시스템 프로그래밍을 위한 C언어

- Traversing Array via Pointer -

현대자동차 입문교육
박대진 교수

Multiple Memory Allocation by Array

You have to define multiple variables one by one, but array can do this easy way. Memory allocation size is same.

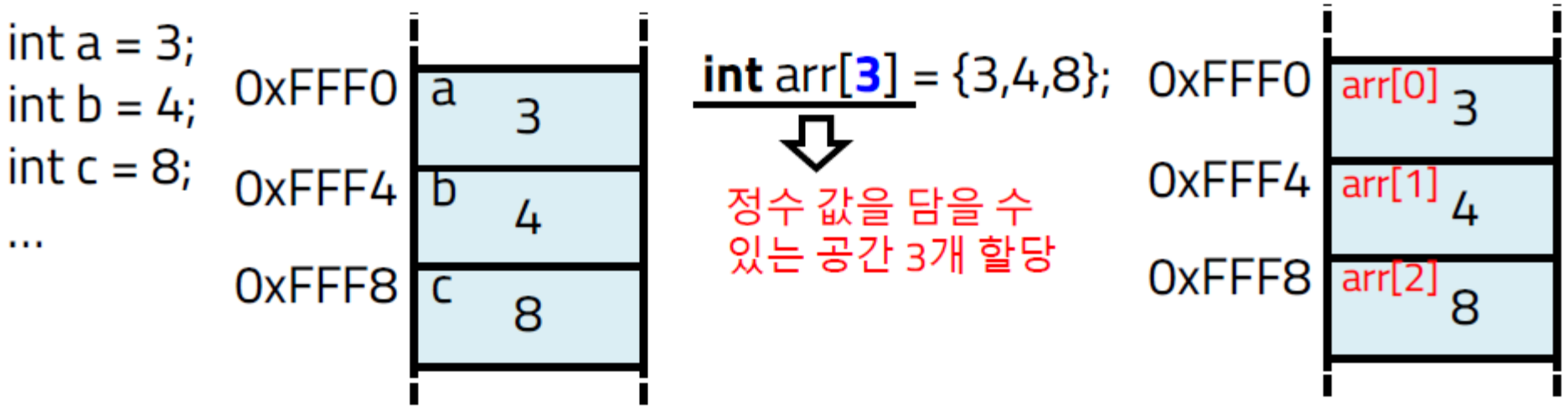


Fig: Comparison of single variable and array

Elements, and Their Values

All elements in array can be accessed via arr[i] using index number i, starting from 0 to N-1.

정수를 3개 담을 공간할당, 그 값을 3,5,8로 초기화

int arr[**3**] = {3,5,8}; ⇨

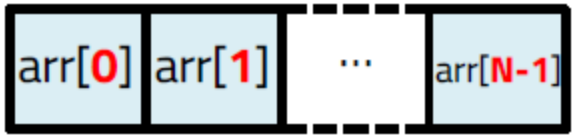
3	5	8
arr[**0**]	arr[**1**]	arr[**2**]

arr[0]=7;
arr[1]=4;
arr[2]=1;

⇨

7	**4**	**1**
arr[**0**]	arr[**1**]	arr[**2**]

int arr[**N**];



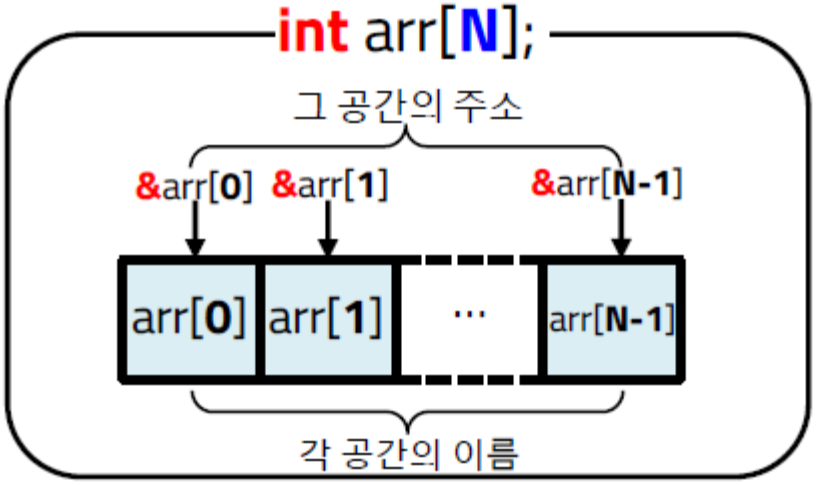
N개의 방이 있고
각 방에는 **int**값이 들어갈 수 있음
방의 index는 **0**부터~**N-1**번까지

