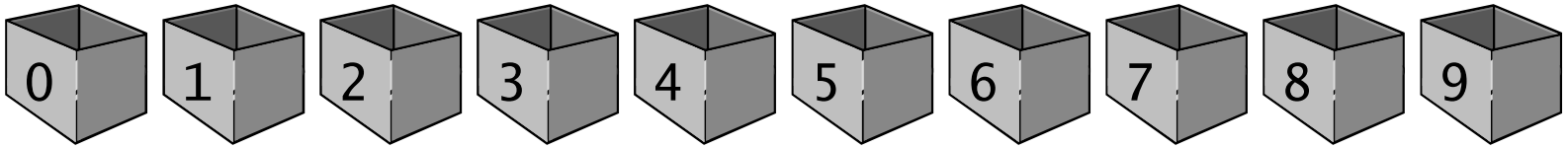
Chapter 2::
Array, Structure, and Pointer (1)

SANGJI University
Kwangman Ko
(kkman@sangji.ac.kr)

배열(array) 이란 ?

- 같은 형의 변수를 여러 개 만드는 경우에 사용
  - int A0, A1, A2, A3, …,A9;
  - int A[10];

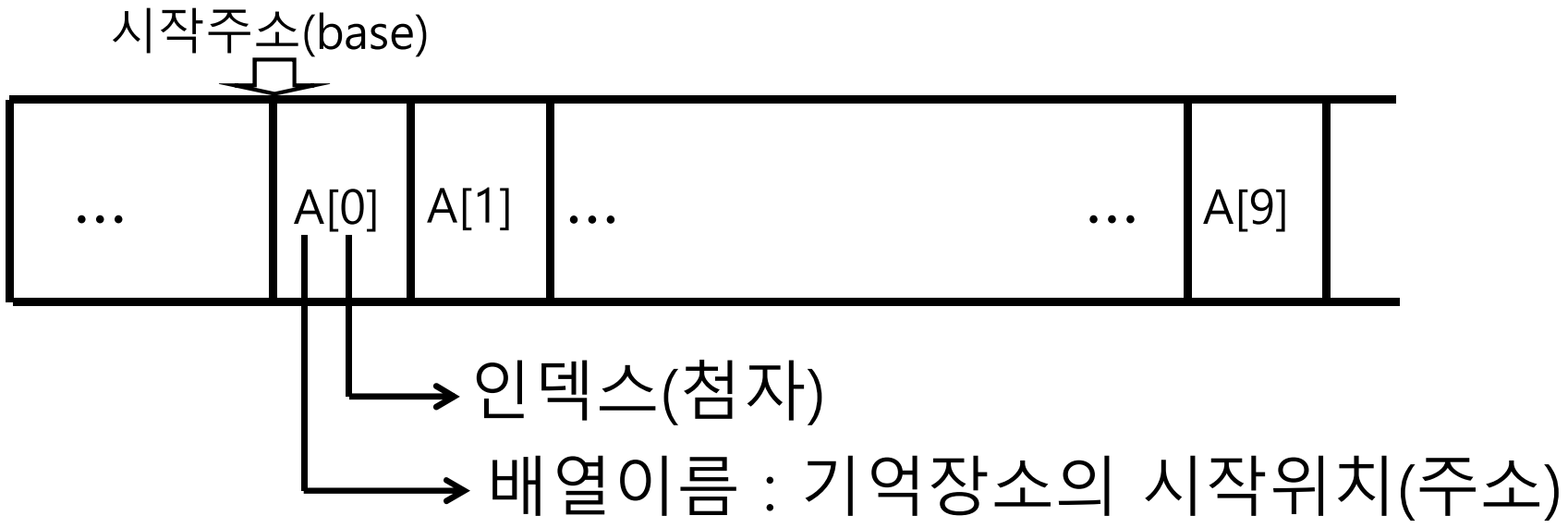


- 반복 코드 등에서 배열을 사용하면 효율적인 프로그래밍이 가능
  - 예) 최대값을 구하는 프로그램: 만약 배열이 없었다면?

```
tmp=score[0];
for(i=1;i<n;i++){
        if( score[i] > tmp )
                tmp = score[i];
}
```

- 배열(array)
  - 동일 자료형(same type)의 자료들이 순서(linear) 있게 구성된 집합
  - 연속된 기억 공간 차지
  - 유한한 개수의 자료가 저장됨
  - 직접접근 (direct access)
    - 기준위치에 대한 상대적 위치를 나타내는 인덱스(index)를 사용하여 가능

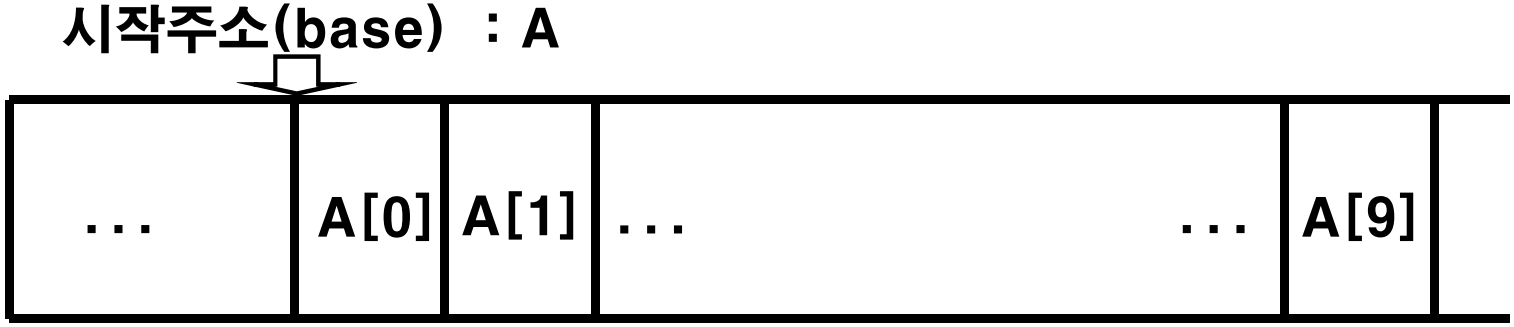
- int A[10];
  - 배열을 구성하는 원소의 개수(size) : 10
    - 인덱스 : 0, 1, … , 9
  - 배열을 구성하는 각 원소의 자료형 : int
  - 배열이 차지는 연속된 기억공간의 크기
    - 4 byte * 10 = 40 byte



배열의 선언과 생성

- 배열의 선언(declaration)과 생성
  - 자료형
  - 배열 이름
  - 배열의 크기

- 배열 예(in C)
  - 자료형 배열이름[배열크기] ;
  - int A[10] ;



