C - HW4

임베디드스쿨1기

Lv1과정

2020. 08. 21

박하늘

1. 포인터

1) 메모리 주소값을 담는 변수
+ 포인터에는 자료형에 대한 정보가 함께 있다.

1-1. ampersand(&) : 주소연산자
- 메모리 주소 명시
- 포인터를 사용하기 위해 먼저 그 주소값을 알아내는 과정이 필요한데, 특정 변수의 포인터를 구하기 위해서 “&”를 사용한다.

1-2. *: 참조연산자
- 해당 변수가 가리키는 메모리 주소의 값
- 포인터를 통해서 기억 공간에 사용하기 위해서 사용한다.
- 참조 연산자로 포인터를 사용하는 방법: 포인터가 가리키는 기억공간을 사용하거나, 기억공간의 값을 사용한다.

```
int a = 10, b = 20;
*&a = *&b                    //변수 b에 저장된 값을 변수 a의 기억 공간에 기억
printf(“a의 값: %d”,a);
```

→ *&a: 기억공간을 사용 / *&b: 기억공간의 값을 사용

1. 포인터

1) 메모리 주소값을 담는 변수

1-1. ampersand(&) : 메모리 주소 명시

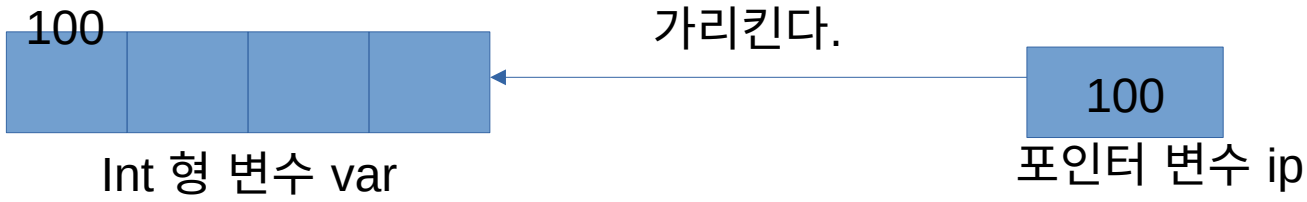
1-2. *: 해당 변수가 가리키는 메모리 주소의 값

```c
#include <stdio.h>

int main(void){
        int var = 20;   //포인터를 구할 변수
        int *ip;        //포인터를 저장할 변수
        ip = &var;      //var의 포인터를 구하여 포인터 변수 ip에 저장한다.

        printf("Address of var variable: %x\n", &var);
        printf("Address stored in ip variable: %x\n", ip);
        printf("Value of *ip variable: %d\n", *ip);     //해당 주소의 값을 불러옴

        return 0;
}
```

```
Address of var variable: 1957528c
Address stored in ip variable: 1957528c
Value of *ip variable: 20
```

ip가 포인터를 저장하면

가리킨다.



1. 포인터

1) 메모리 주소값을 담는 변수

1-1. ampersand(&) : 메모리 주소 명시

1-2. *: 해당 변수가 가리키는 메모리 주소의 값

```c
#include <stdio.h>

const int MAX = 3;

int main(void){
        int var[] = {10, 100, 200};     //배열 초기화
        int i, *ptr;                    //인덱스, 포인터 변수 선언

        ptr = var;                      //배열의 시작주소를 포인터 변수에 저장

        for (i=0; i < MAX ; i++){

                printf("Address of var[%d] = %x\n", i, ptr);
                printf("value of car[%d] = %d\n", i ,*ptr);

                ptr++;
        }
        return 0;
}
```

```
Address of var[0] = a4ba569c
value of car[0] = 10
Address of var[1] = a4ba56a0
value of car[1] = 100
Address of var[2] = a4ba56a4
value of car[2] = 200
```

