C Programming – 4

임베디드스쿨2기

Lv2과정

2021. 04. 09

박태인

1. 모든 것은 포인터다

1) 포인터 통일

해당 내용을 통해서 결국 C언어의 모든 것이
포인터라는 것을 확인 할 수 있을 것이다.
이중 포인터, 삼중 포인터, 배열, 다중 배열,
포인터 배열, 함수 포인터를 별개로 볼 필요가 없다.
다만 이것을 진행하기 위해서는 몇개지 개념이 필요 하다.

1 . 메모리 계층 구조
2. 스택(Stack)은 아래로 자란다.
3. GP Register에 대한 명확한 개념과 각각의 용도

2) 디버깅 명령어

Info registers : 실제 HW 레지스터 정보를 확인 할 수 있고, **여기서는 펌웨어 제어와 관련된 레지스터 정보는 보여 주지 않는다.**
우리가 이 내용을 진행하면서 주의를 둘 부분은 아래와 같다.

rsp, rbp, rip, rax, rcx 정도에 해당한다.

- rsp : 현재 스택의 최상위
- rbp : 현재 스택의 기준점
- rip : 다음에 실행 할 instruction의 주소 값을 가르킴
- rax : 무조건적으로 함수의 리턴값이 저장되며 연산용으로도 활용 가능
- rcx : 보편적으로 for 루프의 카운트에 활용이 되며 연산용으로도 활용 가능

- si : 어셈블리 명령어 기준으로 한 줄씩 실행한다.
- p/x : 16진수로 특정 결과를 출력한다.
- x : 메모리의 내용을 살펴본다.

1. 디버깅 과정(test_func.c) - (1)

```c
#include <stdio.h>

int my_func(int num)
{
        return num >> 1;
}

int main(void)
{
        int num = 3, res;

        res = my_func(num);

        printf("res = %d\n", res);

        return 0;
}
```

```
res = 1
```

```
04$ gcc -g -o test_func test_func.c
04$ gdb test_func
```

```
[Inferior 1 (proces
(gdb) b main
Breakpoint 1 at 0x5
(gdb) r
Starting program: /

Breakpoint 1, main
9       {
(gdb) disas
```

0000 0010 = 3
>> 0000 0001 = 1

오른쪽 쉬프트 연산하여 위와 같이 **num 3**의 값이
Res = 1 의 값으로
출력 된다.
지금부터 이 소스 코드를 이용해 gdb 를 분석한다!

```
Dump of assembler code for function main:
=> 0x000055555555515b <+0>:     endbr64
   0x000055555555515f <+4>:     push   %rbp
   0x0000555555555160 <+5>:     mov    %rsp,%rbp
   0x0000555555555163 <+8>:     sub    $0x10,%rsp
   0x0000555555555167 <+12>:    movl   $0x3,-0x8(%rbp)
   0x000055555555516e <+19>:    mov    -0x8(%rbp),%eax
   0x0000555555555171 <+22>:    mov    %eax,%edi
   0x0000555555555173 <+24>:    callq  0x555555555149 <my_func>
   0x0000555555555178 <+29>:    mov    %eax,-0x4(%rbp)
   0x000055555555517b <+32>:    mov    -0x4(%rbp),%eax
   0x000055555555517e <+35>:    mov    %eax,%esi
   0x0000555555555180 <+37>:    lea    0xe7d(%rip),%rdi        # 0x555555556004
   0x0000555555555187 <+44>:    mov    $0x0,%eax
   0x000055555555518c <+49>:    callq  0x555555555050 <printf@plt>
   0x0000555555555191 <+54>:    mov    $0x0,%eax
   0x0000555555555196 <+59>:    leaveq
   0x0000555555555197 <+60>:    retq
```

1. 디버깅 과정(test_func.c) - (2)

```
Dump of assembler code for function main:
=> 0x000055555555515b <+0>:     endbr64
   0x000055555555515f <+4>:     push   %rbp
   0x0000555555555160 <+5>:     mov    %rsp,%rbp
   0x0000555555555163 <+8>:     sub    $0x10,%rsp
   0x0000555555555167 <+12>:    movl   $0x3,-0x8(%rbp)
   0x000055555555516e <+19>:    mov    -0x8(%rbp),%eax
   0x0000555555555171 <+22>:    mov    %eax,%edi
   0x0000555555555173 <+24>:    callq  0x555555555149 <my_func>
   0x0000555555555178 <+29>:    mov    %eax,-0x4(%rbp)
   0x000055555555517b <+32>:    mov    -0x4(%rbp),%eax
   0x000055555555517e <+35>:    mov    %eax,%esi
   0x0000555555555180 <+37>:    lea    0xe7d(%rip),%rdi        # 0x555555556004
   0x0000555555555187 <+44>:    mov    $0x0,%eax
   0x000055555555518c <+49>:    callq  0x555555555050 <printf@plt>
   0x0000555555555191 <+54>:    mov    $0x0,%eax
   0x0000555555555196 <+59>:    leaveq
   0x0000555555555197 <+60>:    retq
```

=> 표시는 아직 실행되지 않았고
다음에 실행 하게 될 것이라는 표시!

자... 이제 진짜.. 디버깅을 시작해 봅시다!

먼저 push rbp로 이동을 해본다. (si 실행)

```
Dump of assembler code for function main:
   0x000055555555515b <+0>:     endbr64
=> 0x000055555555515f <+4>:     push   %rbp
```

이후 rsp 값 기록한다(si 실행) : 0x7fffffffdf98 (다음에 새로 시작하면 값이 바뀔수도 있으니 주의한다.)

```
End of assembler dump.
(gdb) x $rsp
0x7fffffffdf98: 0xf7deb0b3
(gdb) x $rbp
0x0:    Cannot access memory at address 0x0
(gdb) p/x $rsp
$1 = 0x7fffffffdf98
```

실제로 처음에 이 값은 시작할때 마다 8 바이트씩 바뀌더라..

부가 설명)
현재 최상위 스택인 rsp가
0x7fffffffdf98 이라는 가상의 주소라는 것이고,
현재 스택의 기준점 값인 rbp는 아직
주소 값이 정해지지 않았고,
값은 '0x0' 이라는 것이다!

1. 디버깅 과정(test_func.c) - (3)

```
Dump of assembler code for function main:
   0x000055555555515b <+0>:     endbr64
   0x000055555555515f <+4>:     push   %rbp
=> 0x0000555555555160 <+5>:     mov    %rsp,%rbp
   0x0000555555555163 <+8>:     sub    $0x10,%rsp
```

자.. 이제 si를 한번 더 실행해서 push %rbp명령어를 실행 했습니다.
Push 명령어는 **현재 스택의 최상위 메모리(rsp)**에 값을 저장하는 명령어 입니다.
즉, **현재 스택의 최상위 메모리(rsp)**에 **rbp값을 저장하라**는 의미 겠죠.
그런데, 앞 ppt에서 보듯이 rbp의 값은 0x0 이었으므로 아래와 같이 구성 되겠습니다.

