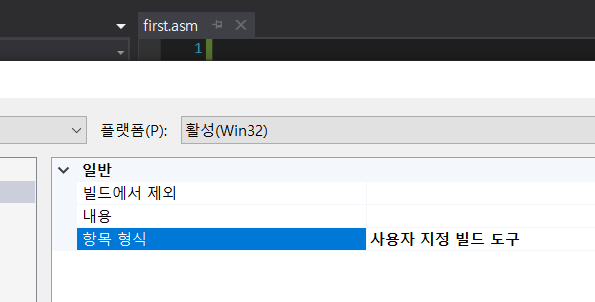
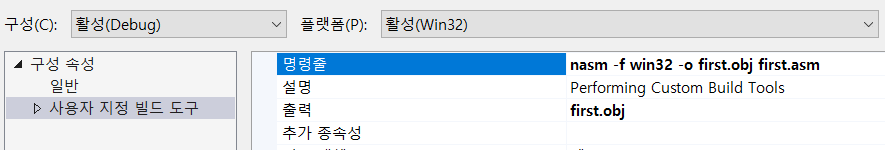


인텔 x86의 경우, eax 레지스터를 사용한다.

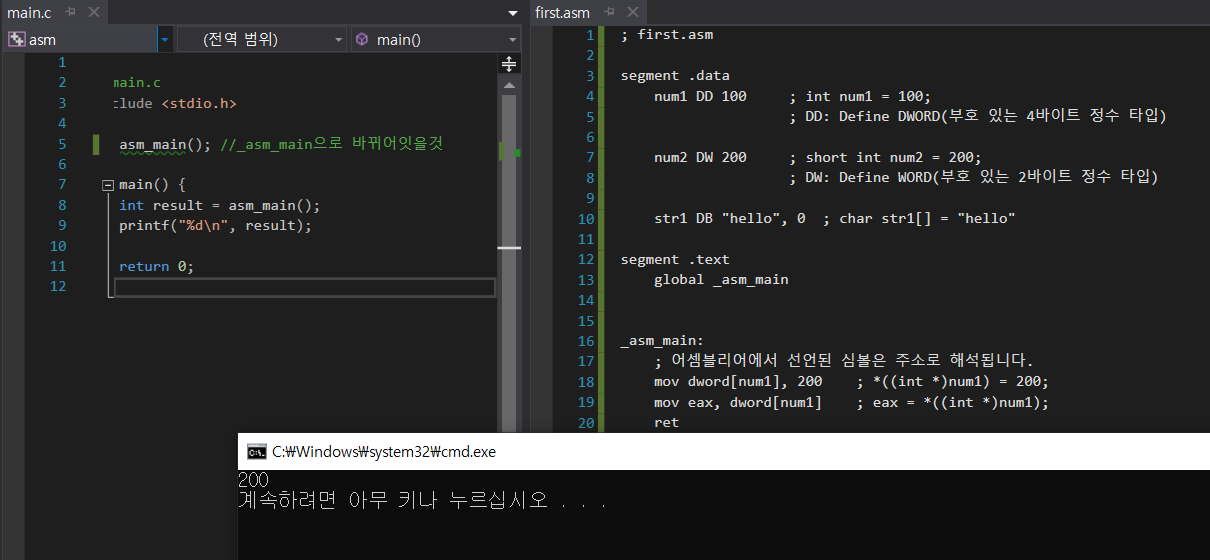
main.c/ first.asm은 목적파일로 각각이 생기며 그후 링커가 하나로 만들어준다.