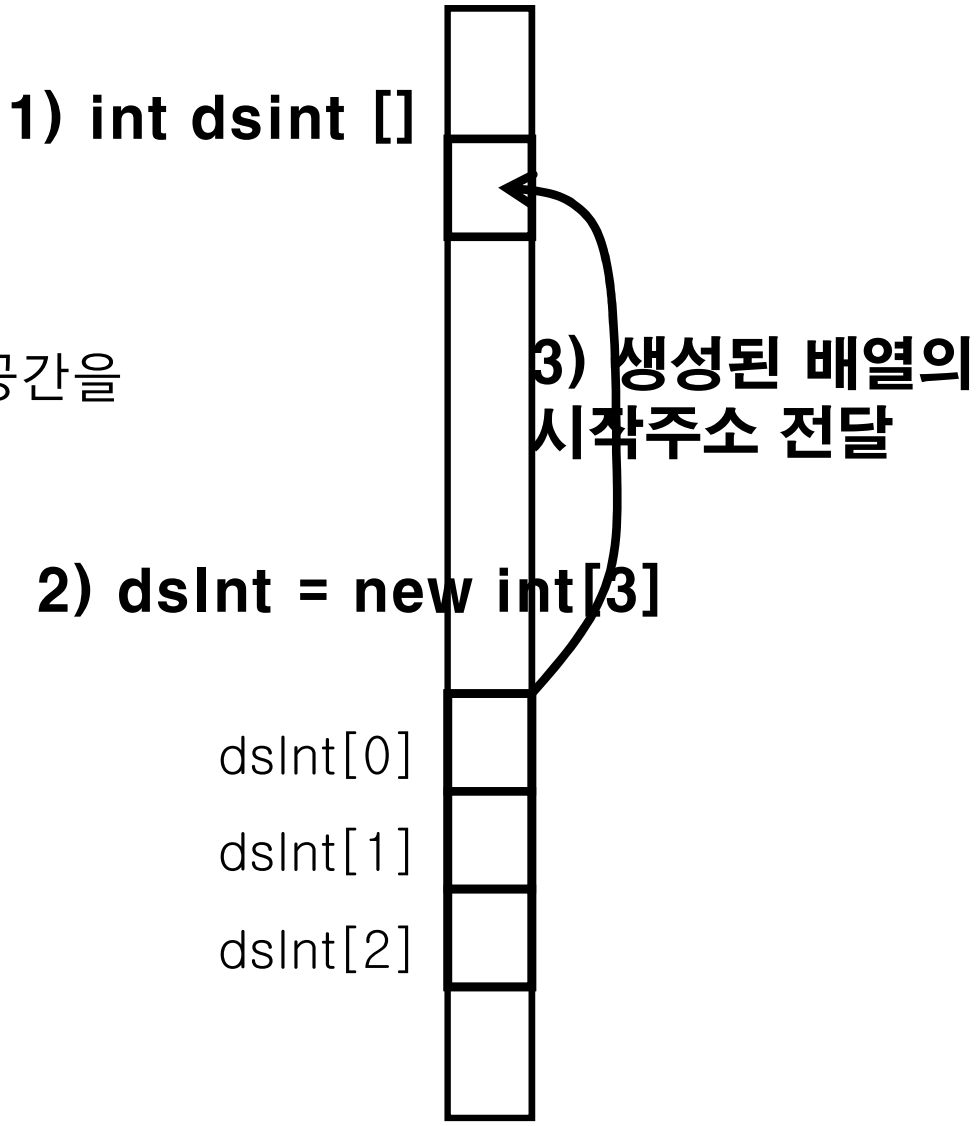
Java에서 배열의 선언과 생성

- 선언(declaration)
  - 생성될 배열 시작 위치 저장(배열 이름)

```
int dsInt [] ;
```

- 생성(creation)
  - 힙 메모리에서 배열 크기만큼의 기억 공간을 할당한 후 시작 주소를 배정

```
dsInt = new int[3] ;
```



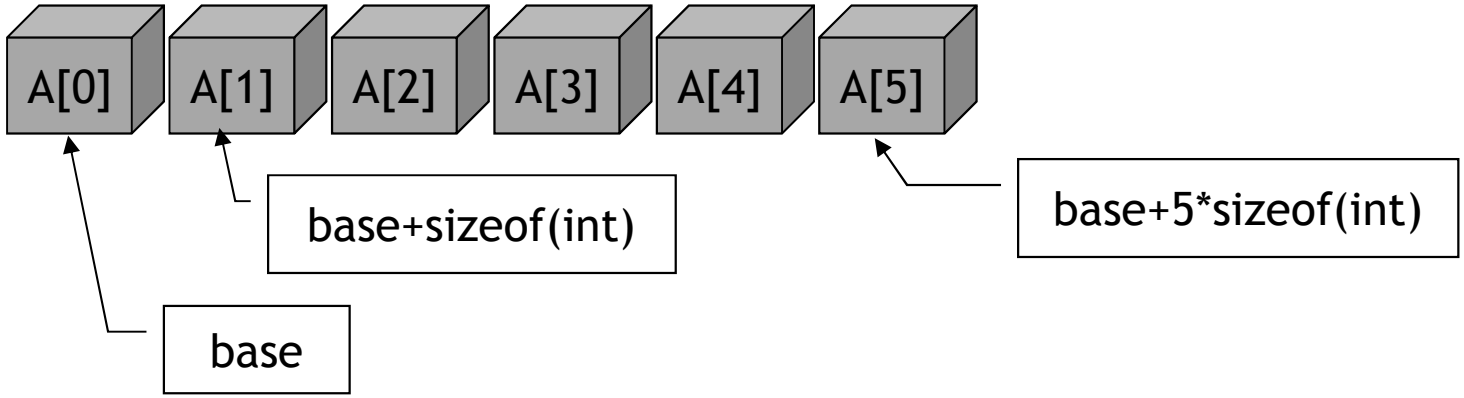
- 배열의 생성과 초기화
  - int array[3];
  - int array[3] = { 1, 2, 3 } ;
  - int array[] = { 1, 2, 3 } ;
    - 배열의 크기가 초기값의 수에 따라 결정
    - 위 배열의 크기 ?

Q) Java 언어를 사용하여 동일하게 선언과 생성하면 ?