0x0 (rbp)	0x0x7fffffffdf90 (rsp)

```
   0x000055555555515b <+0>:     endbr64
   0x000055555555515f <+4>:     push   %rbp
   0x0000555555555160 <+5>:     mov    %rsp,%rbp
=> 0x0000555555555163 <+8>:     sub    $0x10,%rsp
```

```
(gdb) p/x $rsp
$2 = 0x7fffffffdf90
(gdb) x $rsp
0x7fffffffdf90: 0x00000000
(gdb) x $rbp
0x7fffffffdf90: 0x00000000
(gdb) p/x $rbp
$3 = 0x7fffffffdf90
```

ㄴ 사라진 경계선

자 이제 한번 더 si를 실행해서 mov %rsp, %rbp 를 시행 해봅시다.
Mov 명령어는 내용을 복사하는 것 입니다.
즉, mov rsp rbp는 rbp에 rsp 값을 복사합니다. (rsp 값 → rbp 값)
ㄴ일반적인 A = B 꼴에서 B 값이 A로 들어가는 것이 아닌 반대 방향으로 생각해야 하는 것에 주의 하자!

그러면 어떻게 되겠나요?
ㄴ 결국, rbp에 rsp 값을 넣어 버림 으로써 **rsp, rbp 주소 값이 서로 같아지면서 스택의 경계선이 사라집니다!**
→ 이것은 **새로운 스택을 생성 할 준비를 하는 과정** 입니다.

1. 디버깅 과정(test_func.c) - (4)

```
   0x000055555555515b <+0>:     endbr64
   0x000055555555515f <+4>:     push   %rbp
   0x0000555555555160 <+5>:     mov    %rsp,%rbp
   0x0000555555555163 <+8>:     sub    $0x10,%rsp
=> 0x0000555555555167 <+12>:    movl   $0x3,-0x8(%rbp)
   0x000055555555516e <+19>:    mov    -0x8(%rbp),%eax
```

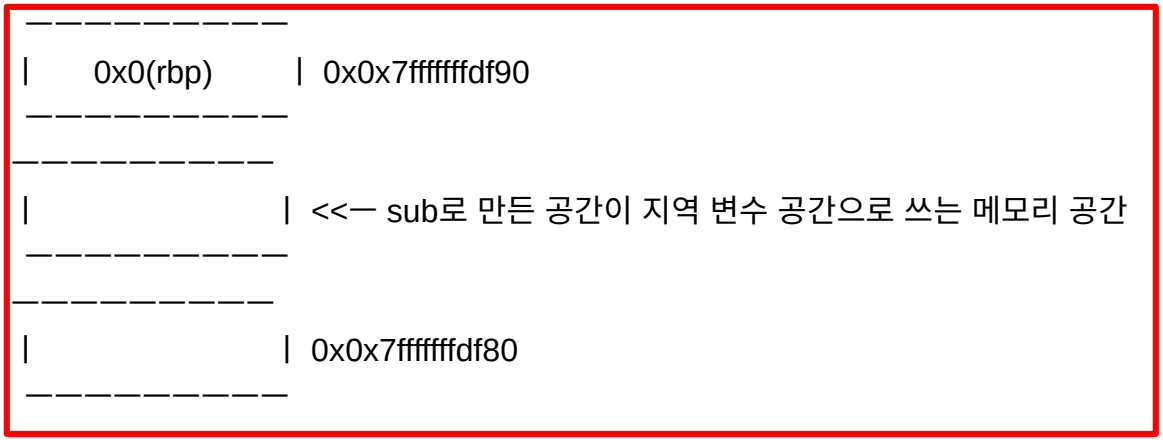
자 이번에는 sub 명령어를 실행 했습니다.

Sub 명령어는 빨셈 명령 입니다.

Sub 0x10, rsp는 현재 rsp 에서 **16바이트를 빼겠다는 의미** 입니다. (0x10 → 0001 0000 : 2^4 = 16 바이트,

가상 주소 공간에서는 바이트 단위로 움직입니다!)

그림으로 나타내어 보면 아래와 같습니다.



ㄴ 이런 구조 때문에 스택은 아래로 자란다고 한 것이다!!

```
(gdb) x $rbp
0x7fffffffdf90: 0x00000000
(gdb) x $rsp
0x7fffffffdf80: 0xffffe080
```

: 현재 스택의 최상위 rsp 80
현재 스택의 기준점 rbp 90

1. 디버깅 과정(test_func.c) - (5)

```
   0x0000555555555160 <+5>:     mov    %rsp,%rbp
   0x0000555555555163 <+8>:     sub    $0x10,%rsp
   0x0000555555555167 <+12>:    movl   $0x3,-0x8(%rbp)
=> 0x000055555555516e <+19>:    mov    -0x8(%rbp),%eax
   0x0000555555555171 <+22>:    mov    %eax,%edi
```

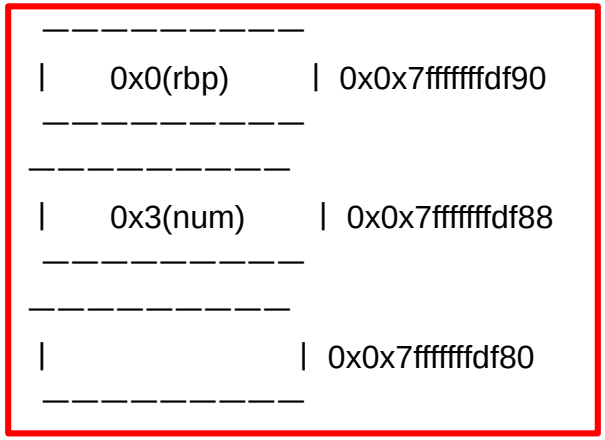
자, 이번에는 movl $0x3, -0x8(%rbp)이 보인다.

L 이 들어가면 4바이트 처리를 하겠다는 의미 이며

Q 가 들어가면 8바이트 처리를 하겠다는 뜻이 된다.

여튼, 위 명령어의 의미는 **rbp를 기준으로 8바이트 뺀 자리에 0x3을 복사한다는 의미** 이다.

그림으로 나타내 보면 아래와 같다.



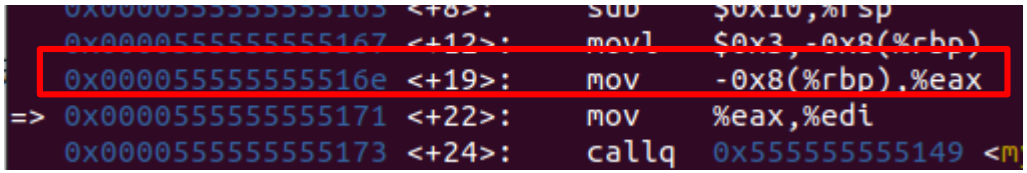

```
#include <stdio.h>

int my_func(int num)
{
        return num >> 1;
}

int main(void)
{
        int num = 3, res;

        res = my_func(num);

        printf("res = %d\n", res);

        return 0;
}
```

ㄴ 원소스의 num에 3을 넣는 과정

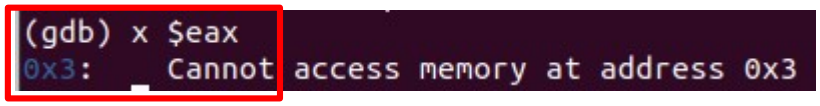
```
(gdb) x $rsp
0x7fffffffdf80: 0xffffe080
(gdb) x $rbp
0x7fffffffdf90: 0x00000000
(gdb) x $rbp-8
0x7fffffffdf88: 0x00000003
(gdb) x $rsp+8
0x7fffffffdf88: 0x00000003
```

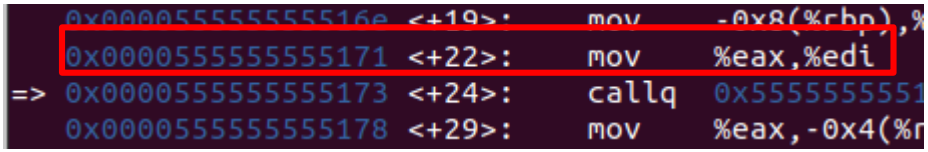
ㄴ 그림 처럼 rsp+8 위치 혹은 rbp-8 위치에 3의 값이 들어 간 것 을 확인 할 수 있다!