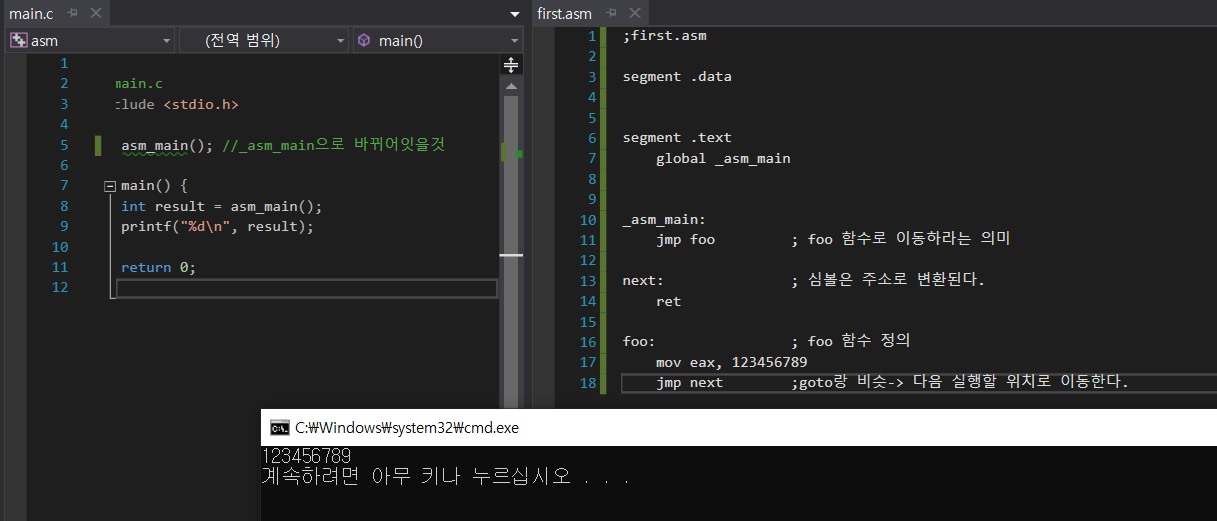




명령줄에 따라 출력이 first.obj가 나올것이다~ 라고 알려주는 역할

* c언어에서 어셈블리어 코드 호출 예제(nasm이 제공해줌..)

