C - HW4

임베디드스쿨1기

Lv1과정

2020. 08. 21

김인겸

배열과 포인터 추가 내용 정리

일반적인 배열과 다차원 배열에서 포인터 개념이 어떻게 쓰이는지 비교하며 알아보자.

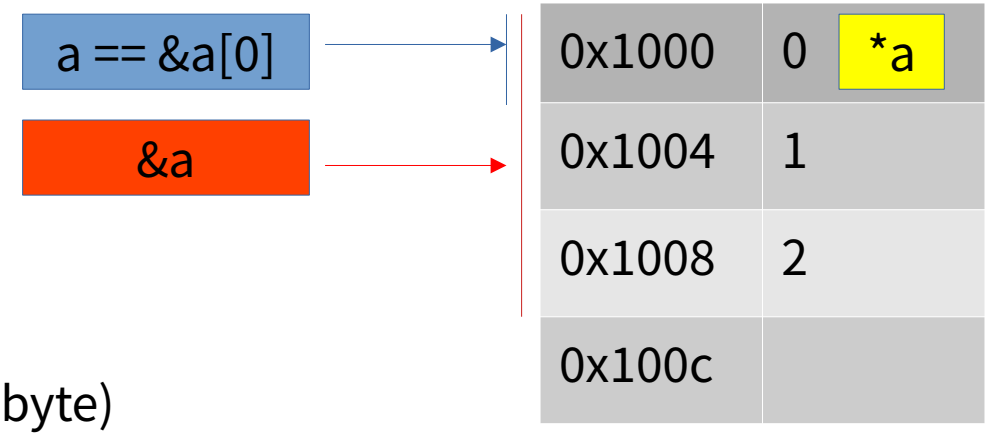
1. 일반적인 배열

int a[3] = {0, 1, 2};

- a의 의미는?
  a의 첫번째 인덱스를 가르키는 주솟값이다.
  a == &a[0] 이 성립한다.

- &a의 의미는?
  a라는 배열 전체 크기의 주소를 의미한다.(크기가 12byte)
  즉 a[0],a[1],a[2] 세 가지 값의 주소를 포괄하며
  그 주솟값을 대표하는 a[0]의 주솟값이 &a가 된다.

- *a의 의미는?
  a가 가르키는 주소의 실제 값을 의미한다.



배열과 포인터 추가 내용 정리

예제

```c
#include <stdio.h>

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

    printf("a 출력 : %p\n", a);
    printf("&a[0] 출력 : %p\n", &a[0]);
    printf("&a출력 : %p\n", &a);
    printf("a+1 값 : %p\n", a+1);
    printf("&a[0] +1 값 : %p\n", &a[0]+1);
    printf("&a+1 값 : %p\n", &a+1);

    return 0;
}
```

```
a 출력 : 0x7fff42b53c8c
&a[0] 출력 : 0x7fff42b53c8c
&a출력 : 0x7fff42b53c8c
a+1 값 : 0x7fff42b53c90
&a[0] +1 값 : 0x7fff42b53c90
&a+1 값 : 0x7fff42b53c98
```

편의상 주솟값 끝의 네 자리만 보자.
a, &a[0], &a의 주소는 모두 3c8c이다.

a+1은 4바이트 더해진 3c90
&a[0] +1 또한 4바이트 더해진 3c90
하지만, &a + 1 은 12바이트 더해진 3c98 (&a가 a배열 전체 크기의 주솟값을 의미하기 때문이다)