배열의 종류

- 1차원 배열
  - 가장 간단한 형태로서 길이가 n으로 초기화된 배열
    - 0에서 n-1까지의 전체 n개의 자료 저장 가능
  - 일차원 배열의 선언과 생성
    - int array[10] ; // in C/C++
    - int array = new int[10] ; // in Java
  - 순차 매핑 (sequential mapping)
    - 배열의 논리적 순서와 메모리의 물리적 순서를 같게 표현
  - 순차 표현 (sequential representation)
    - 순차 매핑을 이용하여 자료 표현

- 예) int A[6] ;



- int dsInt = new int[10];

배열 ADT

- 배열: <인덱스, 요소> 쌍의 집합
- 인덱스가 주어지면 해당되는 요소가 대응되는 구조

배열 ADT

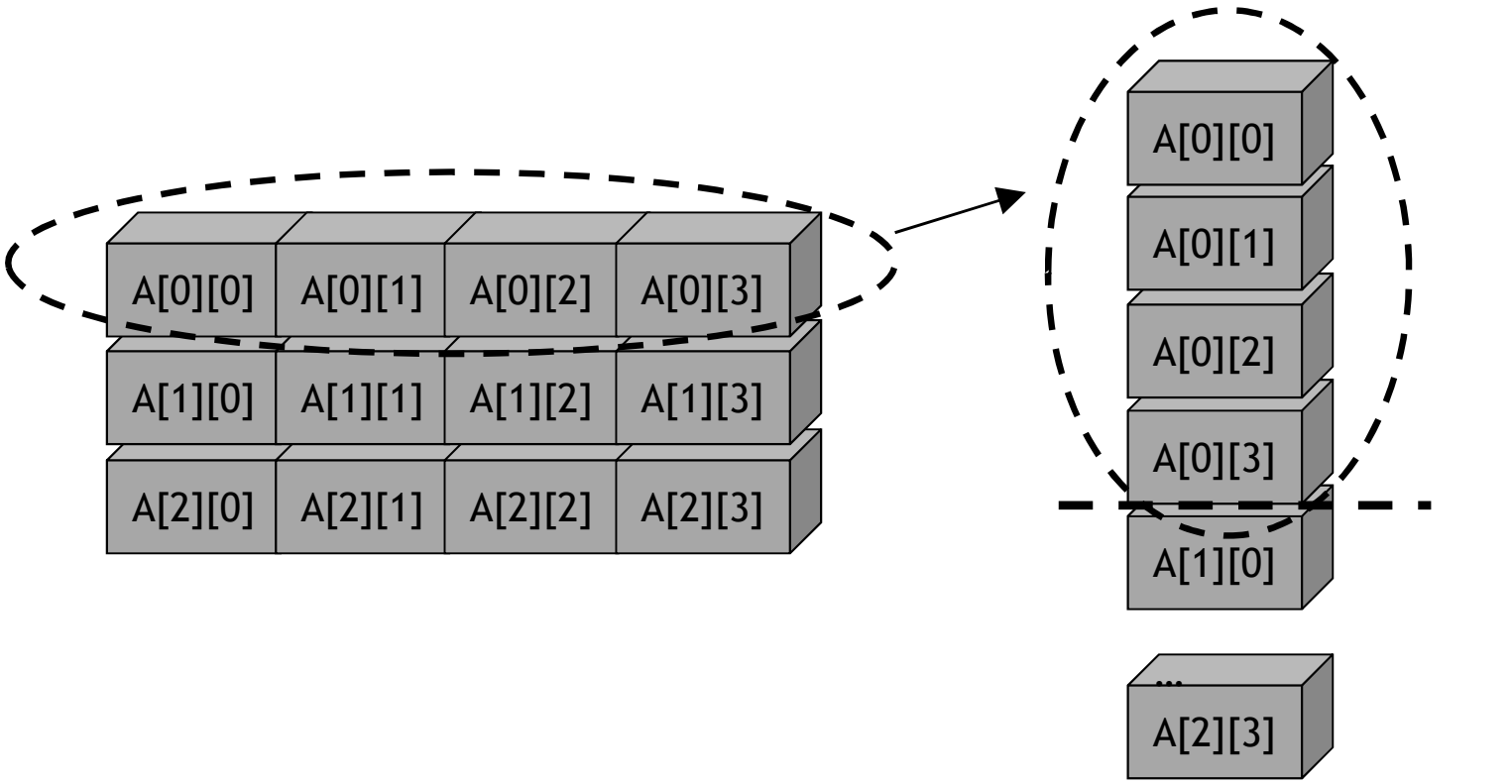
객체: <인덱스, 요소> 쌍의 집합

연산:

- create(n) ::= n개의 요소를 가진 배열의 생성.
- retrieve(A, i) ::= 배열 A의 i번째 요소 반환.
- store(A, i, item) ::= 배열 A의 i번째 위치에 item 저장.

2차원 배열

- int A[3][4];



실제 메모리 안에서의 위치

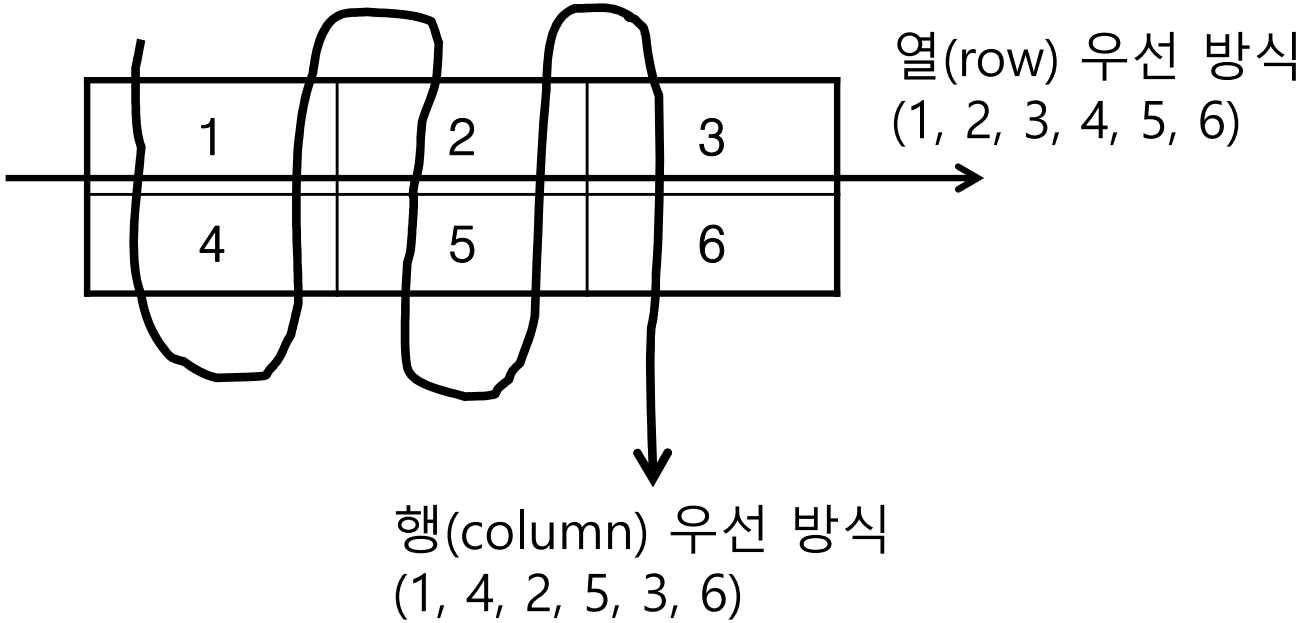
- 다차원 배열(Multi dimensional array)
  - 여러 개의 일차원 배열들의 중첩된 구조
  - 다차원 배열을 실제 일차원 메모리에 표현하는 방법 ?
    - 행우선 방법, 열우선 방법

다차원 배열 => 일차원 배열 매핑

- 행 우선 순서 방법(row major order)
  - 이차원 배열을 "행" 별로 분할
  - 분할된 행들을 차례로 연결
  - 하나의 가상적인 일차원 배열로 매핑

- 열 우선 순서 방법(column major order)
  - 이차원 배열을 "열" 별로 분할
  - 분할된 열들을 차례로 연결
  - 하나의 가상적인 일차원 배열로 만들어 매핑

- 열/행 우선 방법(row/column-major order) in C
  - int array [2][3] = { {1,2,3}, {4,5,6} };



- 열/행 우선 방법(row/column-major order) in Java
  - int array ; // 선언
  - array = new int[2][3] ; // 생성
  - array = new int[2][3] = { {1,2},{3,4},{5,6} } ; //초기화

- 기억공간 표현 ?