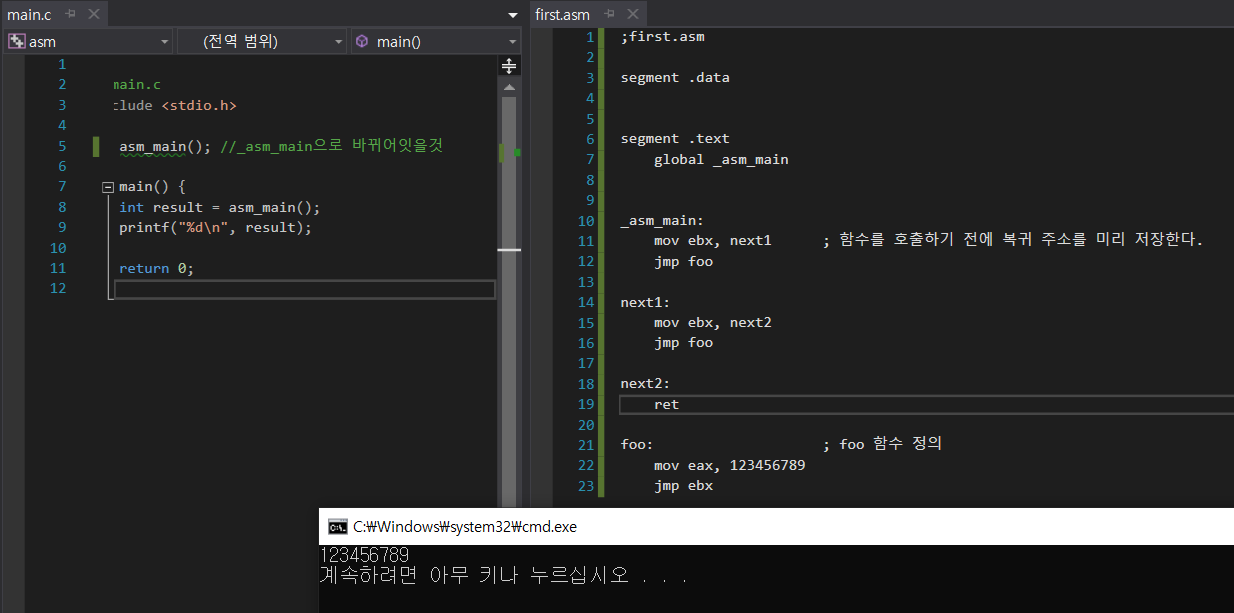


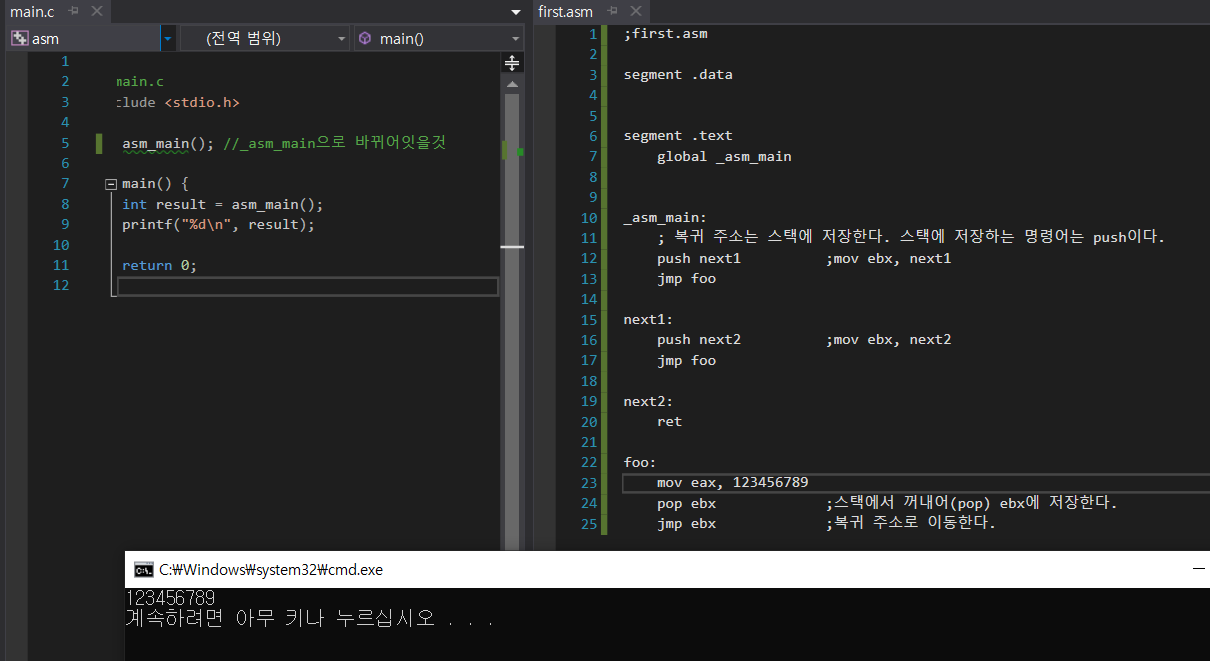


함수가 호출되고나서 복귀할때의 주소가 필요하다

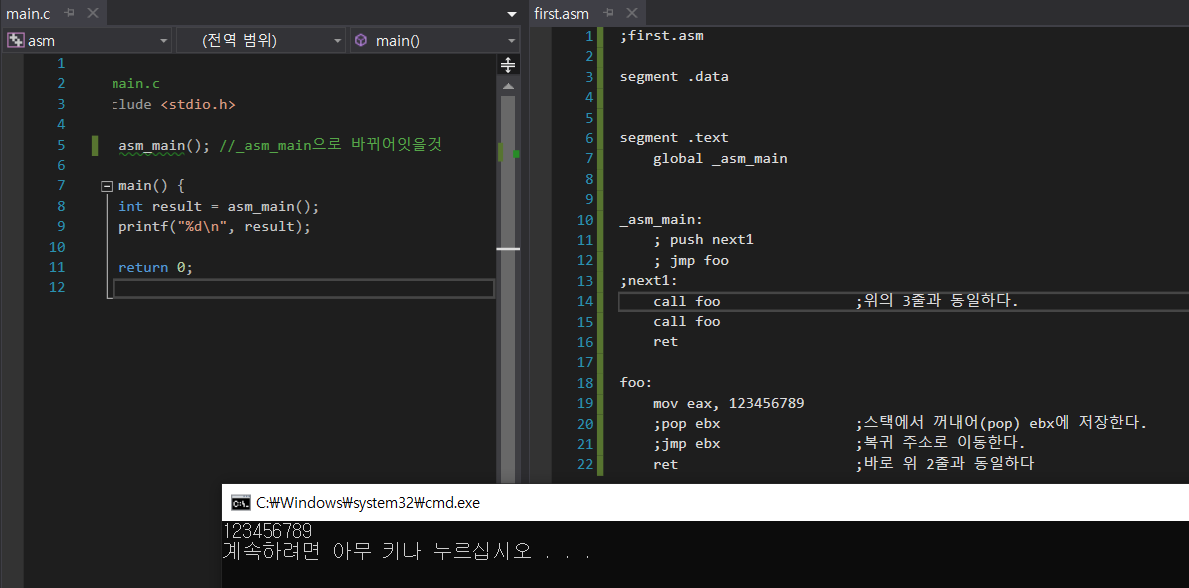
jmp foo를 2번 쓴다고해도 아래에 쓴 jmp foo가 호출되지 않는다. 즉, 복귀주소가 고정되어있어 여러 번 호출해도 한번만 호출된다는 문제가 있다.