Int *ptr = &var[] //배열의 첫번째 항목주소가 전체 배열의 시작주소

Int *ptr = &*(var+0) //배열을 포인터 기법으로 변경

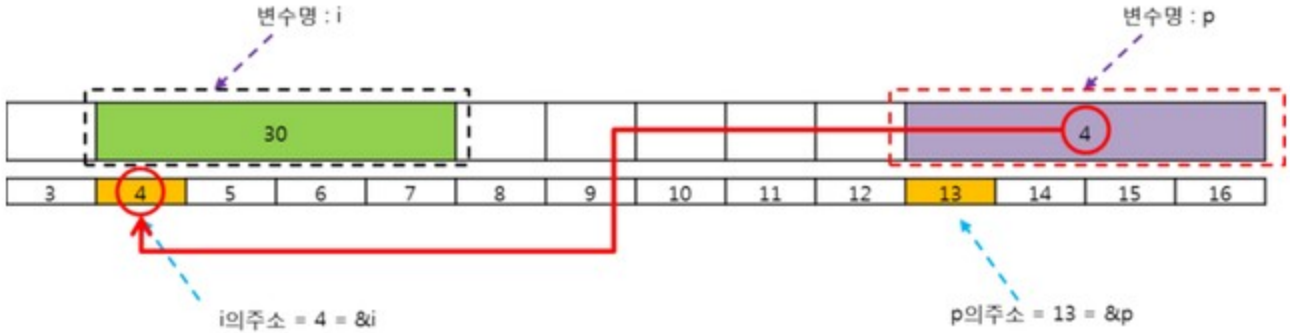
= &*var

= var

2. 이중 포인터

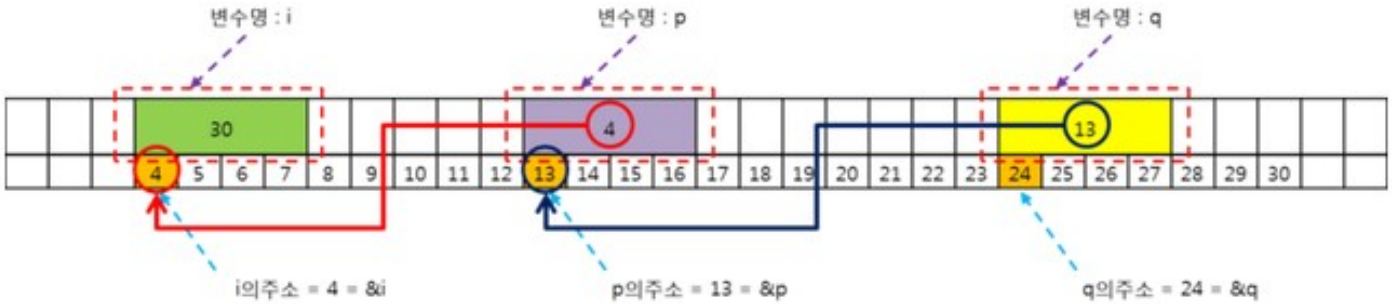
1) 포인터 변수의 주소값을 값이다.

```
int I = 30;
int * p = &i;
```



→ p는 i의 주소값 4를 가지고 있고, p가 i를 포인팅하고 있다. 따라서 , p를 이용해 i를 조작할 수 있다. 이를 한번 더 밟은 것이다.

```
int I = 30;
int *p = &i;
int **q = &p;
```



*p 는 i 를 나타낸다.
*q 는 p 를 나타낸다.
그러므로

*(*q) 는 i 를 나타낸다.
결론적으로
**q = *p = i 가 된다.

→ q는 포인터인데, q는 포인터변수의 주소를 가지고 있다. 즉, p의 주소값을 가지고 있다. 따라서 q는 p를 포인팅하고, p는 i를 포인팅한다. (**q = *p = i)

3. 포인터배열

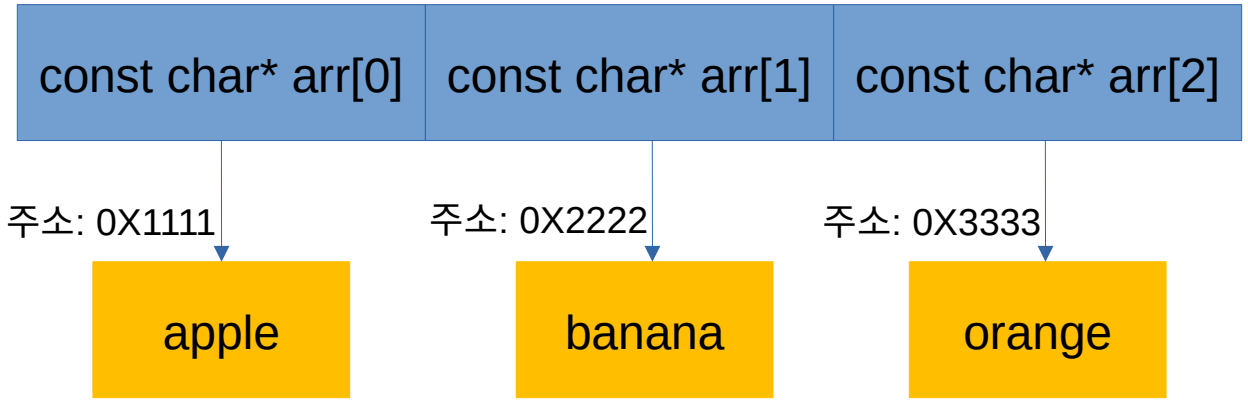
1) (포인터)(배열) = 포인터배열은 포인터로 이루어진 배열이라는 뜻이다.
아래 그림에서 자료형이 char* (포인터)인 배열, 그 배열의 요소의 개수가 3개

```c
#include <stdio.h>

int main(void){
        const char* arr[3];
        int i;

        arr[0] = "apple";
        arr[1] = "banana";
        arr[2] = "orange";

        for (i = 0; i < 3; i++)
        {
                printf("arr[%d] 는 %s\n",i, arr[i]);
        }
        return 0;
}
```

```
arr[0] 는 apple
arr[1] 는 banana
arr[2] 는 orange
```

2) 그림표현

Const char* arr[3]