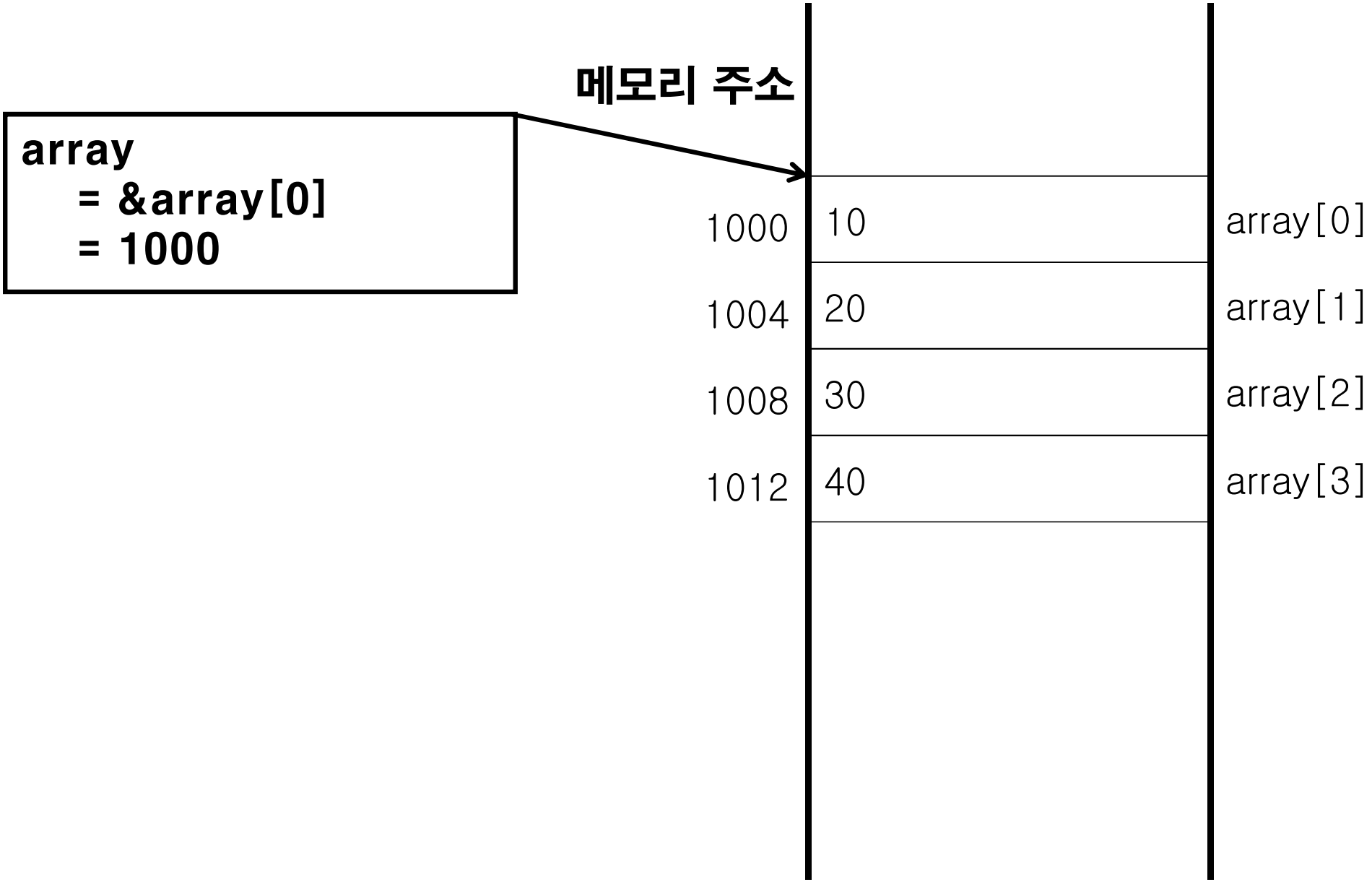
배열 원소의 접근

- 인덱스를 통한 직접 접근
  - 배열은 메모리상에 연속된 기억공간 차지
  - 인덱스의 계산을 통해 해당 객체에 직접 접근
  - 크기 n의 배열, 인덱스 범위 0 ... n-1
  - 배열 이름은 메모리상에서 배열의 시작 위치.

- &array[i] = &array[0] + (i * sizeof(type))
  - &array[0] = &array[0] + (0 * sizeof(int)) = 1000
  - &array[1] = &array[0] + (1 * sizeof(int)) = 1004
  - &array[2] = &array[0] + (2 * sizeof(int)) = 1008
  - &array[3] = &array[0] + (3 * sizeof(int)) = 1012

    - 참고, &array[0] = 1000, sizeof(int) = 4

int array[4] = {10, 20, 30, 40}; //in C



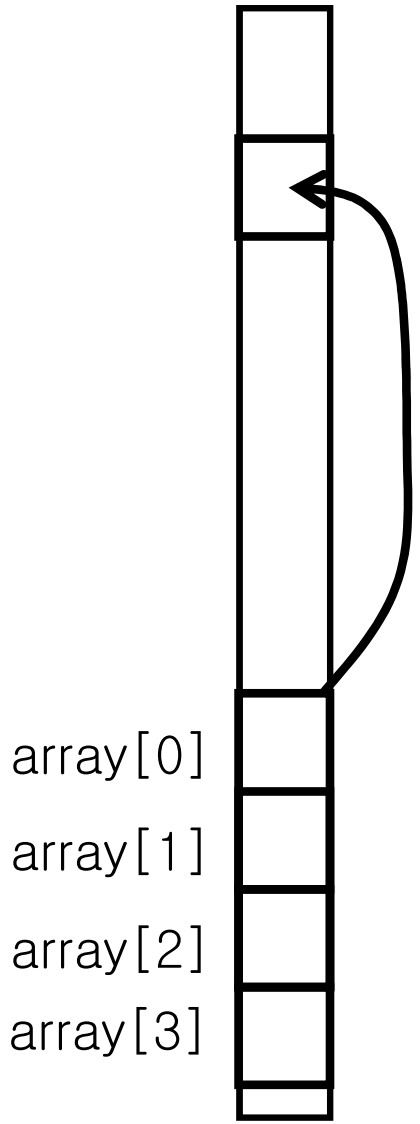
int array = new int[4];

1) int array[]

2) array = new int[4]

3) 생성된 배열의 시작주소 전달

4) 초기화



배열의 초기화

- 일차원 배열 초기화(in Java)
  - 배열을 선언하고 생성한 다음에 초기화를 하는 방법
    - int [ ] dsInt; // 배열 선언
    - dsInt = new int [10]; // 배열 생성
    - dsInt [0] = 12; // 배열의 초기화
    - dsInt [1] = 53; …
  - 배열을 동시에 선언하고 생성한 후에 초기화를 하는 방법
    - int dsInt [ ] = new int [10]; // 배열 선언 및 생성
    - dsInt [0] = 12; // 배열의 초기화
    - dsInt [1] = 53;

- 일차원 배열 초기화(in C) ???