그 문제에 대한 첫번째방법 : 1. 레지스터를 사용 – 함수를 호출하기 전에 복귀주소를 미리 저장!

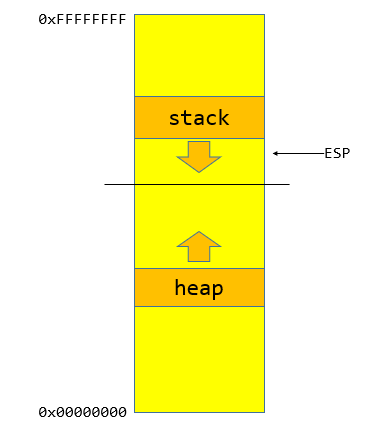




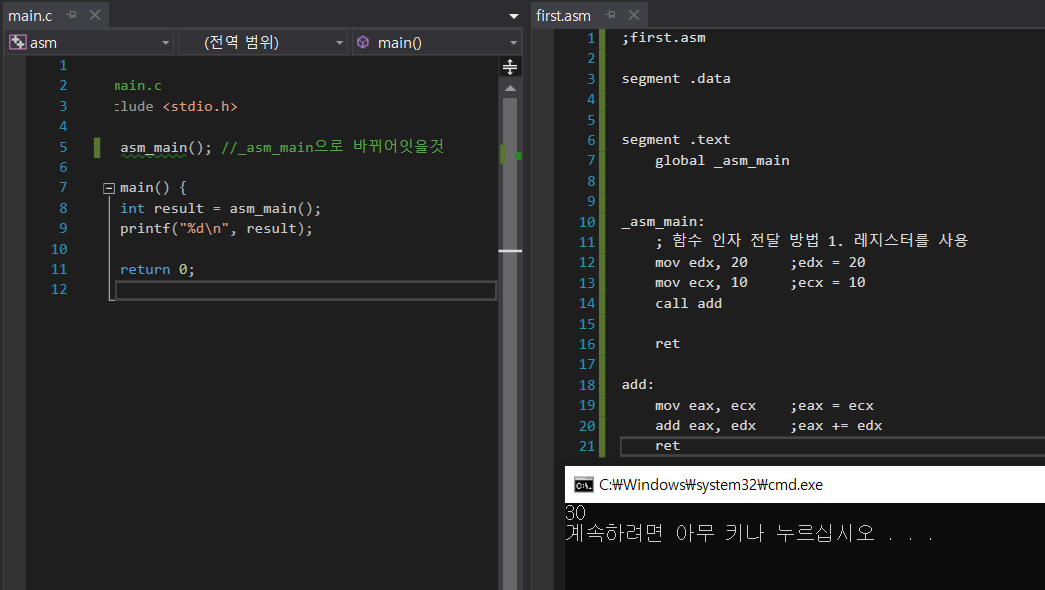
2번째 방법



call함수사용방법(이거 쓰는게 제일 간단. 이거쓰자)



스택이 높은주소에서 낮은주소로! 만약 한계선 지나게되면 interrupt 발생한다.