배열과 포인터 추가 내용 정리

2. 다차원 배열

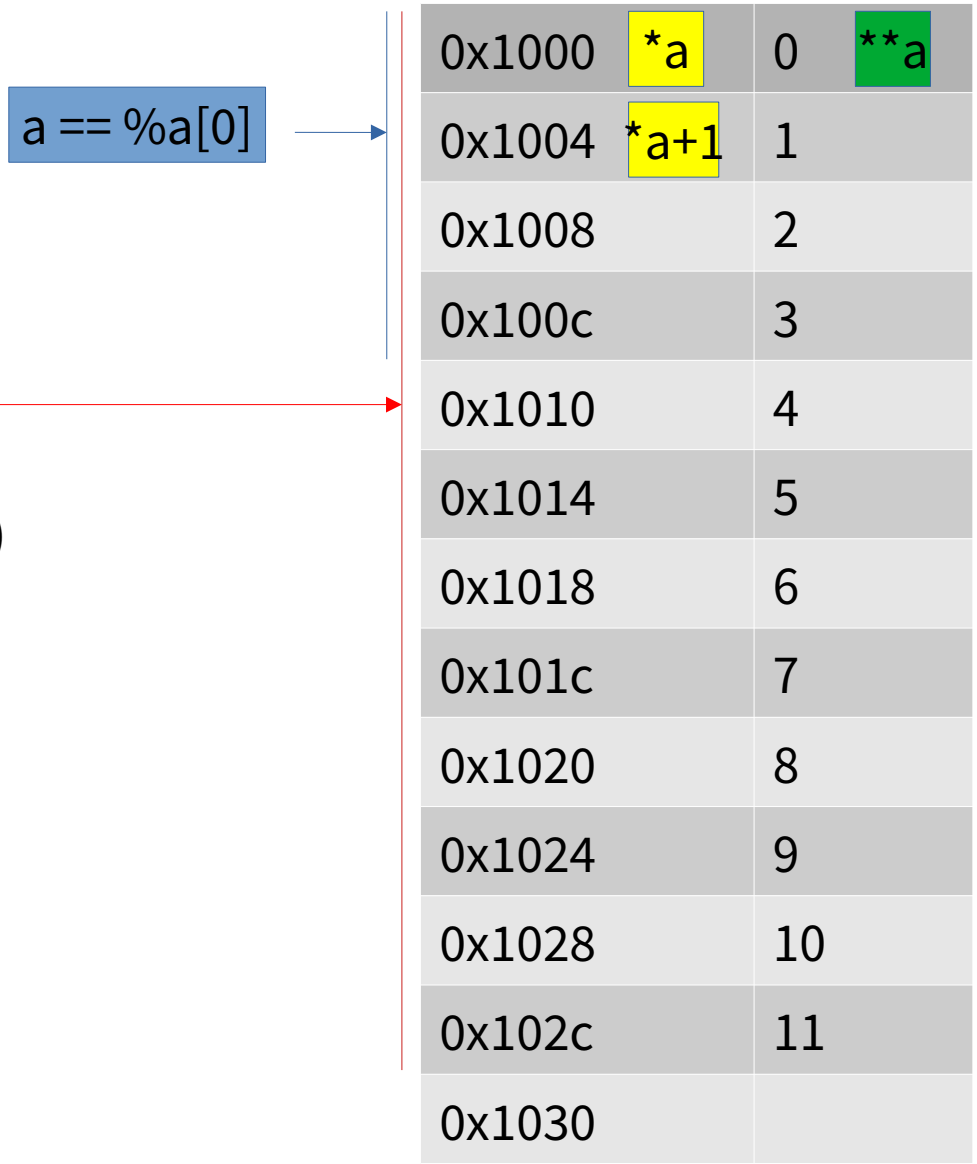
int a[3][4] = { {0,1,2,3}, {4,5,6,7}, {8, 9,10,11} };

- a의 의미는?
  a의 첫번째 인덱스를 가르키는 주솟값이다
  다차원 배열은 배열 안에 배열이 있는 개념이다.
  핵심은 a의 첫번째 인덱스가 배열이라는 점이다.
  16byte의 크기를 갖는다.

- &a의 의미는?
  a라는 배열 전체 크기의 주소를 의미한다.(크기가 48byte)

- *a의 의미는?
  a는 0x1000 ~ 0x100c를 가르키는 주솟값이므로
  *a는 제일 첫 번째 주소인 0x1000을 가르킨다

- **a의 의미는?
  *a의 값이 0x1000이므로
  **a는 0x1000이 가르키는 값인 0이다.



배열과 포인터 추가 내용 정리

```c
#include <stdio.h>

int main(void)
{
    int a[3][4] = { {0, 1, 2, 3} , {4, 5, 6, 7}, {8, 9, 10, 11} };

    printf("%p %p %p %p %p\n", &a, a, a[0], &a[0], &a[0][0]);

    printf("%p %p %p %p %p\n", &a+1, a+1, a[0]+1, &a[0]+1, &a[0][0]+1);

    printf("%d %d %d\n", **a, *a[0] , a[0][0]);

    return 0;
}
```

```
(base) tngyeomkim@tngyeomkim-Inspiron-7590:~/embedded/c_program$ ./main
0x7ffd546c4df0 0x7ffd546c4df0 0x7ffd546c4df0 0x7ffd546c4df0 0x7ffd546c4df0
0x7ffd546c4e20 0x7ffd546c4e00 0x7ffd546c4df4 0x7ffd546c4e00 0x7ffd546c4df4
0 0 0
(base) jngyeomkim@jngyeomkim-Inspiron-7590:~/embedded/c_program$ vi assay.po
```

&a에 1을 더하면 주소값이 48byte 증가한다.
a에 1을 더하면 주소값이 16byte 증가한다.
a[0]에 1을 더하면 주소값이 4byte 증가한다.
&a[0]에 1을 더하면 주소값이 12byte 증가한다.
&a[0][0]에 1을 더하면 주소값이 4byte 증가한다.

