop         ebp
0040100A   ret         8
```

[그림 13-27] __stdcall 함수 내부에서 리턴하는 코드

이 ret 인스트럭션을 그냥 부르면 단순히 스택의 제일 위에서 복귀 주소를 읽어 점프하고 스택 포인터 레지스터를 4바이트만큼 조정할 뿐이지만, 뒤에 저렇게 인자가 있는 경우 그 크기만큼 스택 포인터 레지스터를 추가로 조정합니다. 즉, 앞에서 add esp, 8 하는 것과 동일한 효과가 있는 셈입니다.

한편 이번엔 testfunc 끝에 세 번째 인자를 가변인자로 추가해보겠습니다.

```
int __stdcall testfunc(int a,int b,...)
{
    return a + b;
}
```

그리고 이를 호출하는 코드를 살펴보면 다음과 같습니다.

```
19:   int main(int argc, char* argv[])
20:   {
00401043   push        ebp
00401044   mov         ebp,esp
21:       testfunc(10,20,30);
00401046   push        1Eh
00401048   push        14h
0040104A   push        0Ah
0040104C   call        _testfunc (00401000)
00401051   add         esp,0Ch
22:
23:       return 1;
00401054   mov         eax,1
24:   }
00401059   pop         ebp
0040105A   ret
```

[그림 13-28] __cdecl 형태의 함수를 호출하는 코드

분명 10, 20, 30 세 개의 인자를 전달하므로 push 인스트럭션이 세 번 나오고, testfunc 함수를 call하는 인스트럭션이 있습니다. 그리고 분명 __stdcall로 선언했음에도 불구하고 아까와는 다르게 call 인스트럭션 다음에 add esp, 0Ch라는 코드가 있습니다. 즉, 3개의 인자 크기 12바이트만큼 esp 레지스터를 조정해서 인자 부분에 대한 공간을 스택에서 해지하는 코드입니다.

```
19:   int main(int argc, char* argv[])
20:   {
00401043   push        ebp
00401044   mov         ebp,esp
21:       testfunc(10,20,30);
00401046   push        1Eh
00401048   push        14h
0040104A   push        0Ah
0040104C   call        _testfunc (00401000)
00401051   add         esp,0Ch
22:
23:       return 1;
00401054   mov         eax,1
24:   }
00401059   pop         ebp
0040105A   ret
```

[그림 13-29] __cdecl 형태로 변경된 함수 내부의 리턴 코드

testfunc 자체를 살펴 보면, 아까와는 다르게 ret에 8이나 0Ch 같은 인자가 붙어 있지 않고 __cdecl로 선언된 함수처럼 되어 있습니다. 즉, 실제 함수를 선언할 때는 __stdcall로 선언했지만, 가변인자가 사용됨으로 인해 __stdcall 형태로는 구현할 수 없으므로 컴파일러가 강제로 __cdecl 형태로 코드를 생성한 것입니다.

마지막으로 __fastcall은 최대 두 개까지의 인자는 메모리 엑세스를 하지 않고 바로 레지스터로 전달하게 해서 더욱 속도를 높이고자 만들어진 호출 규약입니다. 이런 경우 앞에서부터 두 개까지는 레지스터 ecx, edx를 사용해서 전달되므로 레지스터에 비해 훨씬 느린 메모리 엑세스를 하지 않아도 된다는 장점이 있긴 합니다만, CPU에 따라 여분의 레지스터가 없을 수도 있으며, 컴파일러가 레지스터를 어떻게 활용하느냐에 따라 구현이 불가능할 수도 있으므로 모든 컴파일러가 지원하지는 않습니다.

참고로, 윈도우에서 사용하는 API 라이브러리 등은 모두 __stdcall 형태의 호출 규약을 사용해서 만들어져 있습니다. 따라서 C 컴파일러에서 선언된 윈도우즈 API 함수들의 헤더는 모두 WINAPI라는 키워드를 포함하고 있습니다. 이 WINAPI 키워드가 바로 다름 아닌 __stdcall을 #define을 사용하여 재정의 해놓은 것뿐입니다.

이것만은 알고 갑시다.

1. ( ) 이란 함수가 호출될 때마다 해당 함수가 실행되기 위해 스택에 할당 받는 메모리 덩어리를 일컫는다.

2. 다음 중 스택에 저장되는 것들을 고르시오

   (a) 전역 변수

   (b) 함수 호출 인자

   (c) 복귀 주소

   (d) 상수

   (e) 지역 변수

3. 인텔사의 80x86 계열 CPU에는 스택을 활용한 함수 호출을 하기 위해 ESP와 EBP라는 레지스터를 제공하고 있다. 두 레지스터의 차이점에 대해 설명하시오

4. __cdecl 호출규약(Calling Convention)을 사용하는 함수 호출은 함수 호출시 사용된 스택프레임의 해지를 ( )에서 담당하게 된다. 이러한 방식의 특징은 ( ) 인자 형태의 함수 호출이 가능해지지만 프로그램 전체 코드의 크기가 ( ) 된다.