1. 디버깅 과정(test_func.c) - (6)

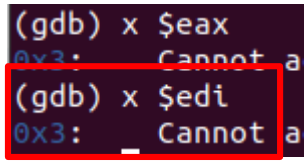


자, 이번에도 mov 명령어이다. 앞서 설명한 바와 같이 mov명령어는 내용을 복사하는 것이다.
그렇다면 mov -0x8(%rbp), %eax 는
eax레지스터(4바이트 레지스터)에 rbp기준 -8 바이트 값, 즉 0x3(num)을 eax에 넣겠다는 것이다.
이 것은 연산에 사용되며, 연산 이후 ax 레지스터를 확인하면 변경된 값을 볼 수 있다.





그리고 다음의 mov eax, edi는 그냥 복사이다.
따라서, 아래와 같이 edi에 3이 들어 가는 것이다.



1. 디버깅 과정(test_func.c) - (7)

```
   0x0000555555555163 <+8>:     sub    $0x10,%rsp
   0x0000555555555167 <+12>:    movl   $0x3,-0x8(%rbp)
   0x000055555555516e <+19>:    mov    -0x8(%rbp),%eax
=> 0x0000555555555171 <+22>:    mov    %eax,%edi
   0x0000555555555173 <+24>:    callq  0x555555555149 <my_func>
   0x0000555555555178 <+29>:    mov    %eax,-0x4(%rbp)
   0x000055555555517b <+32>:    mov    -0x4(%rbp),%eax
```

Callq 0x555555555149 은 매우 중요한 연산이다.

Call 은 기본적으로 push + jmp로 구성되어 있다.

함수 호출이 끝난 이후에 실행해야 할 어셈블리 명령어의 주소값을 psuh로 저장한다.

이후 함수 호출을 수행하기 위해 jmp를 수행한다.

결국 아래와 같은 메모리를 가지게 된다.

0x0(rbp)	0x0x7fffffffdf90
0x3(num)	0x0x7fffffffdf88
	0x0x7fffffffdf80
0x0000555555555178(복귀주소)	0x0x7fffffffdf78

```
(gdb) x $rsp
0x7fffffffdf78: 0x55555178
```

```
(gdb) x $rsp+4
0x7fffffffdf7c: 0x00005555
```

ㄴ 스택의 최상위 값인 rsp가 78이 되고 안에
복귀 주소인 0x0000555555555178 이 저장된 것을 볼 수 있다.
(복귀 주소가 너무 길어서 rsp+4 명령어 사용해서 뷰!)

1. 디버깅 과정(test_func.c) - (8)

```
Dump of assembler code for function my_func:
   0x0000555555555149 <+0>:     endbr64
=> 0x000055555555514d <+4>:     push   %rbp
   0x000055555555514e <+5>:     mov    %rsp,%rbp
   0x0000555555555151 <+8>:     mov    %edi,-0x4(%rbp)
   0x0000555555555154 <+11>:    mov    -0x4(%rbp),%eax
   0x0000555555555157 <+14>:    sar    %eax
   0x0000555555555159 <+16>:    pop    %rbp
   0x000055555555515a <+17>:    retq
```

위는 함수로 이동하게 된 모습이다.
여기서 **push rbp를 진행하게 되면 rsp 값이 추가로 8바이트 빠지므로** 메모리는 아래와 같은 구성을 하게 될 것이다.
또한, **기존의 스택의 기준점인 rbp를 저장**하게 될 것이다.

0x0(rbp)	0x0x7fffffffdf90
0x3(num)	0x0x7fffffffdf88
	0x0x7fffffffdf80
0x0000555555555178(복귀주소)	0x0x7fffffffdf78
0x0x7fffffffdf90(이전 함수의 rbp)	0x0x7fffffffdf70

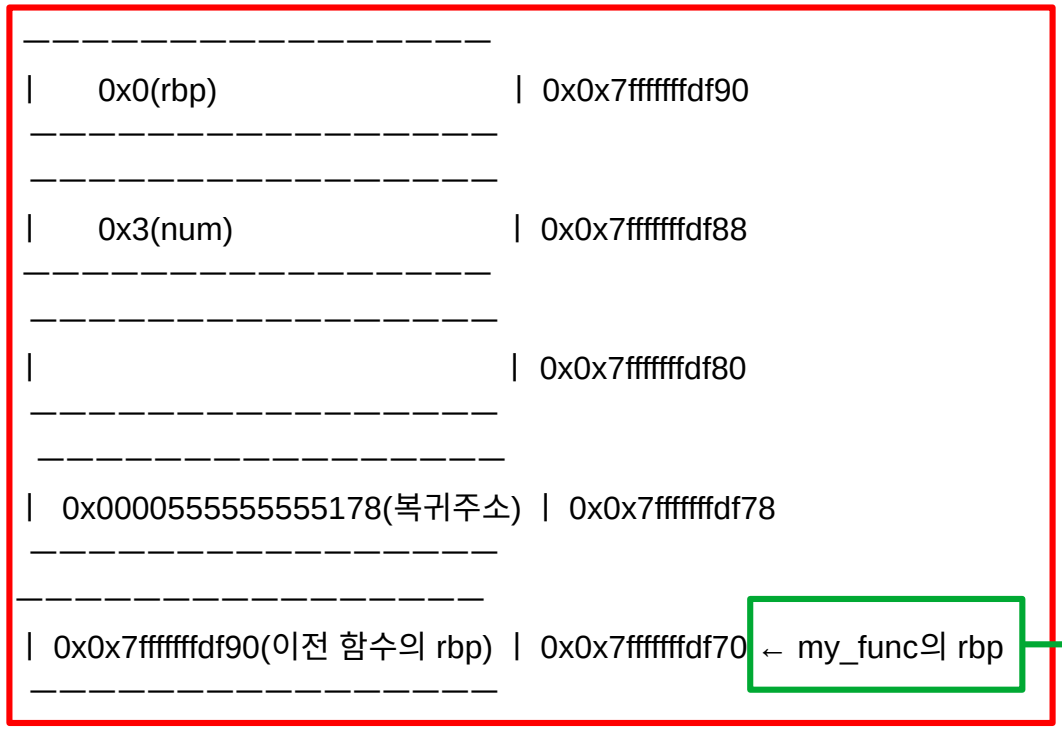
```
(gdb) x $rsp
0x7fffffffdf70: 0xffffdf90
```

ㄴ 스택의 최상위 값인 rsp가 70이 되고 안에
이전 함수의 rbp가 저장된다(아직 rbp는 바뀐적이 없으므로)