위 코드는 잘 돌아가지만 레지스터를 사용한다는 단점이있다. **레지스터의 개수는 한정**되어있지만 덧셈따위를 하는데 이렇게 레지스터를 사용하게되면 성능상으로 별로다. (물론 레지스터를 사용한 인자 전달 방식은 빠르긴하다)

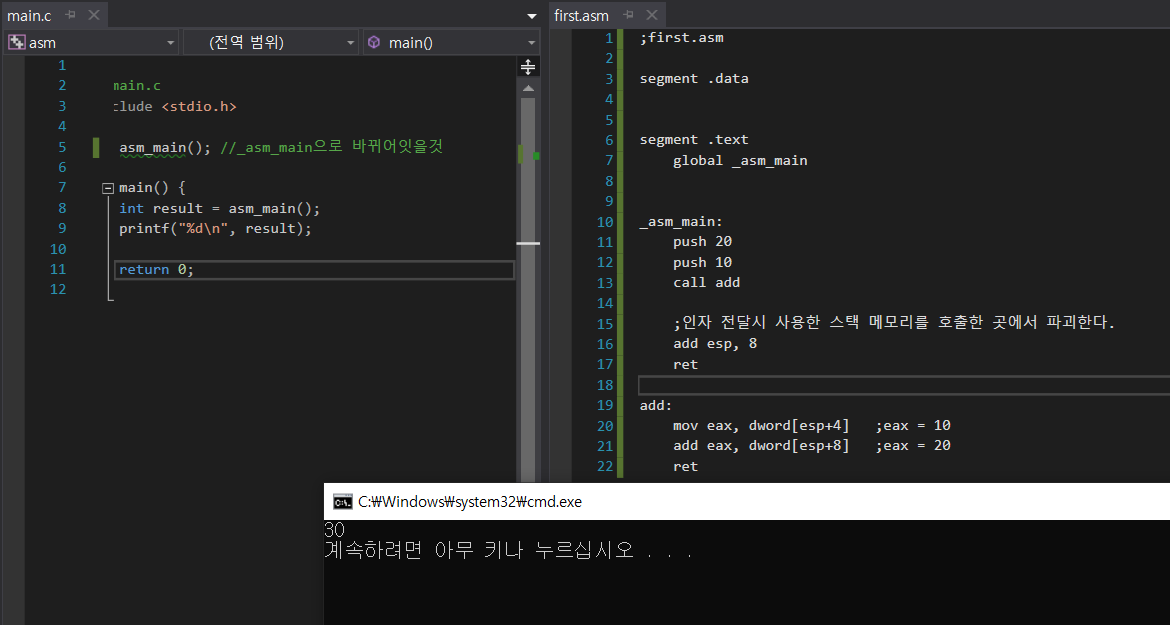
레지스터에 이렇게 넣는걸 권장하지 않는다.



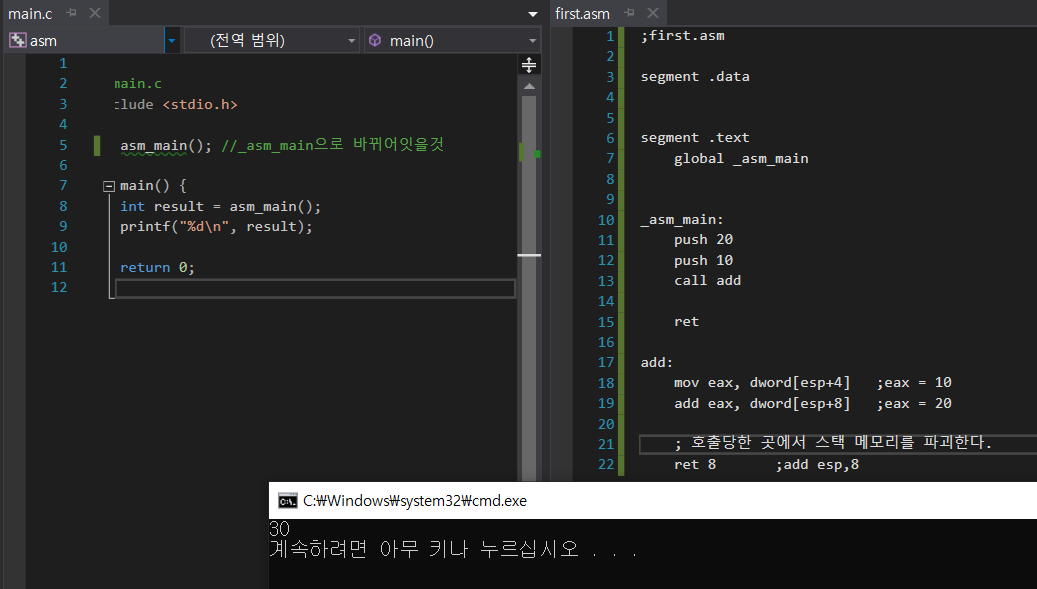
->그렇기에 **스택**을 사용하자! – ESP(Extended Stack Pointer) 은 스택의 top개념으로 스택에 저장할 다음위치를 가리키고있는 포인터이다. push 명령어를 사용할때마다 ESP가 아래로 증가한다.