기억 장소 할당 방법

- 프로그램이 메모리를 할당받는 방법
  - 정적(static) 메모리 할당 => 스택(stack)
  - 동적(dynamic) 메모리 할당 => 힙(heap)

- 정적 메모리 할당(static memory allocation)
  - 메모리의 크기는 프로그램이 시작하기 전에 결정
    - 컴파일 시간에 기억 장소의 크기 결정, 실행시 불변.
  - 프로그램의 수행 도중에 그 크기가 변경될 수는 없음.
    - 처음에 결정된 크기보다 더 큰 입력이 들어온다면 처리하지 못할 것이고 더 작은 입력이 들어온다면 남은 메모리 공간은 낭비.
  - 예, 변수나 배열의 선언
    - int buffer[100];
    - char name[] = “data structure";

- 동적 메모리 할당(dynamic memory allocation)
  - 프로그램의 실행 도중에 메모리를 할당받는 것
  - 필요한 만큼만 할당을 받고 또 필요한 때에 사용하고 반납
  - 메모리를 매우 효율적(=탄력적)으로 사용가능

```
main()
{
    int *pi;
    pi = (int *) malloc( sizeof(int) );       // 동적 메모리 할당
    ...
    ...                                        // 동적 메모리 사용
    ...
    free(pi);                                  // 동적 메모리 반납
}
```

- 동적 메모리 할당 관련 라이브러리 함수(in C)
  - malloc(size) // 메모리 할당
  - free(ptr) // 메모리 할당 해제
  - sizeof(var) // 변수나 타입의 크기 반환(바이트 단위)

- 동적 메모리 할당 방법(in C++) ?

- 동적 메모리 할당 방법(in Java) ?

배열 연산

- 검색(search)
- 삽입(insert)
- 삭제(delete)

배열 응용 : 순차 배열

- 순차 배열
  - 배열 내에 저장된 값들이 어떤 일정한 규칙에 의해 정렬
    - 오름차순, 내림차순, …
  - 빠른 검색 제공
  - 첨가 시에 매우 많은 시간 필요
    - 배열에 저장되어 있는 값들을 서로 비교
    - 정해진 자리에 값을 삽입
    - 자리 이동

배열의 응용: 다항식

- 다항식의 일반적인 형태

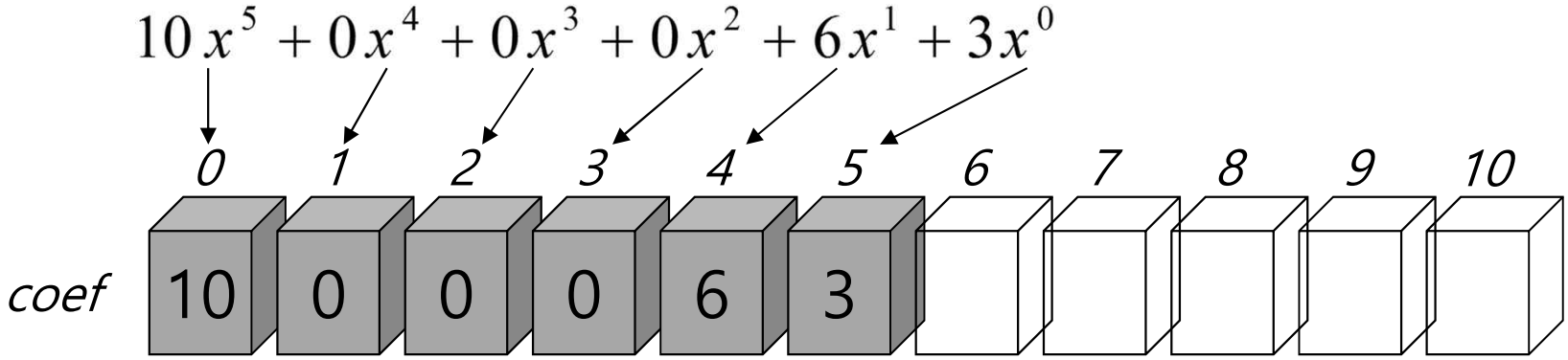
$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

- 프로그램에서 다항식을 처리하려면 다항식을 위한 자료구조가 필요
  - 어떤 자료구조를 사용해야 다항식의 덧셈, 뺄셈,곱셈, 나눗셈 연산을 할때 편리하고 효율적일까?

- 배열을 사용한 2가지 방법
  1) 다항식의 모든 항을 배열에 저장
  2) 다항식의 0이 아닌 항만을 배열에 저장

다항식 표현 방법 #1

- 모든 차수에 대한 계수값을 배열로 저장
- 하나의 다항식을 하나의 배열로 표현




```
typedef struct {
        int degree;
        float coef[MAX_DEGREE];
} polynomial;
polynomial a = { 5, {10, 0, 0, 0, 6, 3} };
```

다항식 표현 방법 #1(계속)