1. 디버깅 과정(test_func.c) - (9)

```
   0x000055555555514d <+4>:     push   %rbp
   0x000055555555514e <+5>:     mov    %rsp,%rbp
=> 0x0000555555555151 <+8>:     mov    %edi,-0x4(%rbp)
   0x0000555555555154 <+11>:    mov    -0x4(%rbp),%eax
   0x0000555555555157 <+14>:    sar    %eax
```

이후 다시 **mov rsp, rbp를 하면 새로운 rbp가 생성**된다. (rsp → rbp)
이것이 my_func에 해당하는 새로운 스택의 기준점이 된다.
결국 함수를 호출 할 때 마다 스택을 새롭게 생성한다는 것이 되며,
우리가 **포인터를 사용하는 이유**는
바로 이 경계선을 넘어 자유롭게 왔다 갔다 할 수 있기 때문이다.




```
(gdb) x $rsp
0x7fffffffdf70: 0xffffdf90
(gdb) x $rbp
0x7fffffffdf70: 0xffffdf90
```

1. 디버깅 과정(test_func.c) - (10)

```
Dump of assembler code for function my_func:
   0x0000555555555149 <+0>:     endbr64
   0x000055555555514d <+4>:     push   %rbp
   0x000055555555514e <+5>:     mov    %rsp,%rbp
   0x0000555555555151 <+8>:     mov    %edi,-0x4(%rbp)
=> 0x0000555555555154 <+11>:    mov    -0x4(%rbp),%eax
   0x0000555555555157 <+14>:    sar    %eax
   0x0000555555555159 <+16>:    pop    %rbp
   0x000055555555515a <+17>:    retq
```

이후 **edi 값을 rbp-4 위치에 배치** 한다.

edi에는 eax에서 옮겨온 인자값 3이 배치가 된다.

```
#include <stdio.h>

int my_func(int num)
{
        return num >> 1;
}

int main(void)
{
        int num = 3, res;

        res = my_func(num);

        printf("res = %d\n", res);

        return 0;
}
```

```
(gdb) p/x $edi
$2 = 0x3
(gdb) x $rbp-4
0x7fffffffdf6c: 0x00000003
```

ㄴ rbp 기준으로 -4 위치에 0x3 의 값 적용

0x0(rbp)	0x0x7fffffffdf90
0x3(num)	0x0x7fffffffdf88
	0x0x7fffffffdf80
0x0000555555555178(복귀주소)	0x0x7fffffffdf78
0x0x7fffffffdf90(이전 함수의 rbp)	0x0x7fffffffdf70 ← my_func의 rbp
0x3 (num아니고 my_func의 num)	0x0x7fffffffdf6c

1. 디버깅 과정(test_func.c) - (11)

```
Dump of assembler code for function my_func:
   0x0000555555555149 <+0>:     endbr64
   0x000055555555514d <+4>:     push   %rbp
   0x000055555555514e <+5>:     mov    %rsp,%rbp
   0x0000555555555151 <+8>:     mov    %edi,-0x4(%rbp)
=> 0x0000555555555154 <+11>:    mov    -0x4(%rbp),%eax
   0x0000555555555157 <+14>:    sar    %eax
   0x0000555555555159 <+16>:    pop    %rbp
   0x000055555555515a <+17>:    retq
```

Rbp-4 위치에 있는 값을 eax에 배치 한다.
(여기에 rbp-4에는 **my_func의 num** 값이 들어간다.)

0x0(rbp)	0x0x7fffffffdf90
0x3(num)	0x0x7fffffffdf88
	0x0x7fffffffdf80
0x0000555555555178(복귀주소)	0x0x7fffffffdf78
0x0x7fffffffdf90(이전 함수의 rbp)	0x0x7fffffffdf70 ← my_func의 rbp
0x3 (num아니고 my_func의 num)	0x0x7fffffffdf6c

```
(gdb) x $eax
0x3:    Cannot
```

포기하면 얻는 건 아무것도 없다.

1. 디버깅 과정(test_func.c) - (12)

```
   0x0000555555555149 <+0>:     endbr64
   0x000055555555514d <+4>:     push   %rbp
   0x000055555555514e <+5>:     mov    %rsp,%rbp
   0x0000555555555151 <+8>:     mov    %edi,-0x4(%rbp)
   0x0000555555555154 <+11>:    mov    -0x4(%rbp),%eax
   0x0000555555555157 <+14>:    sar    %eax
=> 0x0000555555555159 <+16>:    pop    %rbp
   0x000055555555515a <+17>:    retq
```

자, 이번에는 sar 명령어 이다. **Sar 은 Shift Arithmetic Right의 약자**로 오른쪽 쉬프트이다.
Sar %eax의 경우 **뒤쪽에 비트가 표현이 안되어 있으므로 기본적으로 1이다 (>> 1을 의미)**

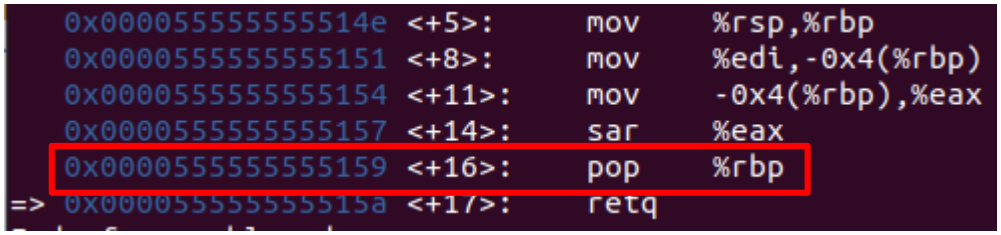
eax는 3이므로 결과는 1이 될 것이다.

```
(gdb) x $eax
0x1:    Canno
```

ax는 함수의 리턴값을 가진다고 했다.
실제로 이 값은 my_func이 리턴하는 값에 해당한다.

0x0(rbp)	0x0x7fffffffdf90
0x3(num)	0x0x7fffffffdf88
	0x0x7fffffffdf80
0x0000555555555178(복귀주소)	0x0x7fffffffdf78
0x0x7fffffffdf90(이전 함수의 rbp)	0x0x7fffffffdf70 ← my_func의 rbp
0x3 (num아니고 my_func의 num)	0x0x7fffffffdf6c

1. 디버깅 과정(test_func.c) - (13)



다음으로 pop %rbp를 수행하는데 **pop은 현재 rsp 에서 값을 빼서**
뒤쪽에 배치된 메모리나 레지스터에 값을 넘겨 준다.(여기서는 현재 rsp 값을 rbp에 주는 것 이겠죠)

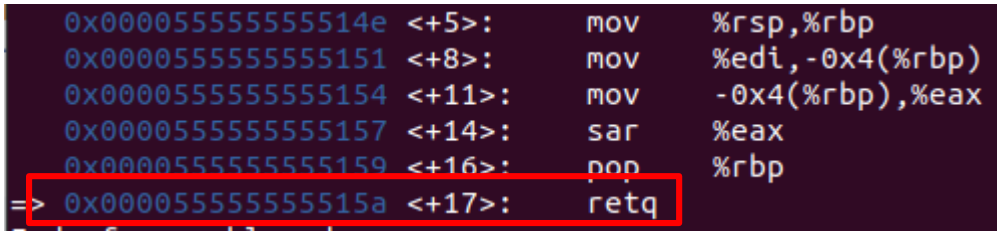
결국 값을 빼내기 때문에 rsp값은 70에서 78로 증가하게 되고
내부에 있는 값은 rbp로 들어가므로 이전의 rbp가 복원된다.
(**결국 스택을 복원하는 작업의 일부에 해당함**)

이 시점의 메모리는 다음과 같다.

```
(gdb) x $rsp
0x7fffffffdf78:
(gdb) x $rbp
0x7fffffffdf90:
```

0x0(rbp)	0x0x7fffffffdf90 ← rbp (rsp안의 값이 이전 함수의 rbp)
0x3(num)	0x0x7fffffffdf88
	0x0x7fffffffdf80
0x0000555555555178(복귀주소)	0x0x7fffffffdf78 ← rsp (밑에 있던 rsp가 올라옴)
0x0x7fffffffdf90(이전 함수의 rbp)	0x0x7fffffffdf70
0x3 (num아니고 my_func의 num)	0x0x7fffffffdf6c

1. 디버깅 과정(test_func.c) - (14)



다음으로 retq를 진행하는데 retq는 아래와 같은 의미를 가진다.
Pop rip 에 해당하는 연산이다.
즉, rsp에서 값을 빼서 rip에 배치하는 것이다.