Fig: Array elements with specific index

Accessing Elements in Array

Each element in array is also a space with specific address, which is allocated in memory.
To get an address for this element, just append & to element, like &arr[i].

모든 공간(변수)에는 주소가 있다. 공간이름 앞에 &




```
int x[3] = {3,5,8};
printf("x[0] is %d at %X\n", x[0], &x[0]);
printf("x[1] is %d at %X\n", x[1], &x[1]);
printf("x[2] is %d at %X\n", x[2], &x[2]);
```

⇨

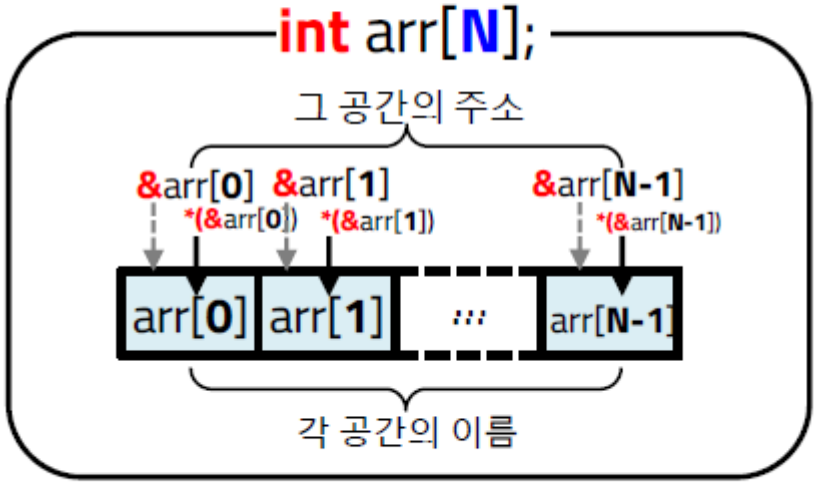
```
x[0] is 3 at FF00
x[1] is 5 at FF04
x[2] is 8 at FF08
```

Fig: An element in array, and its address

Dereferencing by Address

You can access elements anywhere in array, if you have specific address, which points out an element in array. Just append * to address variable, like *(&arr[i]).

주소값 앞에 *를 붙이면 그 주소에 있는 값을 얻음




```
int x[3] = {3,5,8};
printf("x[0] is %d at %X\n", *(&x[0]), &x[0]);
printf("x[1] is %d at %X\n", *(&x[1]), &x[1]);
printf("x[2] is %d at %X\n", *(&x[2]), &x[2]);
```