push한 것을 지워줘야하는데

아래의 두 예제가 각각 2가지 방식이다.



1. 인자 전달시 사용한 스택 메모리를 호출한 곳에서 파괴한다.



1. 호출당한 곳에서 스택 메모리를 파괴한다.

🡪정리!

<함수 호출 규약>

함수 호출시, 인자 전달에 사용한 스택을 해제하는 방식

1. \_\_cdecl : 호출자가 해제하는 방식 -> C 표준

-장점 : 가변인자를 구현할수 있다는 장점이 있다. 함수를 호출한 쪽에서 인자를 전달한 개수를 알수있기 때문이다.

int printf(const char \*fmt, …); -> 이 … 을 쓸수있다.

-단점 : 함수 호출이 끝날때마다 스택을 해제하는 코드가 추가되므로 프로그램의 크기가 커질수 있다.

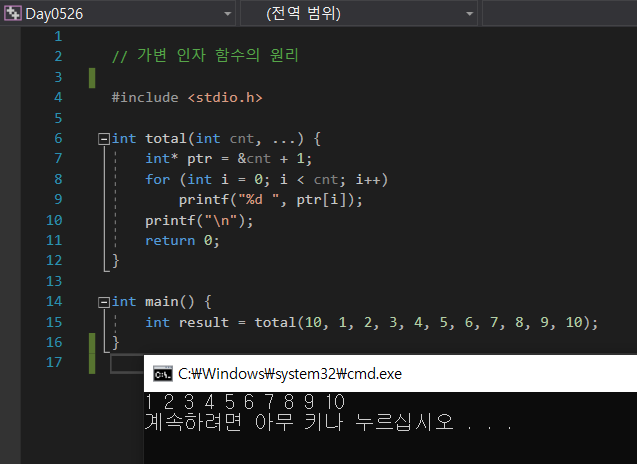
1. \_\_stdcall : 피호출자가 해제하는 방식 -> Win32 API 등

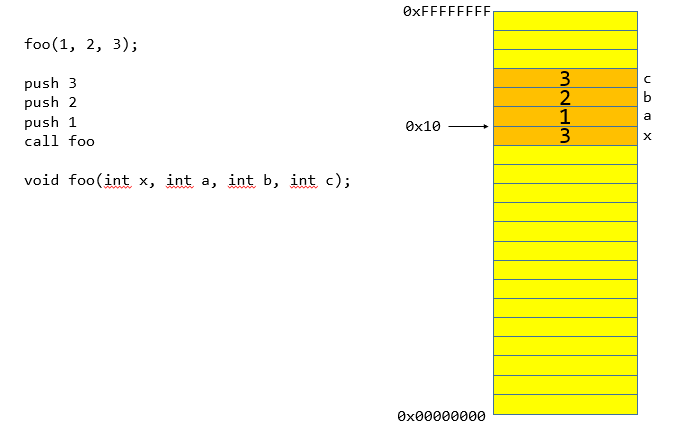
-장점 : 스택을 해제하는 코드가 함수 안에 있으므로 프로그램의 크기가 상대적으로 작아진다는 장점이 있다.

-단점 : 가변인자 함수를 구현할수 없다.

<함수 호출 규약>







이미 가변인자함수를 제공하므로 굳이 만들어쓸 필요는 없으나 원리만 알아두자!

스택이 오른쪽 인자부터 들어가는데 스택이 배열과 같은 느낌으로 쌓인다.

과제 va\_list / va\_start / va\_args / va\_end 매크로에 대해 기술문자 작성해보기 – 가변인자함수