4. 배열포인터

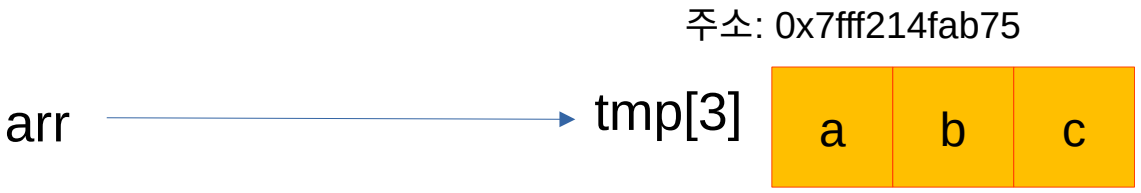
1) (배열)(포인터) = 배열포인터는 배열만 가리키는 하나의 포인터라는 뜻이다.
아래 그림에서 char타입의 인덱스 3개를 가지고 있는 배열을 가리키는 포인터,
그냥 하나의 포인터

```c
#include <stdio.h>
int main(void)
{
        int i;
        char (*arr)[3];

        char tmp[3] = {'a','b','c'};
        printf("tep[3]의 주소:%p\n",tmp);

        arr = &tmp;

        printf("arr의 주소: %p\n 문자열:", arr);
        for (i = 0; i <(int)sizeof(*arr); i++)
        {
                printf("%c", (*arr)[i]);
        }
        printf("\n");
        return 0;
}
```

```
tep[3]의 주소:0x7fff214fab75
arr의 주소: 0x7fff214fab75
 문자열:abc
```

2) 그림표현

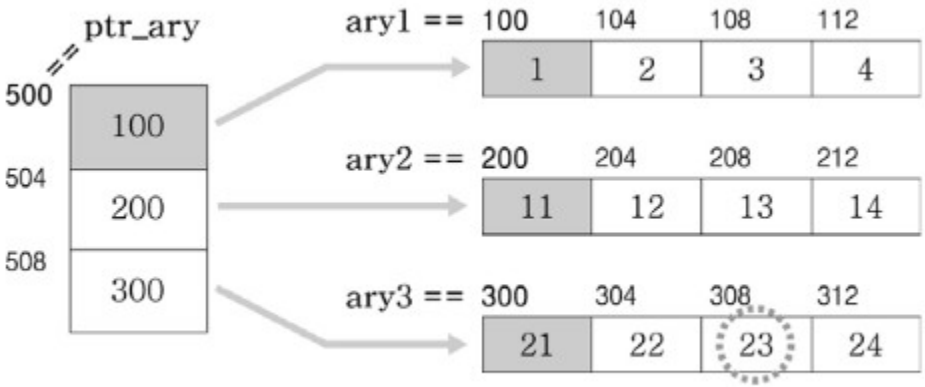
Char (*arr)[3]



4. 배열포인터 - 2차원 배열 표현 방법(1/3)

1) 1차원 배열의 배열명을 포인터배열에 저장하면 포인터배열을 2차원 배열처럼 사용할 수 있다.

```
int ary1[4]={1, 2, 3, 4};
int ary2[4]={11, 12, 13, 14};
int ary3[4]={21, 22, 23, 24};
int *ptr_ary[3]={ary1, ary2, ary3};
// 각 배열명을 포인터배열에 초기화한다.
```



(각 기억공간의 주소값은 설명의 편의를 위해 임의로 붙인 것입니다.)

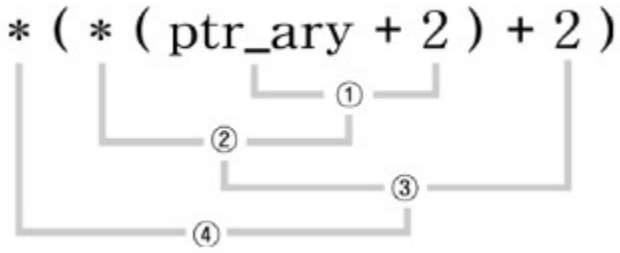
– ary3배열의 세 번째 배열요소(23값)을 참조하는 과정

1. 먼저 ptr_ary 배열의 세번째 배열요소를 참조한다.
   → ptr_ary[2]

2. 참조된 배열요소 ptr_ary[2]는 배열명 ary3을 저장한 포인터변수이므로 배열명처럼 사용하여 ary3의 세번째 배열요소를 참조한다.
   → printf(“%d”,ptr_ary[2][2]);

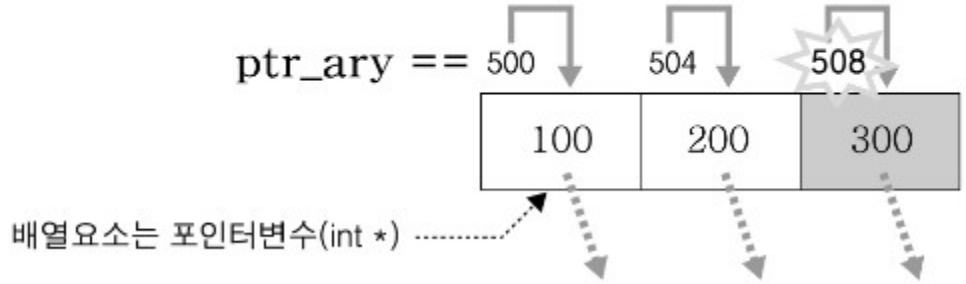
4. 배열포인터 - 2차원 배열 표현 방법(2/3)

2) ptr_ary[2][2]가 참조되는 과정의 주소값을 계산해보자
-일단 포인터표현으로 바꾸고 연산순서를 따라간다.



– ①번 연산: 포인터배열의 세 번째 배열요소를 가리키는 포인터가 구해진다.