⇨

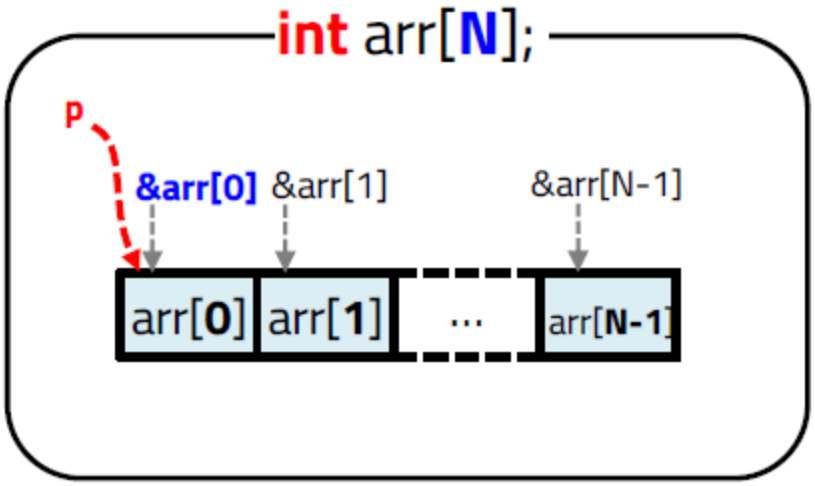
```
x[0] is 3 at FF00
x[1] is 5 at FF04
x[2] is 8 at FF08
```

Fig: Access elements using their address

Pointer-based Address Calculation

A pointer variable can be used to access the elements in array. First copy the address to the pointer variable, then you can access each element in array by calculating address for specific element in i index location.

주소를 저장하는 포인터 변수로 행렬에 접근해보자




```
int x[3] = {3,5,8};
Int* p; // 주소값을 담을 수 있는 공간
p=&x[0]; // 행렬의 첫번째 공간의 주소
printf("x[0] is %d at %X\n", * p    , p    );
printf("x[1] is %d at %X\n", *(p+1), p+1);
printf("x[2] is %d at %X\n", *(p+2), p+2);
```

⇨

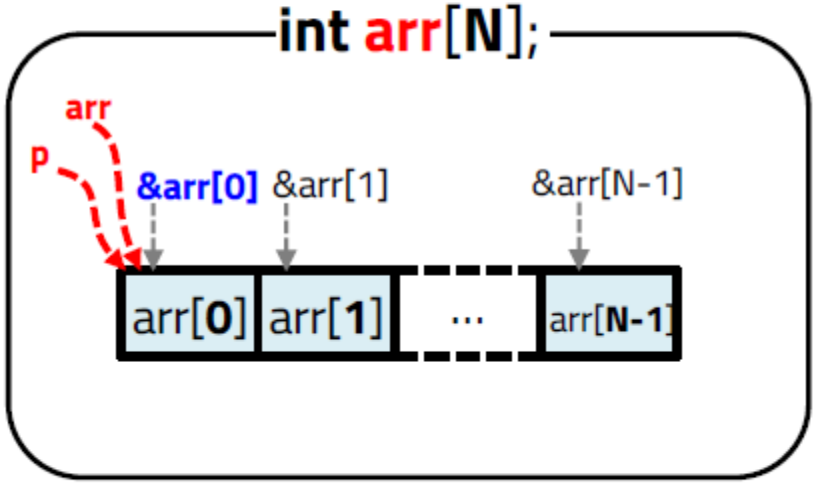
```
x[0] is 3 at FF00
x[1] is 5 at FF04
x[2] is 8 at FF08
```

Fig: Pointer-based array access

Array Name is First Address

Array name is itself an address to point out the start of array. So you can directly access the element using address calculation for the distance from the start address, which is given by the name of address.

행렬 이름 그 자체는 행렬의 첫번째 방의 주소임




```
int x[3] = {3,5,8};
Int* p; // 주소값을 담을 수 있는 공간
p=x; // 행렬이름은 첫번째 공간의 주소임
printf("x[0] is %d at %X\n", * p    , p    );
printf("x[1] is %d at %X\n", *(p+1), p+1);
printf("x[2] is %d at %X\n", *(p+2), p+2);
```