결국 값을 빼내기 때문에 rsp값은 78에서 8로 증가하게 되고
내부에 있는 값은 rip으로 들어간다..

이 시점의 메모리는 다음과 같다.

```
(gdb) x $rsp
0x7fffffffdf80:
(gdb) x $rip
0x555555555178
```

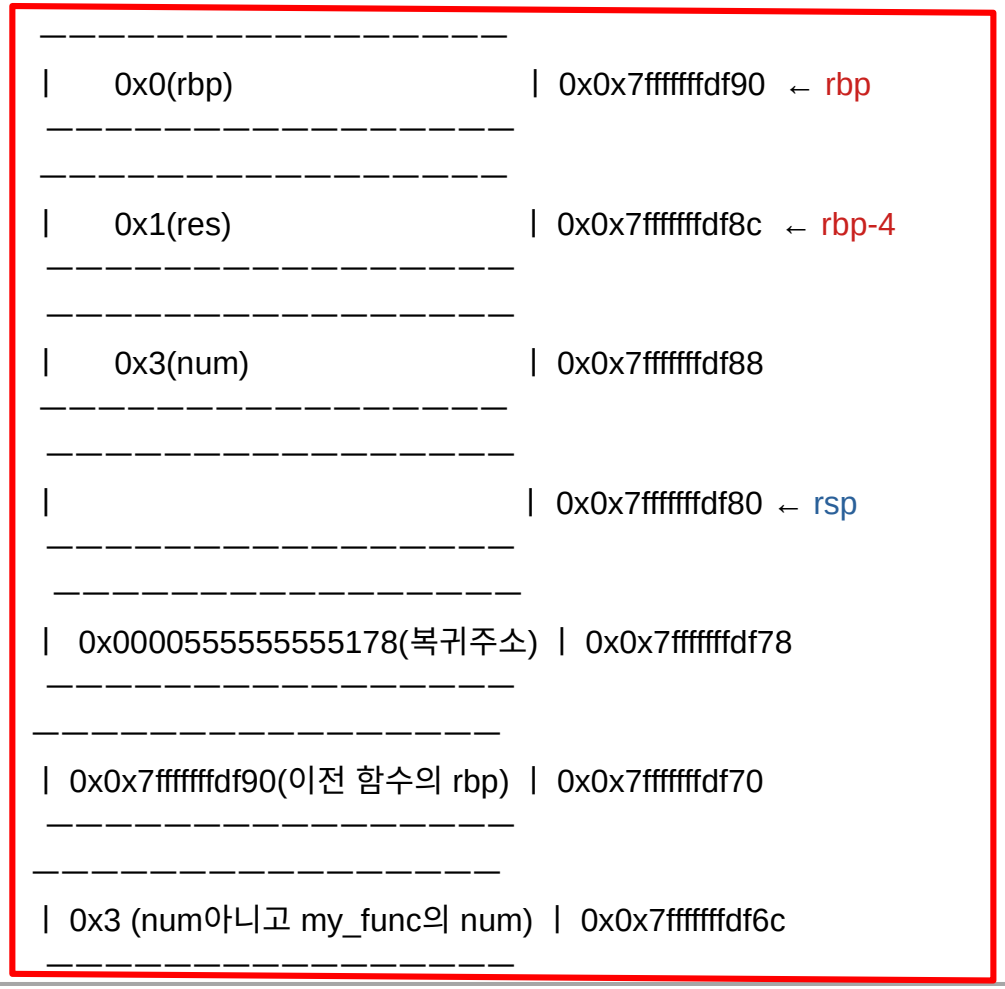
0x0(rbp)	0x0x7fffffffdf90 ← rbp
0x3(num)	0x0x7fffffffdf88
	0x0x7fffffffdf80 ← rsp (밑에 있던 rsp가 올라옴)
0x0000555555555178(복귀주소)	0x0x7fffffffdf78
0x0x7fffffffdf90(이전 함수의 rbp)	0x0x7fffffffdf70
0x3 (num아니고 my_func의 num)	0x0x7fffffffdf6c

1. 디버깅 과정(test_func.c) - (15)

```
   0x0000555555555167 <+12>:    movl   $0x3,-0x8(%rbp)
   0x000055555555516e <+19>:    mov    -0x8(%rbp),%eax
   0x0000555555555171 <+22>:    mov    %eax,%edi
   0x0000555555555173 <+24>:    callq  0x555555555149 <my_func>
=> 0x0000555555555178 <+29>:    mov    %eax,-0x4(%rbp)
   0x000055555555517b <+32>:    mov    -0x4(%rbp),%eax
   0x000055555555517e <+35>:    mov    %eax,%esi
   0x0000555555555180 <+37>:    lea    0xe7d(%rip),%rdi        #
```

복귀 후에 rbp-4 자리에 리턴값인 eax 레지스터를 배치한다.
메모리 구조는 오른쪽과 같다.

```
(gdb) x $rbp-4
0x7fffffffdf8c: 0x00000001
```