- 장점: 다항식의 각종 연산이 간단해짐
- 단점: 대부분의 항의 계수가 0이면 공간의 낭비가 심함.
- 예) 다항식의 덧셈 연산

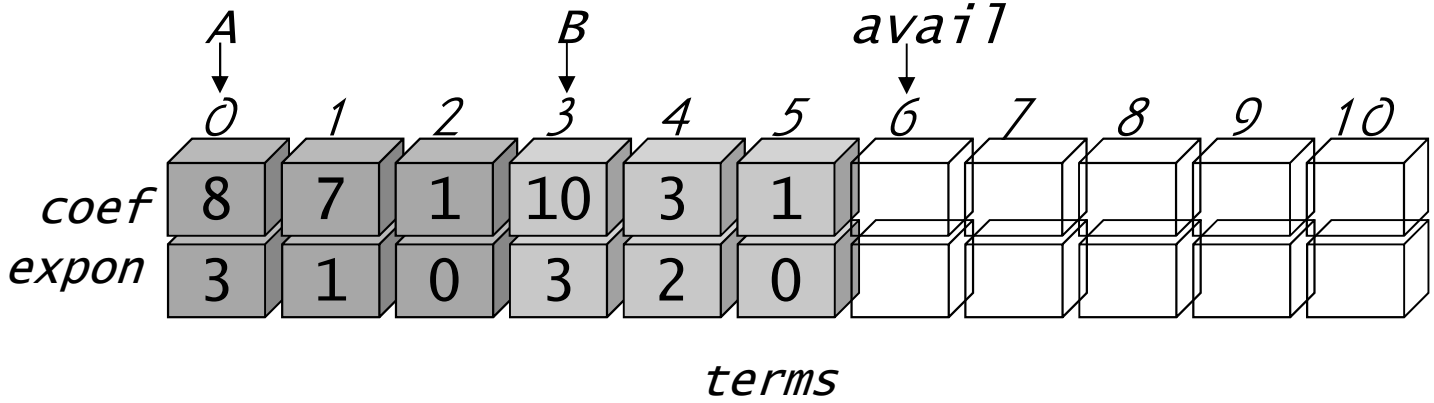
```
while( Apos<=A.degree && Bpos<=B.degree ){
        if( degree_a > degree_b ){  // A항 > B항
          C.coef[Cpos++]= A.coef[Apos++];
          degree_a--;
        }
        else if( degree_a == degree_b ){  // A항 == B항
          C.coef[Cpos++]=A.coef[Apos++]+B.coef[Bpos++];
          degree_a--; degree_b--;
        }
        else {                            // B항 > A항
          C.coef[Cpos++]= B.coef[Bpos++];
          degree_b--;
        }
}
```

다항식 표현 방법 #2

- 다항식에서 0이 아닌 항만을 배열에 저장
- (계수, 차수) 형식으로 배열에 저장
  - (예) $10x^5+6x+3$ -> ((10,5), (6,1), (3,0))

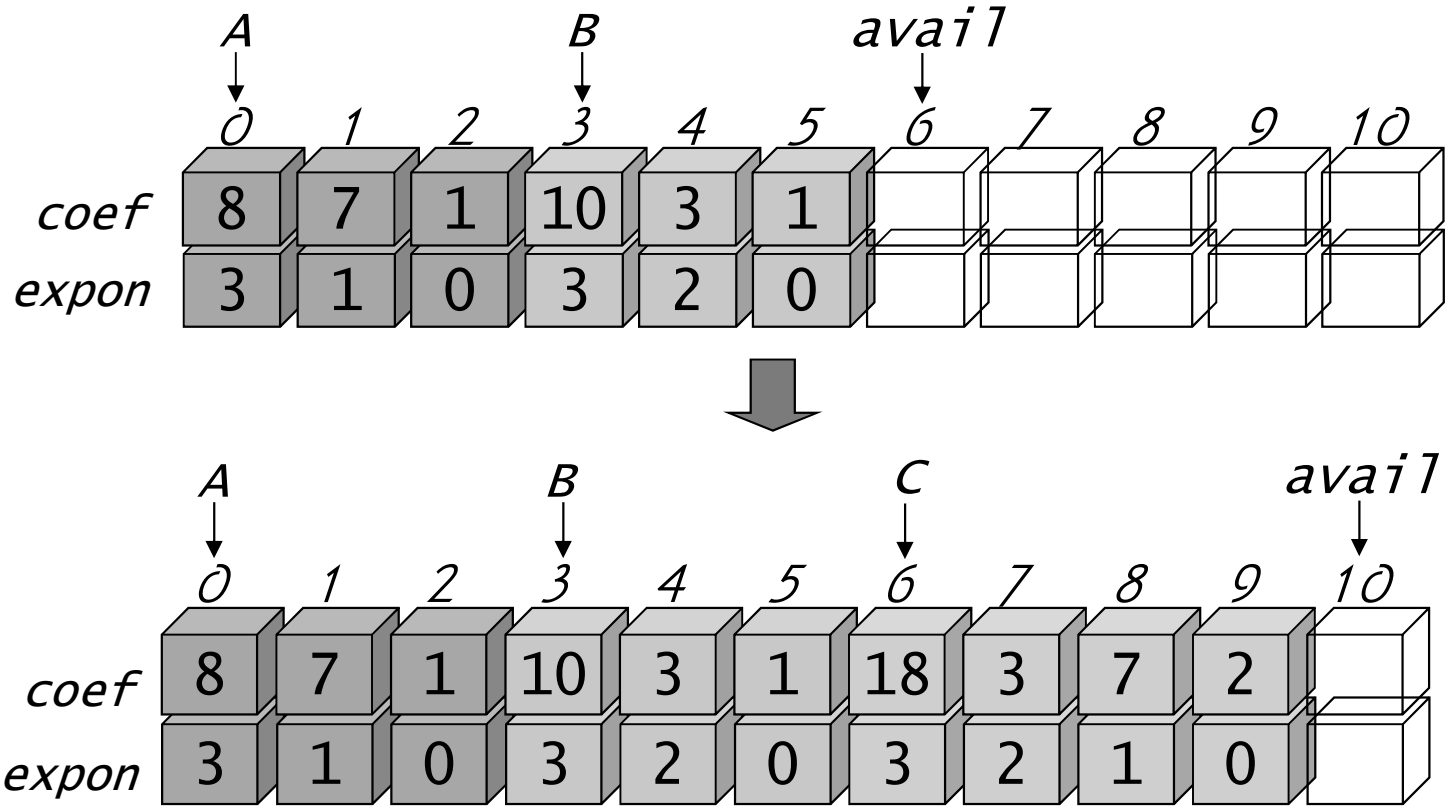
```
struct {
        float coef;
        int expon;
} terms[MAX_TERMS]={ {10,5}, {6,1}, {3,0} };
```

- 하나의 배열로 여러 개의 다항식을 나타낼 수 있음.



다항식 표현 방법 #2(계속)

- 장점: 메모리 공간의 효율적인 이용
- 단점: 다항식의 연산들이 복잡해진다(프로그램 3.3 참조).
  - (예) 다항식의 덧셈 $A=8x^3+7x+1$, $B=10x^3+3x^2+1$, $C=A+B$



희소 행렬

- 배열을 이용하여 행렬(matrix)를 표현하는 2가지 방법
  (1) 2차원 배열을 이용하여 배열의 전체 요소를 저장하는 방법
  (2) 0이 아닌 요소들만 저장하는 방법