⇨

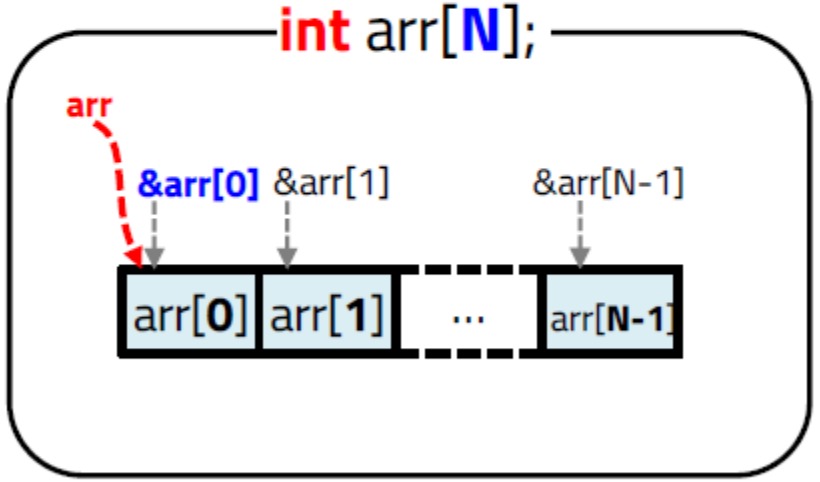
```
x[0] is 3 at FF00
x[1] is 5 at FF04
x[2] is 8 at FF08
```

Fig: Array name-based access to elements in array

Array Name-based Address Calculation

Array name is itself an address to point out the start of array. So you can directly access the element using address calculation for the distance from the start address, which is given by the name of address.

행렬 이름 그 자체는 행렬의 첫번째 방의 주소임




```
int x[3] = {3,5,8};

printf("x[0] is %d at %X\n", * x     , x    );
printf("x[1] is %d at %X\n", *(x+1), x+1);
printf("x[2] is %d at %X\n", *(x+2), x+2);
```

⇨

```
x[0] is 3 at FF00
x[1] is 5 at FF04
x[2] is 8 at FF08
```

Fig: Array name-based access to elements in array

Two-Dimensional Array

Two dimensional array is same everything to one dimensional array, except the size and access index with i, j. The two dimensional array arr[M][N] has total MxN number of elements. Each element is m[i][j] for index i, j.

M(row) x N(column) 2차원 행렬 → M*N개의 공간

int m[**M**][**N**]

가로 N개, 세로 M개의 행렬, 공간은 M*N개 각 방에는 int값 저장

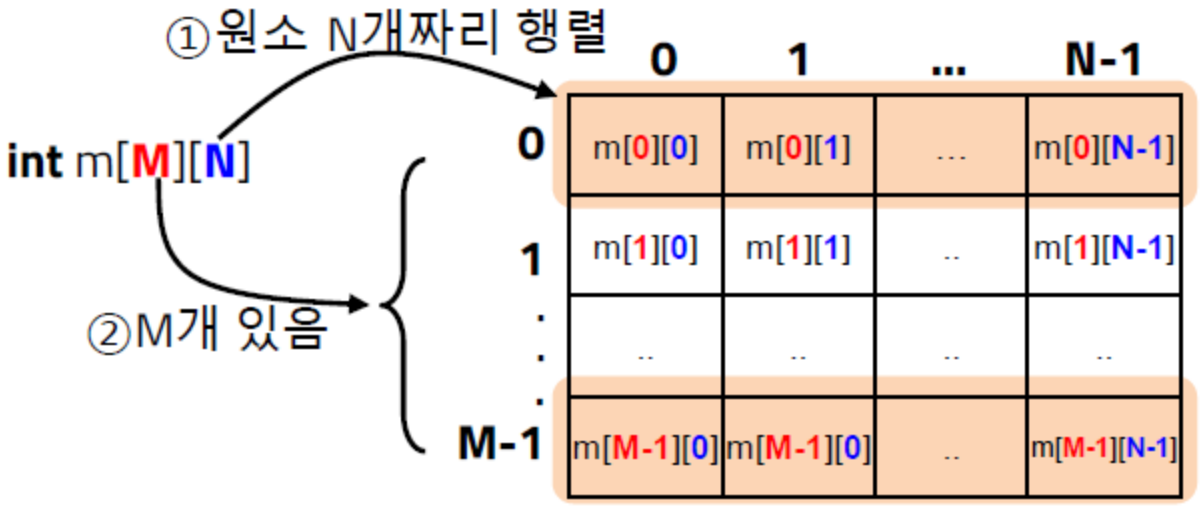
	0	**1**	**...**	**N-1**
0	m[0][0]	m[0][1]	...	m[0][N-1]
1	m[1][0]	m[1][1]	..	m[1][N-1]
:	..	..	..	..
M-1	m[M-1][0]	m[M-1][0]	..	m[M-1][N-1]

```
int m[3][4] =
{
  {1,2,3,4},
  {5,6,7,8},
  {3,5,1,9}
};
```