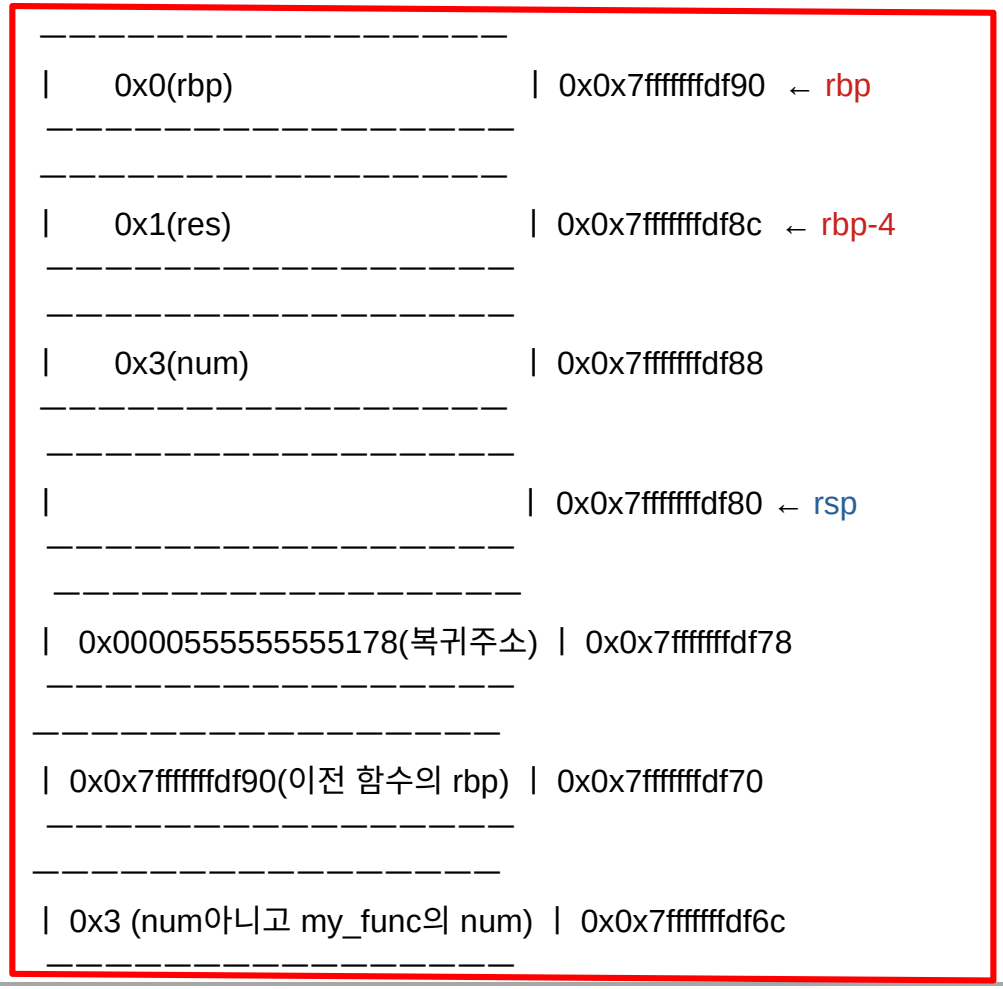


1. 디버깅 과정(test_func.c) - (16)

```
   0x0000555555555178 <+29>:    mov    %eax,-0x4(%rbp)
   0x000055555555517b <+32>:    mov    -0x4(%rbp),%eax
=> 0x000055555555517e <+35>:    mov    %eax,%esi
   0x0000555555555180 <+37>:    lea    0xe7d(%rip),%rdi
   0x0000555555555187 <+44>:    mov    $0x0,%eax
   0x000055555555518c <+49>:    callq  0x555555555050 <pri
   0x0000555555555191 <+54>:    mov    $0x0,%eax
```

다음으로 rbp-4에 값을 eax에 배치 한다.
즉, 0x1(res)를 eax에 넣는거죠.

```
(gdb) x $rbp-4
0x7fffffffdf8c: 0x00000001
(gdb) x $eax
0x1:    Cannot access memory
```



1. 디버깅 과정(test_func.c) - (17)

```
   0x0000555555555178 <+29>:    mov    %eax,-0x4(%rbp)
   0x000055555555517b <+32>:    mov    -0x4(%rbp),%eax
=> 0x000055555555517e <+35>:    mov    %eax,%esi
   0x0000555555555180 <+37>:    lea    0xe7d(%rip),%rdi
   0x0000555555555187 <+44>:    mov    $0x0,%eax
```

다음으로 eax 값은 esi로 복사.
즉, 0x1값이 eax, esi에 각각 배치 된다.

```
(gdb) x $eax
0x1:    Cannot
(gdb) x $esi
0x1:    Cannot
```

```
   0x000055555555517e <+35>:    mov    %eax,%esi
   0x0000555555555180 <+37>:    lea    0xe7d(%rip),%rdi        # 0x555555556004
=> 0x0000555555555187 <+44>:    mov    $0x0,%eax
```

Lea 명령어의 경우 **배열**인데, 0xe7d(%rip)의 값이 %rdi에 배치 된다고 라고만 이해하자.

```
(gdb) x $rdi
0x555555556004: 0x20736572
```

```
   0x0000555555555180 <+37>:    lea    0xe7d(%rip),%r
=> 0x0000555555555187 <+44>:    mov    $0x0,%eax
   0x000055555555518c <+49>:    callq  0x555555555050
   0x0000555555555191 <+54>:    mov    $0x0,%eax
```

0x0 값을 eax에 복사 배치 한다.

```
(gdb) x $eax
0x0:    Cannot
```

1. 디버깅 과정(test_func.c) - (18)

```
   0x0000555555555171 <+22>:    mov    %eax,%edi
   0x0000555555555173 <+24>:    callq  0x555555555149 <my_func>
   0x0000555555555178 <+29>:    mov    %eax,-0x4(%rbp)
   0x000055555555517b <+32>:    mov    -0x4(%rbp),%eax
   0x000055555555517e <+35>:    mov    %eax,%esi
   0x0000555555555180 <+37>:    lea    0xe7d(%rip),%rdi        # 0x
=> 0x0000555555555187 <+44>:    mov    $0x0,%eax
   0x000055555555518c <+49>:    callq  0x555555555050 <printf@plt>
   0x0000555555555191 <+54>:    mov    $0x0,%eax
   0x0000555555555196 <+59>:    leaveq
```

Callq 0x555555555050 의 연산이다.(앞서 만들었던 함수 호출처럼)

Call 은 기본적으로 push + jmp로 구성되어 있다.

함수 호출이 끝난 이후에 실행해야 할 어셈블리 명령어의 주소값을 psuh로 저장한다.

이후 함수 호출을 수행하기 위해 jmp를 수행한다.

결국 아래와 같은 메모리를 가지게 된다.

부가 설명 : push로 주소값 저장하게 되므로 현재 rsp에서 8바이트 더해진 주소에 복귀주소가 저장된다. (앞 있던 복귀 주소가 엎어쓰기 된다!!)

0x0(rbp)	0x0x7fffffffdf90 ← rbp
0x1(res)	0x0x7fffffffdf8c ← rbp-4
0x3(num)	0x0x7fffffffdf88
	0x0x7fffffffdf80 ← rsp
0x0000555555555191(복귀주소)	0x0x7fffffffdf78
0x0x7fffffffdf90(이전 함수의 rbp)	0x0x7fffffffdf70
0x3 (num아니고 my_func의 num)	0x0x7fffffffdf6c

1. 디버깅 과정(test_func.c) - (19)

```
=> 0x0000555555555050 <+0>:     endbr64
   0x0000555555555054 <+4>:     bnd jmpq *0x2f75(%rip)        # 0x555555557fd0 <printf@got.plt>
   0x000055555555505b <+11>:    nopl   0x0(%rax,%rax,1)
```

Pritnf 함수의 어셈블리 내용이다.
0x2f75의 포인터 배열에 값을 rip에 배치를 하고
널 값을 리턴 후 나오는 구조 인듯하다.
(si를 하면 미궁에 빠지므로 ni로 넘길 것)

0x0(rbp)	0x0x7fffffffdf90 ← rbp
0x1(res)	0x0x7fffffffdf8c ← rbp-4
0x3(num)	0x0x7fffffffdf88
	0x0x7fffffffdf80 ← rsp
0x0000555555555191(복귀주소)	0x0x7fffffffdf78
0x0x7fffffffdf90(이전 함수의 rbp)	0x0x7fffffffdf70
0x3 (num아니고 my_func의 num)	0x0x7fffffffdf6c