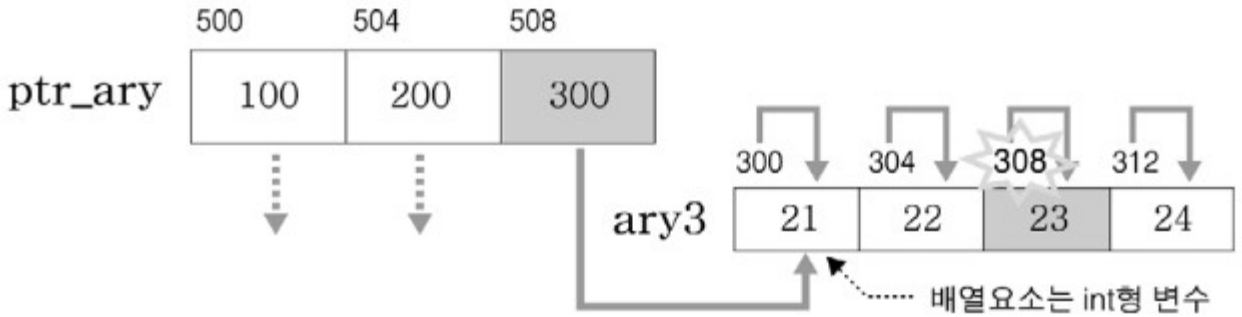
$$ptr\_ary+2 = ptr\_ary+(2*sizeof(ptr\_ary[0])) = 500+(2*4) = 508$$



4. 배열포인터 - 2차원 배열 표현 방법(3/3)

- ② 번 연산: 포인터배열의 세번째 배열요소의 값 300번지가 구해진다.
- ③ 번 연산: ary3배열의 세번째 기억공간을 가리키는 포인터가 구해진다.

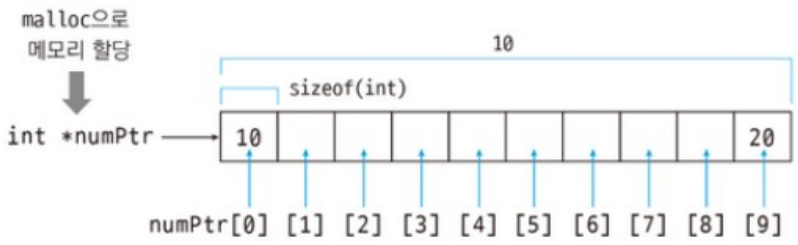
300+2 => 300+(2*sizeof(ary3[0])) => 308



- ④ 번 연산: 308번지는 포인터이므로 참조연산을 수행하면 값 23이 참조된다.

5. 동적할당 malloc, free

1) 포인터에 할당된 메모리를 배열처럼 사용한다. 포인터배열.
2) 자료*포인터이름 = malloc(sizeof(자료형)*크기);
3) malloc함수: 동적으로 할당하는 함수 (Heap영역에 메모리를 할당)
4) malloc은 메모리만 할당하는 함수이므로 개발자가 어떠한 데이터 형을 저장했는지 예측할 수 없다.
5) int형 데이터를 저장하기 위해서는 리턴되는 void*을 int*로 반환해야 한다.
6) 동적 할당 후 더 사용할 필요가 없다면 꼭 free함수로 메모리를 해제시켜준다.




```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
        int* numPtr = (int*)malloc(sizeof(int)*10);//int 10개 크기만큼 동적 메모리 할당

        numPtr[0] = 10;
        numPtr[1] = 20;

        printf("%d\n",numPtr[0]);//배열처럼 인덱스로 접근하여 값 할당
        printf("%d\n",numPtr[9]);

        free(numPtr);//동적으로 할당한 메모리 해제

        return 0;
}
```

```
10
20
```

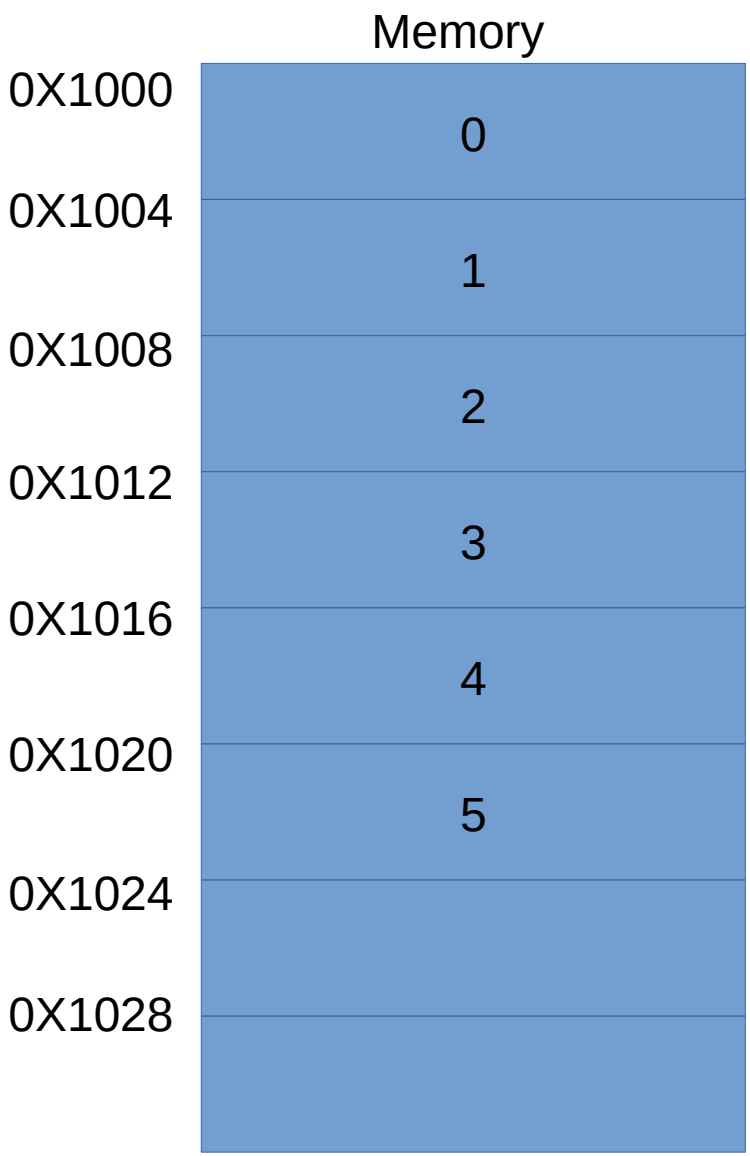
heap은 큰 크기의 데이터를 담고자 동적으로 할당하는 메모리 공간을 지칭한다.
– 장점: 상황에 따라 원하는 크기만큼의 메모리가 할당되므로 경제적이며, 이미 할당된 메모리라도 언제든지 크기를 조절할 수 있다.
– 단점: 더 이상 사용하지 않을 때 명시적으로 메모리를 해제해 주어야 한다.