Accessing Elements in 2D Array

MxN array is considered as M number of N-size array in horizontal way.

M(row) x N(column) 2차원 행렬 → N개 크기의 행렬이 M개 있다




```
int m[3][4] =
{
 {1,2,3,4},
 {5,6,7,8},
 {3,5,1,9}
};
```

Fig: Array of one dimensional array

Pointer-based Access on 2D Array

All elements can be accessed based on address calculation from the start address of array.

행렬도 값과, 주소에 할당된 연속된 공간일 뿐. 주소로 모두 접근가능

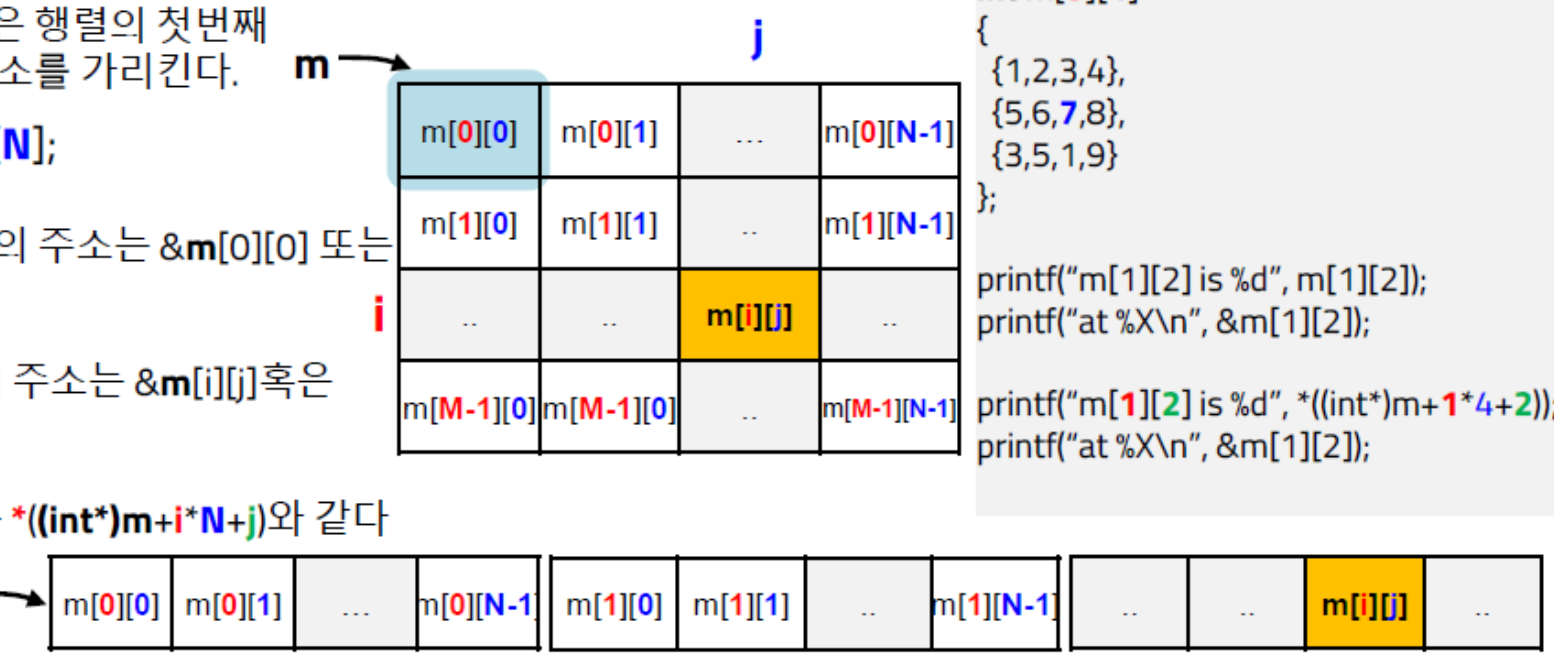
행렬이름은 행렬의 첫번째 원소의 주소를 가리킨다.