포인터 추가 내용 정리

1. 포인터의 크기 : 포인터는 주소를 저장하는 변수이므로 포인터의 크기는 x86에서 4byte, x64에서 8byte이다. 어떤 자료형의 포인터든 동일하다.

```c
#include <stdio.h>

int main(void)
{
    char a = 'a';
    float f = 3.14f;
    double d = 123.456;

    char* a_ptr = &a;
    float* f_ptr = &f;
    double* d_ptr = &d;

    printf("%zd %zd %zd\n", sizeof(a_ptr), sizeof(f_ptr), sizeof(d_ptr));
    printf("%zd %zd %zd\n", sizeof(&a), sizeof(&f), sizeof(&d));
    printf("%zd %zd %zd\n", sizeof(char*), sizeof(float*), sizeof(double*));

    return 0;
}
```

```
8 8 8
8 8 8
8 8 8
```

포인터 추가 내용 정리

2. 포인터의 산술연산 : 포인터에 +1은 자료형의 크기만큼의 증가를 의미한다
포인터끼리의 덧셈은 의미가 없지만 포인터끼리의 뺄셈은 주솟값의 차이라는 의미를 가진다

```c
#include <stdio.h>

int main(void)
{
    int* ptr = 0;
    printf("%p\n", ptr);
    printf("%p\n", ptr+1);
    printf("%p\n", ptr+2);


    return 0;
}
```

```
(nil)
0x4
0x8
```

```c
#include <stdio.h>

int main(void)
{
    int arr[5] = {1, 2, 3, 4, 5};
    int* ptr1 = &arr[1];
    int* ptr2 = &arr[3];

    printf("%ld\n", ptr2- ptr1);

    return 0;
}
```

```
2
```

포인터 추가 내용 정리

3. 포인터와 배열의 차이점 : 배열의 이름은 배열이 시작하는 첫 번째 주소라는 특별한 의미를 가진다.
배열의 이름을 포인터처럼 사용할 수는 있지만
배열의 이름은 포인터처럼 메모리 공간을 가지진 않는다.

포인터ptr += 2; (가능)
배열 arr += 2; (불가능)

포인터 추가 내용 정리

4. 배열을 함수에게 전달해주는 방법 :
ave함수에 배열을 인수로 건내주면
함수에서는 배열의 주소를 포인터 값으로
입력받는다.

매개변수인 포인터가 배열처럼
ptr[i]이렇게 쓰일 수도 있구나.(23줄)

```c
#include <stdio.h>

double ave(double* , int n);

int main(void)
{
    double arr1[5] = {10, 13, 12, 7, 8};
    double arr2[3] = {1.8, -0.2, 6.3};

    printf("average1 = : %f\n", ave(arr1,5));
    printf("average1 = : %f\n", ave(arr2,3));

    return 0;
}

double ave(double* ptr, int n)
{
    float ave;
    float sum = 0;
    int i;

    for(i = 0; i < n ; i++)
        sum += ptr[i];
    ave = sum / n;

    return ave;
}
```

포인터 추가 내용 정리

* 포인터에 배열을 집어넣으면
  포인터에 인덱스를 참조해서 배열처럼 쓸 수 있다.

```c
#include <stdio.h>

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

    int* a_ptr = a;

    printf("%d %d %d\n", a[0], a[1], a[2]);
    printf("%d %d %d\n", a_ptr[0], a_ptr[1], a_ptr[2]);

    a_ptr[0] = 99;
    printf("%d %d %d\n", a[0], a[1], a[2]);
    printf("%d %d %d\n", a_ptr[0], a_ptr[1], a_ptr[2]);

    return 0;
}
```

```
0 1 2
0 1 2
99 1 2
99 1 2
```

10. Structure

1. 구조체란 : 서로 다른 자료형을 하나로 묶어서 관리할 수 있게 해준다
   메로리에 순차적으로 데이터가 모두 쌓인다.