1. 디버깅 과정(test_func.c) - (20)

```
   0x000055555555518c <+49>:    callq  0x555555555050 <printf@plt>
   0x0000555555555191 <+54>:    mov    $0x0,%eax
=> 0x0000555555555196 <+59>:    leaveq
   0x0000555555555197 <+60>:    retq
```

mov로 0x0값을 eax에 배치 하고,

```
(gdb) x $eax
0x0:    Cannot
```

Leaveq 는 스택해제 명령어 이다.

```
(gdb) x $rsp
0x7fffffffdf98:
```

ㄴrsp가 안드로메다로 갔쥬

```
————————————————————————
|     0x0(rbp)                  | 0x0x7fffffffdf90  ← rbp
————————————————————————
————————————————————————
|     0x1(res)                  | 0x0x7fffffffdf8c  ← rbp-4
————————————————————————
————————————————————————
|     0x3(num)                  | 0x0x7fffffffdf88
————————————————————————
————————————————————————
|                               | 0x0x7fffffffdf80  ← rsp
————————————————————————
————————————————————————
| 0x0000555555555191(복귀주소)  | 0x0x7fffffffdf78
————————————————————————
————————————————————————
| 0x0x7fffffffdf90(이전 함수의 rbp) | 0x0x7fffffffdf70
————————————————————————
————————————————————————
| 0x3 (num아니고 my_func의 num) | 0x0x7fffffffdf6c
————————————————————————
```

2. if문에 대한 기계어 분석 (test_if.c) - (1)

```c
#include <stdio.h>

int main(void)
{
        int num1 = 1, num2 = 2;


        if(num1 > num2)
        {
                printf("num1(%d)가 num2(%d)보다 큽니다.\n", num1, num2);
        }
        else
        {
                printf("num2(%d)가 num1(%d)보다 큽니다.\n", num2, num1);
        }

        return 0;
}
```

If 문 예제 작성

If 문의 어셈블리 동작을 분석하기 위한 코드.

Num1,2의 변수를 비교해서 값을 출력한다.

2. if문에 대한 기계어 분석 (test_if.c) - (2)

```
   0x0000555555555149 <+0>:     endbr64
   0x000055555555514d <+4>:     push   %rbp
=> 0x000055555555514e <+5>:     mov    %rsp,%rbp
   0x0000555555555151 <+8>:     sub    $0x10,%rsp
```

자.. 이제 si를 실행해서 push %rbp명령어를 실행 했습니다.
Push 명령어는 **현재 스택의 최상위 메모리(rsp)**에 값을 저장하는 명령어 입니다.
즉, **현재 스택의 최상위 메모리(rsp)**에 **rbp값을 저장하라**는 의미 겠죠.
그런데, 앞 ppt에서 보듯이 rbp의 값은 0x0 이었으므로 아래와 같이 구성 되겠습니다.

0x0 (rbp)	0x0x7fffffffdfa0 (rsp)

```
(gdb) x $rsp
0x7fffffffdfa0:
(gdb) x $rbp
```

```
   0x000055555555514e <+5>:     mov    %rsp,%rbp
=> 0x0000555555555151 <+8>:     sub    $0x10,%rsp
```

자 이제 한번 더 si를 실행해서 mov %rsp, %rbp 를 시행 해봅시다.
Mov 명령어는 내용을 복사하는 것 입니다.
즉, mov rsp rbp는 rbp에 rsp 값을 복사합니다. (rsp 값 → rbp 값)
ㄴ일반적인 A = B 꼴에서 B 값이 A로 들어가는 것이 아닌 반대 방향으로 생각해야 하는 것에 주의 하자!

```
(gdb) x $rsp
0x7fffffffdfa0:
(gdb) x $rbp
0x7fffffffdfa0:
```

ㄴ 사라진 경계선

그러면 어떻게 되겠나요?
ㄴ 결국, rbp에 rsp 값을 넣어 버림 으로써 **rsp, rbp 주소 값이 서로 같아지면서 스택의 경계선이 사라집니다!**
→ 이것은 **새로운 스택을 생성 할 준비를 하는 과정** 입니다.

2. if문에 대한 기계어 분석 (test_if.c) - (3)

```
   0x000055555555514d <+4>:     push   %rbp
   0x000055555555514e <+5>:     mov    %rsp,%rbp
   0x0000555555555151 <+8>:     sub    $0x10,%rsp
=> 0x0000555555555155 <+12>:    movl   $0x1,-0x8(%rbp)
   0x000055555555515c <+19>:    movl   $0x2,-0x4(%rbp)
```

자 이번에는 sub 명령어를 실행 했습니다.

Sub 명령어는 빨셈 명령 입니다.

Sub 0x10, rsp는 현재 rsp 에서 **16바이트를 빼겠다는 의미** 입니다. (0x10 → 0001 0000 : 2^4 = **16 바이트,**

가상 주소 공간에서는 바이트 단위로 움직입니다!)

그림으로 나타내어 보면 아래와 같습니다.

```
 ──────────
|    0x0(rbp)     | 0x0x7fffffffdfa0  ← rbp
 ──────────
──────────
|                 | <<─ sub로 만든 공간이 지역 변수 공간으로 쓰는 메모리 공간
 ──────────
──────────
|                 | 0x0x7fffffffdf90 ← rsp
 ──────────
```

ㄴ 이런 구조 때문에 스택은 아래로 자란다고 한 것이다!!

```
(gdb) x $rbp
0x7fffffffdfa0: 0x00000000
(gdb) x $rsp
0x7fffffffdf90: 0xffffe090
```

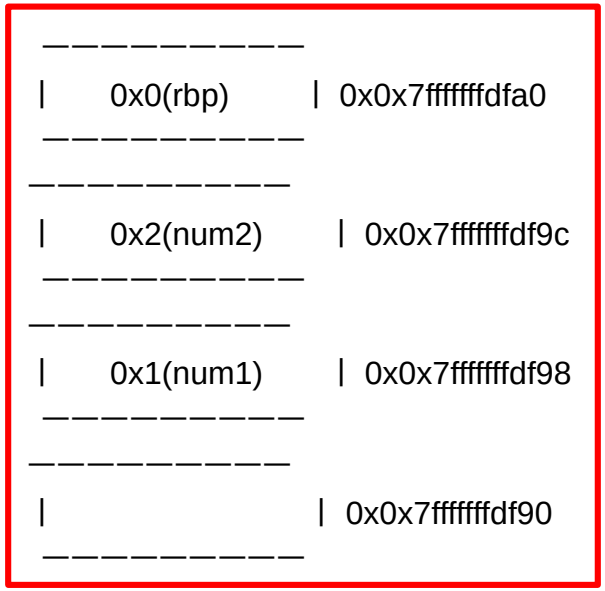
: 현재 스택의 최상위 rsp 90
현재 스택의 기준점 rbp a0

2. if문에 대한 기계어 분석 (test_if.c) - (4)

```
   0x0000555555555155 <+12>:    movl   $0x1,-0x8(%rbp)
   0x000055555555515c <+19>:    movl   $0x2,-0x4(%rbp)
=> 0x0000555555555163 <+26>:    mov    -0x8(%rbp),%eax
   0x0000555555555166 <+29>:    cmp    -0x4(%rbp),%eax
```

자, 이번에는 movl $0x1, -0x8(%rbp)이 보인다.
L 이 들어가면 4바이트 처리를 하겠다는 의미 이며
Q 가 들어가면 8바이트 처리를 하겠다는 뜻이 된다.
여튼, 위 명령어의 의미는 **rbp를 기준으로 8바이트 뺀 자리에 0x2을 복사한다는 의미** 이다.
그리고 movl $0x2, -0x4(%rbp), **rbp를 기준으로 4바이트 뺀 자리에 0x1을 복사한다는 의미**.