int m[**M**][**N**];

→ **m**[0][0]의 주소는 &**m**[0][0] 또는 **m** 그 차체

→ **m**[**i**][**j**]의 주소는 &**m**[i][j]혹은 **m**+**i***N+**j**

→ m[i][j]는 ***((int*)m+i*N+j)**와 같다




```
int m[3][4] =
{
 {1,2,3,4},
 {5,6,7,8},
 {3,5,1,9}
};

printf("m[1][2] is %d", m[1][2]);
printf("at %X\n", &m[1][2]);

printf("m[1][2] is %d", *((int*)m+1*4+2));
printf("at %X\n", &m[1][2]);
```

Fig: Pointer-based access to elements in two dimensional array

Passing Array into Function

Array has multiple elements, but when assigning it into another array, all elements will not be copied, actually assign operation is not allowed except passing function argument. Just handle it using start address of array. If you duplicate new array, first allocate memory using array, then manually copy all elements one by one.

=를 이용하여 행렬을 다른 행렬에 복사 불가능, 주소만 저장할 수 있음

```
char data1[5] = { 1, 2, 3, 4, 5 };

// not allowed
char data2[5] = data1

// only address copy is allowed
char* data2p = data1; // it's ok
data2p[2] = 8; // but, data1 is modified

// To copy array, manually copy elements
for(int i=0; i<5; i++)
  data2[i] = data1[i];
```

```
void assign(char a[5])
{
 ...
}

void main(void)
{
   char arr[5];
   assign(arr);
}
```

```
void assign(char a[])
{
 ...
}

void main(void)
{
   char arr[5];
   assign(arr);
}
```

```
void assign(char* a)
{
 ...
}

void main(void)
{
   char arr[5];
   assign(arr);
}
```

- 함수의 인자로 받을 때는 위의 3가지 형태로 표현가능, 이때도 a는 arr의 주소만 복사함
- 그러나 이것은 컴파일러마다 지원여부가 달라서 확인이 필요하다.

Fig: Assigning 1-D array into another variable

Accessing Memory Region using Offset-based Displacement

- Hardware is allocated on specific memory region
- So, we can access (read/write) hardware at specific address using offset-based displacement calculation

```
unsigned int arr[5] = {0x12345678,2,3,4,5};
print_array(arr, 5);
for(int i=0; i<5; i++)
    printf("arr[%d] is %d at %p\n", i, arr[i], &arr[i]);
```

```
unsigned char* abp = (unsigned char*)arr;
for(int i=0; i<4; i++)
    printf("mem[%d] is %2X at %p\n", i, *(abp+i), abp+i);

*(abp+2) = 0x5A;
```

```
for(int i=0; i<N; i++)
    printf("mem[%d] is %2X at %p\n", i, *(abp+i), abp+i);
```

Linear Memory Conversion-based Access to 2D Array

```c
unsigned int mat[3][4] = {
    {1,2,3,4},
    {5,6,7,8},
    {3,5,1,9}
};
unsigned int* mp = (unsigned int*)mat;

for(int i=0; i<3; i++) {
    for(int j=0; j<4; j++) {
        //printf("%2X ", mat[i][j]);
        printf("%2X ", *(mp+i*4+j));
    }
    printf("\n");
}
```