선언하는 방법ex)1

```
struct Books
{
	char title[50];
	char author[50];
};
```

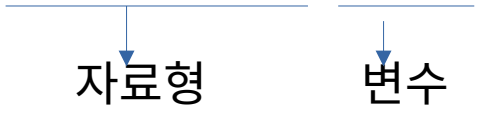
선언하는 방법ex)2

```
typedef struct Books
{
	char title[50];
	char author[50];
}books;
```

10. Structure

2. 사용법

- 자료형 선언 : Struct Books book1;



- 문자열 입력 방법 : strcpy(자료형 , “문자열”);
  (<string.h>라이브러리를 포함해야됨)
  ex) strcpy(book1.tilte, “C Programming);

- 정수를 입력하는 방법 : 자료형 = 정수;
  ex) book1.b00k_id = 123456;

- 구조체 변수를 매개변수로 받는 함수를 만들 수 도 있다.
  ex) void printBook (struct Books book);

- 구조체도 변수이기 때문에 포인터를 사용할 수 있다.
  구조체의 주소를 매개변수로 입력받는 함수를 사용할 경우
  ‘->’를 사용해서 값에 접근할 수 있다.

```
print2(&book1);
```

```
void print2(struct Books* book)
{
    printf("%s\n", book->title);
    printf("%s\n", book->author);
    printf("%d\n", book->book_id);
}
```

11. Union

1. 공용체란 : 구조체와 비슷하게 사용된다.
   구조체는 메모리에 모든 데이터가 쌓이지만
   공용체는 데이터타입이 가장 큰 값 만큼의 메모리만 할당된다.

```
 4 union data{
 5     int i;
 6     float f;
 7     char str[20];
 8 };
 9
10 int main(void)
11 {
12     union data data1;
13
14     printf("%zd\n", sizeof(data1));
15
16     return 0;
17 }
```

data1의 크기는 20byte이다.
왜냐하면 메모리를 가장 많이 차지하는변수 str[20]의 크기만큼 공용체의 크기가 할당된다

11. Union

다음예제를 통해
공용체 data1에는
data1.i와 data1.f에는 쓰레기값이 들어있고
data1.str만 정상적으로 출력됨을 알 수 있다.

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

union data{
    int i;
    float f;
    char str[20];
};

int main(void)
{
    union data data1;

    data1.i = 1;
    data1.f = 3.14;
    strcpy(data1.str, "C programming");

    printf("%d\n", data1.i);
    printf("%f\n", data1.f);
    printf("%s\n", data1.str);

    return 0;
}
```

```
1919950915
475618588044190956120111597056.000000
C programming
```

11. Union

공용체에 값을 입력하는 순서를 바꿔주면
어떻게 될까?

이전 예제와는 다르게
data1.str와 data1.i에 쓰레기값이 들어가있고
data1.f만 정상적으로 값이 출력됨을 알 수
있다

공용체는 마지막에 입력한 데이터만 메모리에
저장됨을 알 수 있다.

지금 예제에서 공용체의 크기
(sizeof(data1))은 여전히 20byte이다.

데이터를 쓰고 지울 수 있으므로 용량 관리에
유리하다는 것을 알 수 있다.

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

union data{
    int i;
    float f;
    char str[20];
};

int main(void)
{
    union data data1;

    data1.i = 1;
    strcpy(data1.str, "C programming");
    data1.f = 3.14;

    printf("%d\n", data1.i);
    printf("%f\n", data1.f);
    printf("%s\n", data1.str);

    return 0;
}
```

```
1078523331
3.140000
H@ogramming
```

11. bit feilds

1. bit fields 란? : 구조체와 비슷하게 사용된다
bit fields를 사용하면 구조체에서 메모리의 크기를 지정할 수 있다.

2. 비트필드의 패딩(padding)
구조체 status1의 크기는 unsigned int가 2개
이므로 8byte이지만,
비트필드를 사용한 구조체 status2는
가장 큰 데이터타입 1개 만큼의 크기가 할당되므로
크기가 4byte이다(unsigned int 1개만큼의 크기)
공용체의 특징을 가짐을 알 수 있다.

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

struct{
    unsigned int width;
    unsigned int heigh;

} status1;

struct{
    unsigned int width : 1;
    unsigned int heigh : 1;
} status2;

int main(void)
{
    printf("%zd\n", sizeof(status1));
    printf("%zd\n", sizeof(status2));

    return 0;
}

```

```
8
4
```

11. bit feilds

*다음구조체의 크기는 8byte(unsigned long long의 크기)이다.

```
struct{
    unsigned long long width : 1;
    unsigned int heigh : 1;
} status2;
```

*비트필드에서 지정된 비트를 초과하는 값을 할당하면 오버플로우가 발생한다.

HW. USBCREGFlag는 어떻게 구성되어 있는지 분석하기

1. flag란? : 특정 동작을 수행할지 말지 결정하는 변수를 플래그라고 부른다. 0과 1의 1bit로 표현된다,.

2. flag register란? : 레지스터의 종류 중 하나로서 연산결과의 상태를 나타내는 bit flag들이 모인 레지스터이다.

3. flag register 종류: CF(캐리 플래그), PF(패리티플 래그), AF(보조 플래그), ZF(제로 플래그), SF(사인 플래그), OF(오버플로우 플래그) 등이 있다.