그림으로 나타내 보면 아래와 같다.




```
(gdb) x $rbp
0x7fffffffdfa0: 0x00000000
(gdb) x $rsp
0x7fffffffdf90: 0xffffe090
(gdb) x $rbp-8
0x7fffffffdf98: 0x00000001
(gdb) x $rbp-4
0x7fffffffdf9c: 0x00000002
```

ㄴ 그림 처럼 rbp-8에 0x1 배치, rbp-4에 0x2 배치

```c
#include <stdio.h>

int main(void)
{
        int num1 = 1, num2 = 2;


        if(num1 > num2)
        {
                printf("num1(%d)가 num
        }
        else
        {
                printf("num2(%d)가 num
        }

        return 0;
}
```

ㄴ 원소스의 num1에 1을
num2에 2를 넣는 과정.

2. if문에 대한 기계어 분석 (test_if.c) - (5)

```
   0x000055555555515c <+19>:    movl   $0x2,-0x4(%rbp)
   0x0000555555555163 <+26>:    mov    -0x8(%rbp),%eax
=> 0x0000555555555166 <+29>:    cmp    -0x4(%rbp),%eax
   0x0000555555555169 <+32>:    jle    0x555555555186 <main+61>
   0x000055555555516b <+34>:    mov    -0x4(%rbp),%edx
   0x000055555555516e <+37>:    mov    -0x8(%rbp),%eax
```

```c
#include <stdio.h>

int main(void)
{
        int num1 = 1, num2 = 2;


        if(num1 > num2)
        {
                printf("num1(%d)가 num
        }
        else
        {
                printf("num2(%d)가 num
        }

        return 0;
}
```

ㄴ 원소스의 num1 > num2 비교 후 else로 점프

자, 이번에는 cmp -0x4(%rbp) **[source]** , %eax **[dest]**
는 rbp 기준 -4바이트 위치(9c)의 값(num2)과 eax값(이전에 num1의 1을 넣어둠)을 비교 합니다.

그림으로 나타내 보면 아래와 같다.

0x0(rbp)	0x0x7fffffffdfa0
0x2(num2)	0x0x7fffffffdf9c
0x1(num1)	0x0x7fffffffdf98
	0x0x7fffffffdf90

cmp의 경우 보통 jmp 명령어와 같이 사용하게 되는데,
Jle = jump less or equal 로 작거나 같을 때 점프한다는 의미 이다.**(dest가 기준이다)**

num2(source) , num1(dest)

dest인 num1(1)은 souce인 num2(2)보다 작으므로 jle 명령어가 발동하여 **0x55555555186** 주소로 점프 합니다.

2. if문에 대한 기계어 분석 (test_if.c) - (6)

```
   0x0000555555555186 <+61>:    mov    -0x8(%rbp),%edx
   0x0000555555555189 <+64>:    mov    -0x4(%rbp),%eax
   0x000055555555518c <+67>:    mov    %eax,%esi
=> 0x000055555555518e <+69>:    lea    0xe9b(%rip),%rdi
```

0x000055555555186 으로 점프 되었고, mov 를 통해
-0x8(%rbp)의 값인 0x1을 edx에
-0x4(%rbp)의 값인 0x2를 eax에
Eax 값을 esi에 배치 합니다.

```
(gdb) p/x $edx
$4 = 0x1
(gdb) p/x $eax
$5 = 0x2
(gdb) p/x $esi
$6 = 0x2
```

0x0(rbp)	0x0x7fffffffdfa0
0x2(num2)	0x0x7fffffffdf9c
0x1(num1)	0x0x7fffffffdf98
	0x0x7fffffffdf90

```
   0x000055555555518c <+67>:    mov    %eax,%esi
=> 0x000055555555518e <+69>:    lea    0xe9b(%rip),%rdi        # 0x555555556030
   0x0000555555555195 <+76>:    mov    $0x0,%eax
```

ㄴ 다음으로 lea는 배열 명령어 이며 0xe9b(%rip)값이 %rdi에 배치 되었다고 보면 된다.

```
(gdb) x $rdi
0x555555556030: 0x326d756e
```

```
   0x000055555555518c <+67>:    mov    %eax,%esi
=> 0x000055555555518e <+69>:    lea    0xe9b(%rip),%rdi        # 0x555555556030
   0x0000555555555195 <+76>:    mov    $0x0,%eax
```

ㄴ 다음으로 0의 값을 eax에 배치 한다.

```
(gdb) x $eax
0x0:    Cannot
```

2. if문에 대한 기계어 분석 (test_if.c) - (7)

```
   0x0000555555555195 <+76>:    mov    $0x0,%eax
=> 0x000055555555519a <+81>:    callq  0x555555555050 <printf@plt>
   0x000055555555519f <+86>:    mov    $0x0,%eax
   0x00005555555551a4 <+91>:    leaveq
   0x00005555555551a5 <+92>:    retq
```

Printf 함수는 ni로 넘긴다.

Callq 0x555555555050 은 매우 중요한 연산이다.
Call 은 기본적으로 push + jmp로 구성되어 있다.
함수 호출이 끝난 이후에 실행해야 할 어셈블리 명령어의 주소값을 psuh로 저장한다.
이후 함수 호출을 수행하기 위해 jmp를 수행한다.

결국 아래와 같은 메모리를 가지게 된다.

0x0(rbp)	0x0x7fffffffdfa0
0x2(num2)	0x0x7fffffffdf9c
0x1(num1)	0x0x7fffffffdf98
	0x0x7fffffffdf90
0x000055555555519f(복귀주소)	0x0x7fffffffdf88

```
(gdb) x $rsp
0x7fffffffdf88: 0x5555519f
(gdb) x $rsp+4
0x7fffffffdf8c: 0x00005555
```

ㄴ 스택의 최상위 값인 rsp가 88이 되고 안에
복귀 주소인 0x000055555555519f 이 저장된 것을 볼 수 있다.
(복귀 주소가 너무 길어서 rsp+4 명령어 사용해서 뷰!)

2. if문에 대한 기계어 분석 (test_if.c) - (8)

```
   0x0000555555555195 <+76>:    mov    $0x0,%eax
=> 0x000055555555519a <+81>:    callq  0x555555555050 <printf@plt>
   0x000055555555519f <+86>:    mov    $0x0,%eax
   0x00005555555551a4 <+91>:    leaveq
   0x00005555555551a5 <+92>:    retq
```

Mov 로 0x0을 eax 에 배치 시키고

```
(gdb) x $eax
0x0:    Canno
```

Leaveq 는 스택해제 명령어 이다.

```
(gdb) x $rsp
0x7fffffffdfa8: 0xf7deb0b3
```

0x0(rbp)	0x0x7fffffffdfa0
0x2(num2)	0x0x7fffffffdf9c
0x1(num1)	0x0x7fffffffdf98
	0x0x7fffffffdf90
0x000055555555519f(복귀주소)	0x0x7fffffffdf88