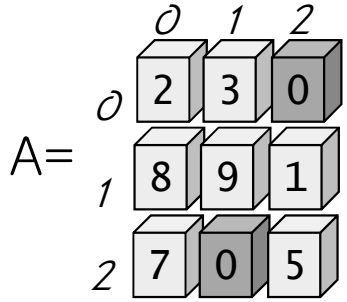
- 희소행렬: 대부분의 항들이 0인 배열

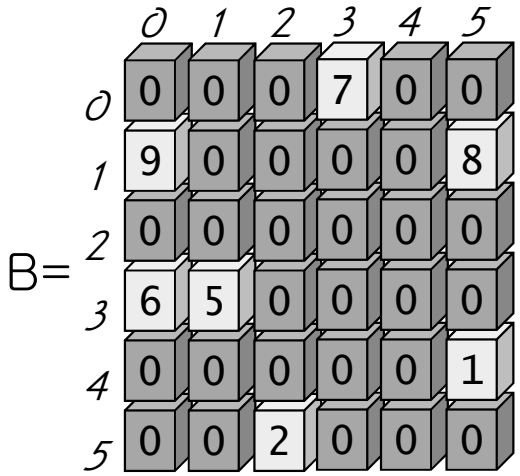
$$A = \begin{bmatrix} 2 & 3 & 0 \\ 8 & 9 & 1 \\ 7 & 0 & 5 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 0 & 0 & 7 & 0 & 0 \\ 9 & 0 & 0 & 0 & 0 & 8 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$

희소행렬 표현방법 #1

- 2차원 배열을 이용하여 배열의 전체 요소를 저장하는 방법
  - 장점: 행렬의 연산들을 간단하게 구현할 수 있다.
  - 단점: 대부분의 항들이 0인 희소 행렬의 경우 많은 메모리 공간 낭비

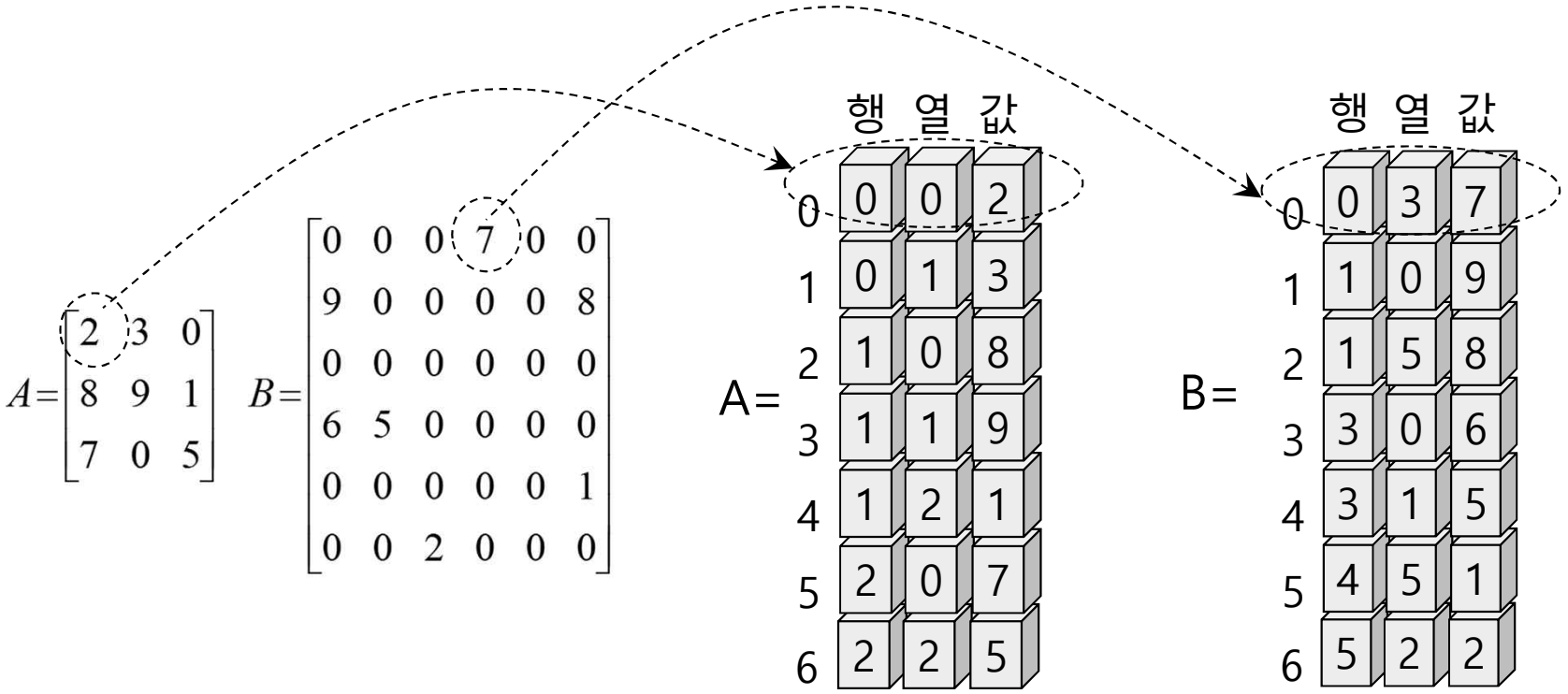
$$A = \begin{bmatrix} 2 & 3 & 0 \\ 8 & 9 & 1 \\ 7 & 0 & 5 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 0 & 0 & 7 & 0 & 0 \\ 9 & 0 & 0 & 0 & 0 & 8 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$



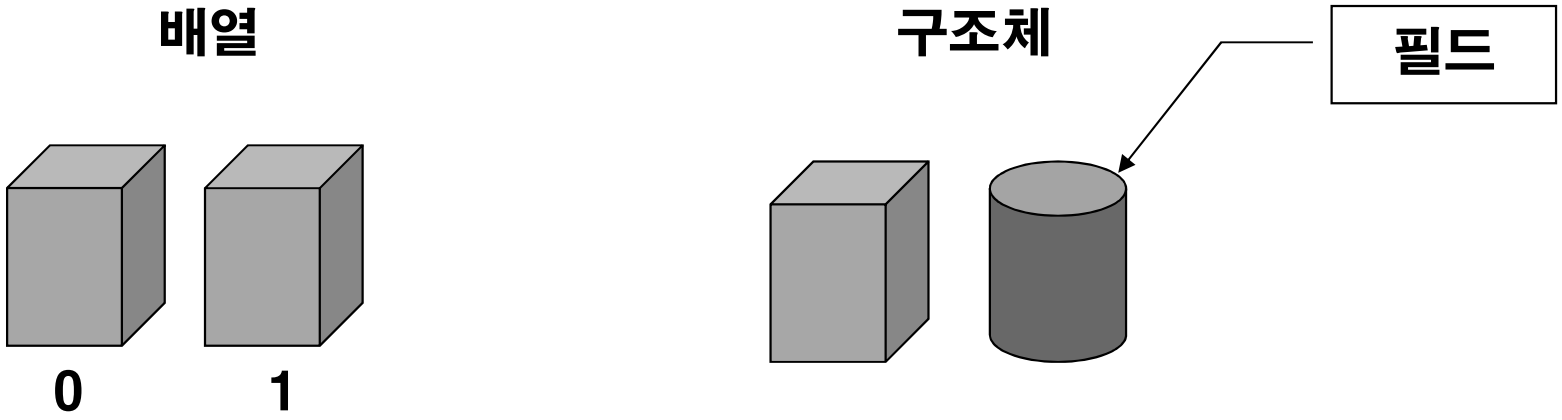


희소행렬 표현방법 #2

- 0이 아닌 요소들만 저장하는 방법
  - 장점: 희소 행렬의 경우, 메모리 공간의 절약
  - 단점: 각종 행렬 연산들의 구현이 복잡해진다.



