

* 포인터의 크기

pointer 의 크기는 컴퓨터 비트에 따라 달라진다. (컴퓨터의 산술 연산이 ALU 에 의존적)

64bit 컴퓨터의 경우 1byte 가 8bit 이므로 총 8byte 로 구성이 되어있다.

32bit 컴퓨터의 경우 1byte 가 8bit 이므로 총 4byte 로 구성이 되어있다.

16bit 컴퓨터의 경우 1byte 가 8bit 이므로 총 2byte 로 구성이 되어있다.

이와 관련하여 왜 32bit 컴퓨터 상에는 램의 용량이 4GB 밖에 쓰지 못하는지 알 수 있다.

2^{10} byte = 1KB

2^{10} KB = 1MB

2^{10} MB = 1GB

32bit 컴퓨터는 2^{32} 가지 표현(총 주소 할당)이 가능하다.

따라서 4×2^{30} byte = 4GB 이므로, 일반적으로 32bit 컴퓨터는 총 4GB 의 램을 인식할 수 있다.

*2 진수 16 진수 변환 정리

2 진수와 16 진수의 변환법은 간단하다. 최대 공약수를 구하는 공식과 비슷하다.

예를 들면,

십진수 31 이라는 숫자가 있을 경우,

2/31

2/15...1

2/7 ...1

2/3 ...1

2/1 ...1

1

위 숫자를 그대로 이어 붙이면 11111(2) 가 된다.

검산을 해보면 $2^4 + 2^3 + 2^2 + 2^1 + 1$ 이므로 31 이 된다.

16 진수 변환법은 2 진수를 4 개씩 끊어서 십진수 변환 뒤 해당 값을 16 진수로 변환한다.

31 이라는 숫자는 2 진수로 11111(2)이다.

4 개씩 끊어보면 0001 1111(2) 로 바뀌게 된다.

해당 값을 각각 십진수로 보면, 1 15 가 된다. 각각의 값을 16 진수 변환하면 1 f (16)이다.

(16 진수는 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f 순으로 나아간다.)

* 기계어 분석

```
#include <stdio.h>
int func1(int i){
    return i+3;
}

int main(void){
    int a=3,res;
    res = func1(a);
    printf("after func value is %d\n",res);

    return 0;
}
```

와 같은 c program 을 구성하였을 경우 디버그를 활용하여 어셈블리어를 보면,
<메인>

```
0x0000000000400535 <+0>:    push  %rbp // *rsp=rbp
0x0000000000400536 <+1>:    mov   %rsp,%rbp // rbp=rsp
0x0000000000400539 <+4>:    sub   $0x10,%rsp// rsp-=16
0x000000000040053d <+8>:    movl  $0x3,-0x8(%rbp) // *(rbp-8)=3
0x0000000000400544 <+15>:   mov   -0x8(%rbp),%eax// eax = 3
0x0000000000400547 <+18>:   mov   %eax,%edi // edi=eax
0x0000000000400549 <+20>:   callq 0x400526 <func1>
                               // *rsp= 0x40054e and jump 0x400526
0x000000000040054e <+25>:   mov   %eax,-0x4(%rbp)
0x0000000000400551 <+28>:   mov   -0x4(%rbp),%eax
0x0000000000400554 <+31>:   mov   %eax,%esi
0x0000000000400556 <+33>:   mov   $0x4005f4,%edi
0x000000000040055b <+38>:   mov   $0x0,%eax
0x0000000000400560 <+43>:   callq 0x400400 <printf@plt> //printf call
0x0000000000400565 <+48>:   mov   $0x0,%eax //return 0
0x000000000040056a <+53>:   leaveq
0x000000000040056b <+54>:   retq // goto os
```

<함수부>

```
0x0000000000400526 <+0>:    push  %rbp // *rsp=rbp
0x0000000000400527 <+1>:    mov   %rsp,%rbp // rbp=rsp
0x000000000040052a <+4>:    mov   %edi,-0x4(%rbp) // *(rbp-4)=edi=eax=3
0x000000000040052d <+7>:    mov   -0x4(%rbp),%eax // eax=*(rbp-4)
0x0000000000400530 <+10>:   add   $0x3,%eax // eax+=3
0x0000000000400533 <+13>:   pop   %rbp // rbp=*rsp
0x0000000000400534 <+14>:   retq // rip=*&rsp=0x40054e
```

주석 처리한 것처럼 실행이 된다.

이와 같은 메모리 형태를 그림으로 표현하면 다음과 같다.