HW. 포인터 - 메모리 맵핑 시키기

아래에 해당하는 값을 넣으시오

```
int main(void)
{
        int a[2][3] = {{0,1,2},{3,4,5}};
}
```

1) &a = 0x1000
→ a의 시작주소
2) &a+1 = 0x1024
→ a의 배열의 크기가 다 끝난 다음의 처음 시작 주소
3) a+1 = 0x1012
→ a[0][0] = a
→ a[1][0] = a+1
4) *a = 0x1000
→ a의 배열을 가리키는 포인터: a배열의 주소
5) *a+1 = 0x1004:
→ *a배열의 주소에 int형 4byte를 더한
6) **a = 0
→ a의 배열의 주소가 가리키는



6. 구조체 struct

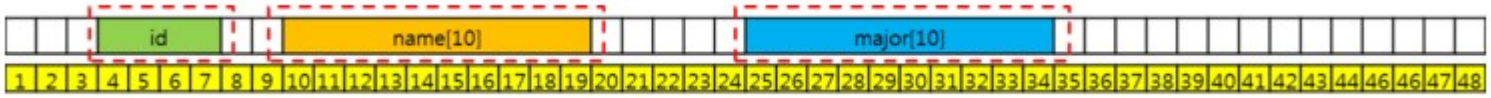
1) 정의: C언어에서 유일한 객체지향언어의 클래스(Class)개념을 가진 자료형(data type)이다.

```
struct myName
{
        원하는 자료형;
        원하는 자료형;
        원하는 자료형;
        ...
}
```

```
struct student
{
        int id;
        char name [10] ;
        char major [10] ;
}
```

```
Typedef struct Student{
        char name[20];
        int age;
}Student;
```

만약 구조체를 사용하지 않으면 변수들을 선언하면 아와 같이 빈 공간에 있는 곳에 할당 된다.



하지만, 구조체를 정의하면 id, name, major 3가지를 모두 묶어서 A라는 하나의 이름으로 관리 가능.

2) 구조체 멤버접근:

구조체변수명.맴버명

```
struct student A;
A.id = 20150921;
strcpy(A.nmae , "전지현");
strcpy(A.major, "영화학");
```

3) 초기화

```
struct student A = {20150921, "전지현", "영화학"};
```

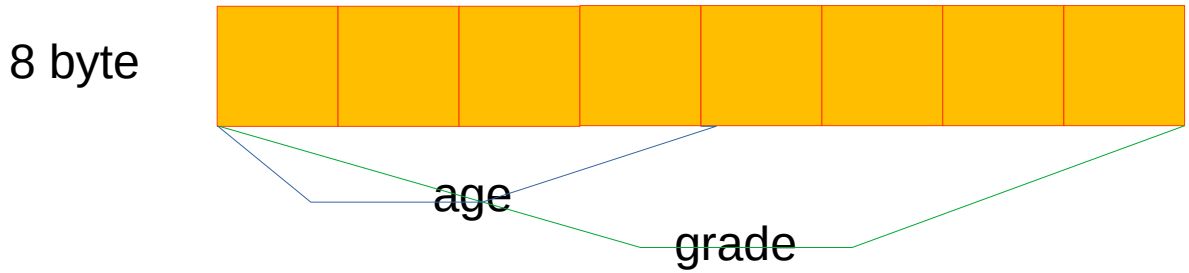
7. 공용체 union

1) union은 struct와 달리 메모리를 공유한다. 메모리가 부족할때 많이 사용된다.

```c
#include <stdio.h>
union student{
        int age;
        double grade;
}person;
int main(void)
{
        union student person = {20};

        printf("나이%d\n", person.age);
        person.grade = 90.25;
        printf("평균: %lf\n", person.grade);
        printf("평균사용중 나이: %d\n", person.age);
        return 0;
}
```

```
나이20
평균: 90.250000
평균사용중 나이: 0
```

첫번째 맴버변수 age만 20으로 초기화가 된다.
또한, age를 사용하다가 grade를 사용하면 age값이 사라진다.
→ 메모리 공간을 공유하기 때문에 유니온은 맴버변수 한번에 하나씩만 사용할 수 있다.



8. bit fields

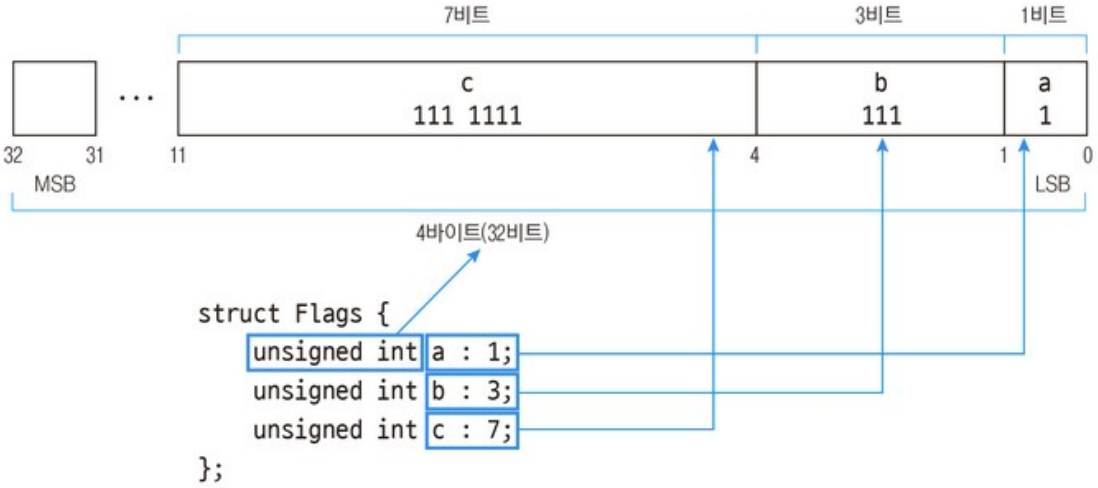
1) 구조체 비트 필드를 사용하면 구조체 멤버를 비트 단위로 저장할 수 있다.

```
struct 구조체이름 {
    정수자료형 멤버이름 : 비트수;
};
```

```
#include <stdio.h>

struct Flags{
        unsigned int a : 1;
        unsigned int b : 3;
        unsigned int c : 7;
};

int main()
{
        printf("%d\n",sizeof(struct Flags));
        return 0;
}
```

```
4
```

2) 비트 필드의 멤버를 unsigned int로 선언했으므로 구조체의 크기는 4가 된다.



3) 비트 필드의 각 멤버는 최하위 비트(Least Significant Bit, LSB)부터 차례대로 배치된다. 따라서 a가 최하위 비트에 오고 나머지 멤버들은 각각 상위비트에 배치된다.

HW. 구조체 + 공용체 + bitfields Register Flag

1) 비트필드는 하위비트부터 순서대로 설정된다. 즉, 상위비트에 먼저 값이 들어가야 하는 경우에는 선언 순서를 반대로 해야 정확한 값을 입력할 수 있다.

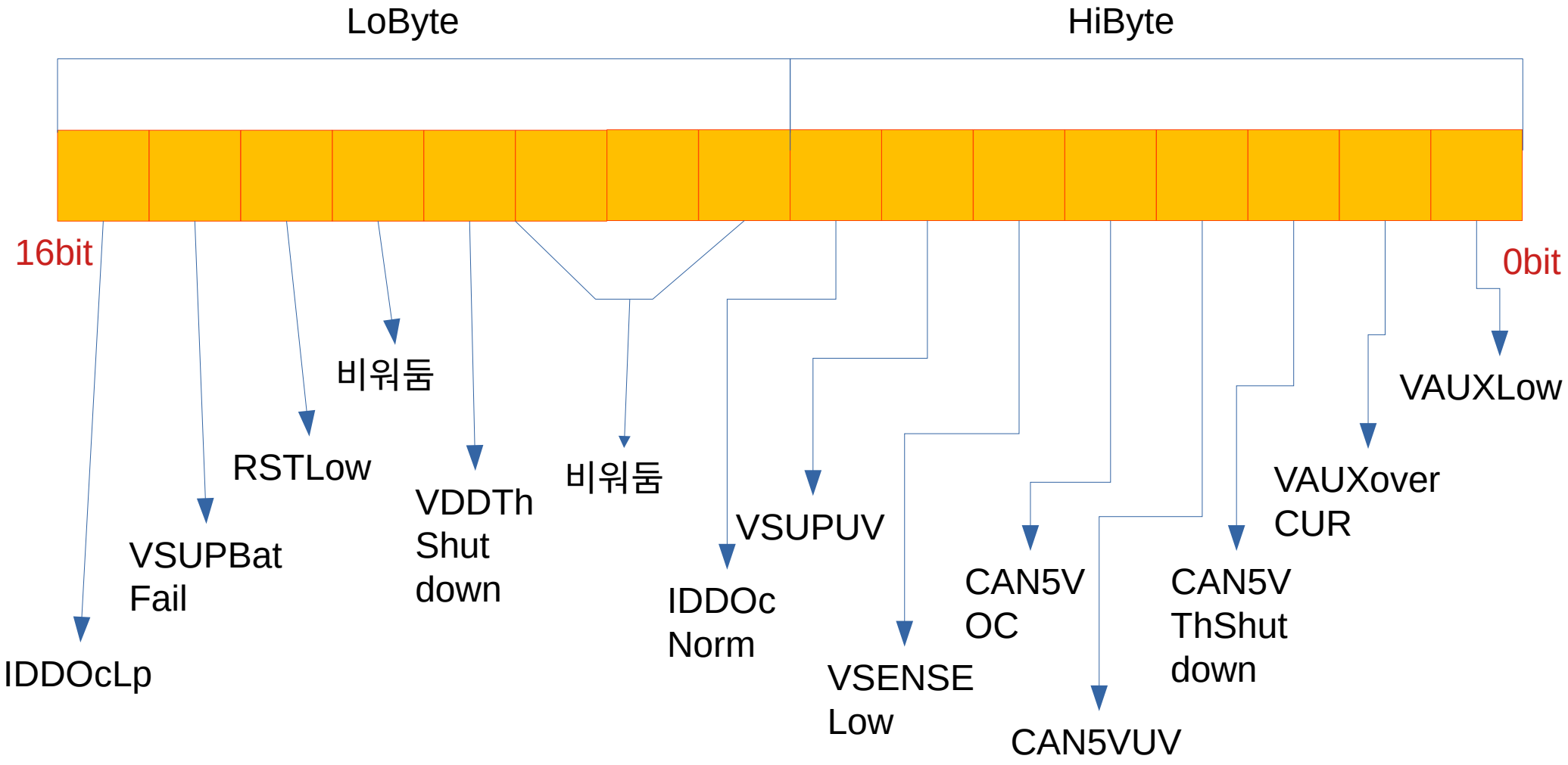
2) 주소연산이 불가능하다. 메모리상의 주소는 바이트단위이므로 비트단위인 비트필드의 경우 주소연산자를 붙일 수 없다. Scanf 사용불가능함.

3) 선언에서 필드명을 적어주지 않은 필드의 경우 사용이 불가능하게 된다. 예를들어, VDDThShutdown의 앞뒤 3/1 비트는 사용불가능하다.

```
typedef union
{
        UINT16 ALL;
        struct
        {
                UINT16 Hibyte : 8;
                UINT16 Lowbyte : 8;
        }Byte;

        struct
        {
                UINT16 VAUXLow :1;
                UINT16 VAUXoverCUR :1;
                UINT16 CAN5VThShutdown :1;
                UINT16 CAN5VUV :1;
                UINT16 CAN6VOC:1;
                UINT16 VSENSELow :1;
                UINT16 VSUPUV :1;
                UINT16 IDDOcNorm :1;

                UINT16 : 3;
                UINT16 VDDThShutdown :1;
                UINT16 :1;
                UINT16 RSTLow :1;
                UINT16 VSUPBatFail :1;
                UINT16 IDDOcLp :1;
        }Bit;
}USBCREGFlag;
```

HW. 구조체 + 공용체 + bitfields Register Flag

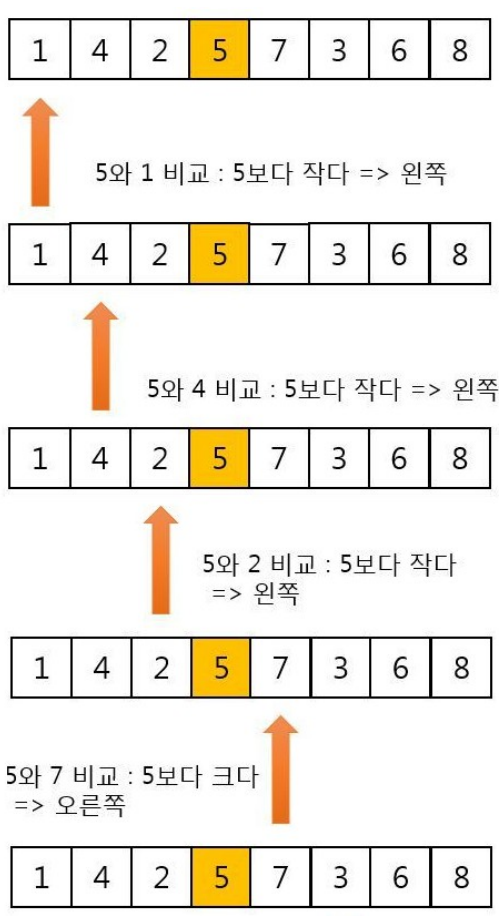
4)그림

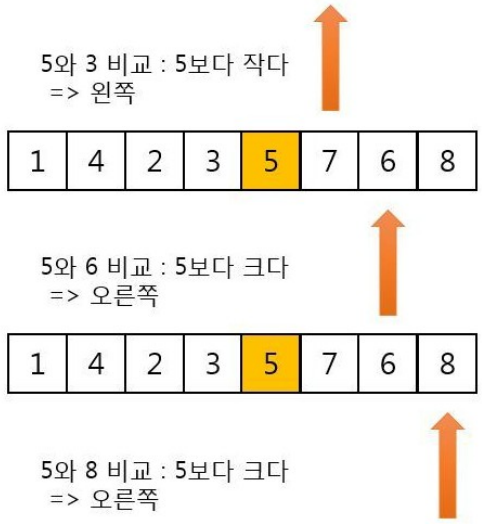


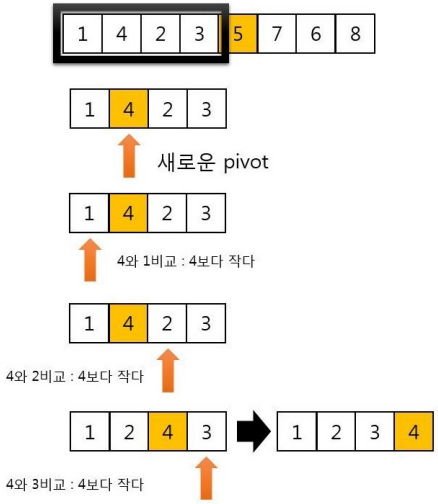
HW. Quick sort란?

1) sorting 알고리즘 중 가장 효율적이고 빠른 방식
→ 기본적으로 o(N^2)으로 정렬하는 알고리즘은 순회를 하면서 복잡해진다.
반면 퀵소트는 직관적으로 비교해 반씩 줄여나가는 재귀를 이용한 분할정복 알고리즘이다.
2) idea: 특정한 기준을 pivot이라고 하고, 이 pivot을 기준으로 pivot보다 작으면 왼쪽으로, 크면 오른쪽으로 이동한다.

.





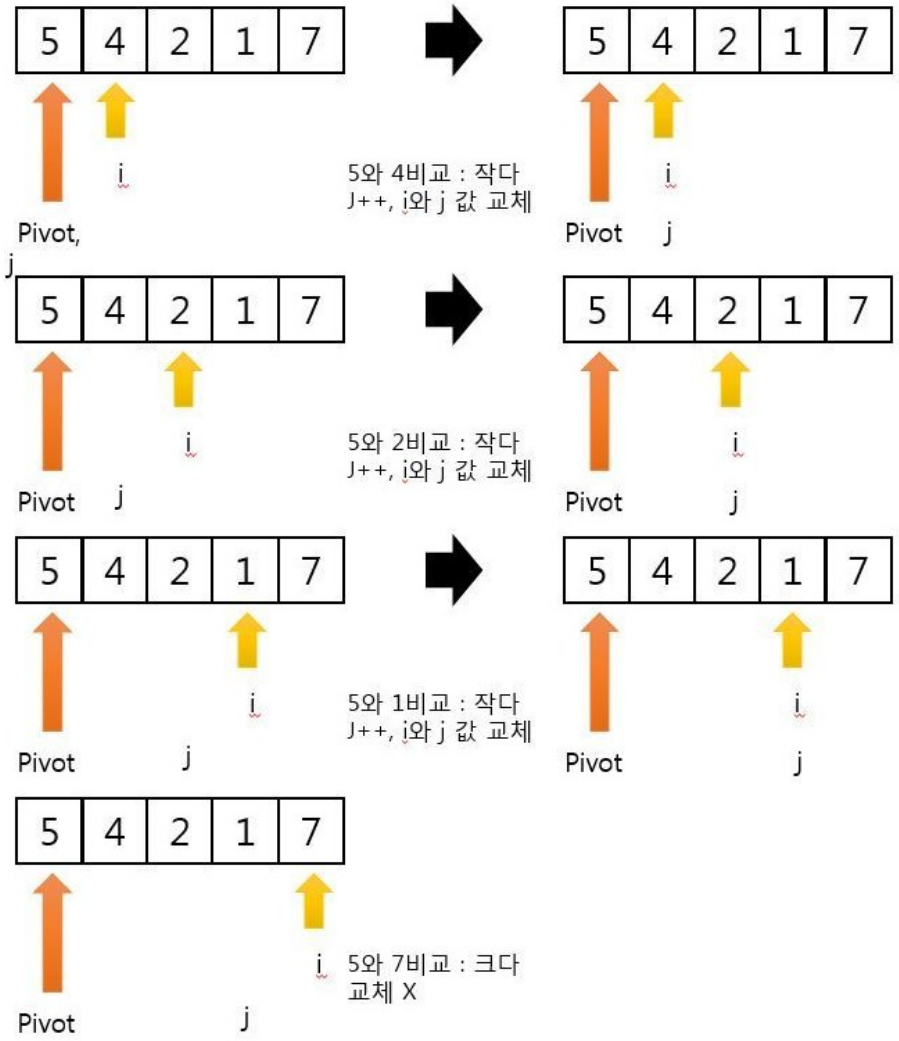


이제 이렇게 되면 5의 왼쪽과
오른쪽끼리는 섞을 필요가 없다.

남은 구간도 나누어서
분할처리하여 정렬하면 된다.

HW. Quick sort

3) 실제 구현에서는, Pivot과 j는 첫번째로 두고, i는 j+1로 두었다.
i와 pivot을 비교하여 작으면 pivot의 왼쪽으로 보낸다. 왼쪽으로 보내는 것을 j 변수로 설정했다. (pivot:기준, I:비교할 값, j: 교체할 값)



실제 4~7까지 pivot인 5와 비교하여 결과적으로 바뀌어지는 수가 없어진다. 이제 마지막 과정인 pivot과 j를 교체하자.

이제 나머지도 똑같은 방법으로 정렬하면 된다.