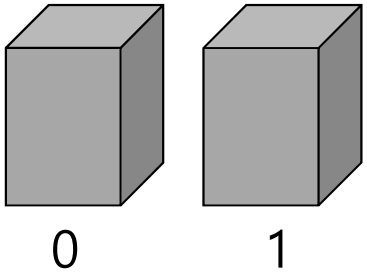
동일자료형 vs. 이질자료형



구조체

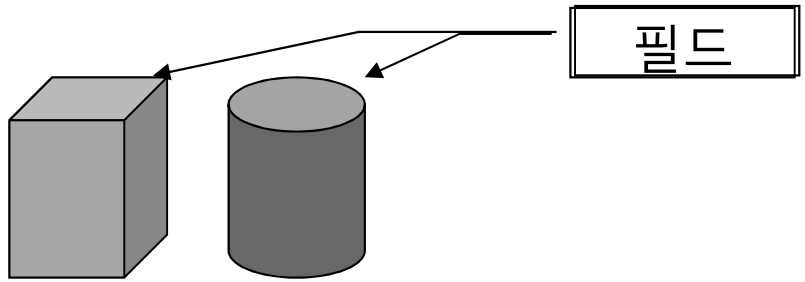
- 구조체(structure): 타입이 다른 데이터를 하나로 묶는 방법
- 배열(array): 타입이 같은 데이터들을 하나로 묶는 방법

배열




```
char carray[100];
```

구조체




```
struct example {
        char cfield;
        int ifield;
        float ffield;
        double dfield;
};
struct example s1;
```

구조체의 사용 예

- 구조체의 선언과 구조체 변수의 생성

```
struct person {
        char name[10];    // 문자배열로 된 이름
        int age;          // 나이를 나타내는 정수값
        float height;     // 키를 나타내는 실수값
};
struct person a;          // 구조체 변수 선언
```

- typedef을 이용한 구조체의 선언과 구조체 변수의 생성

```
typedef struct person {
        char name[10];    // 문자배열로 된 이름
        int age;          // 나이를 나타내는 정수값
        float height;     // 키를 나타내는 실수값
} person;
person a;                 // 구조체 변수 선언
```

구조체의 대입과 비교 연산

- 구조체 변수의 대입: 가능

```
struct person {
        char name[10];    // 문자배열로 된 이름
        int age;          // 나이를 나타내는 정수값
        float height;     // 키를 나타내는 실수값
};
main( ) {
        person a, b;
        b = a;            // 가능
}
```

- 구조체 변수끼리의 비교: 불가능

```
main( ) {
        if( a > b )
                printf("a가 b보다 나이가 많음");      // 불가능
}
```

자체참조 구조체

- **자체 참조 구조체(**self-referential structure**):**
  - 필드중에 자기 자신을 가리키는 포인터가 한 개 이상 존재하는 구조체

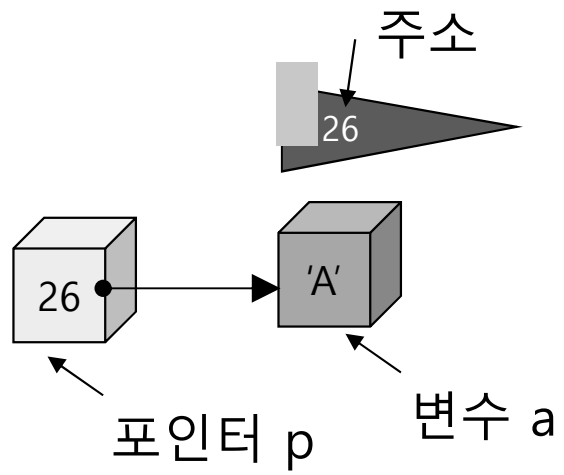
- 연결 리스트나 트리에 많이 등장

```
typedef struct   ListNode {
            char    data[10];
            struct ListNode *link;
} ListNode;
```

포인터(pointer)

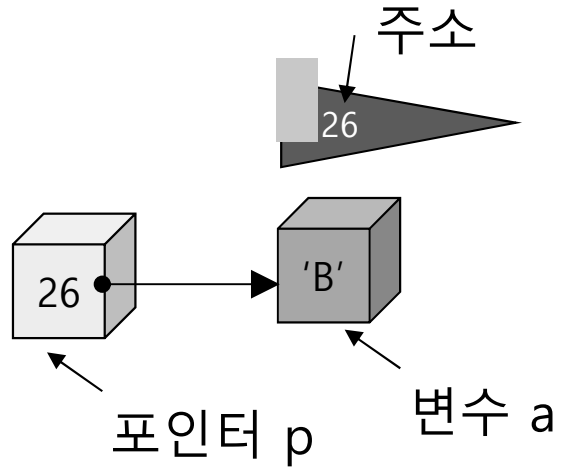
- 포인터: 다른 변수의 주소를 가지고 있는 변수

```
char a='A';
char *p;
p = &a;
```



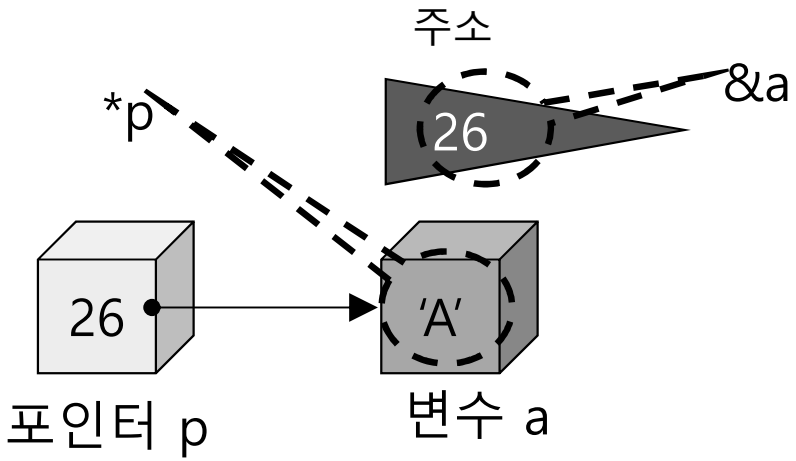
- 포인터가 가리키는 내용의 변경: * 연산자 사용

```
*p= 'B';
```



포인터와 관련된 연산자

- & 연산자:
  - 변수의 주소를 추출
- * 연산자:
  - 포인터가 가리키는 곳의 내용을 추출




```
 p       // 포인터
*p       // 포인터가 가리키는 값
*p++     // 포인터가 가리키는 값을 가져온 다음, 포인터를 한칸 증가.
*p--     // 포인터가 가리키는 값을 가져온 다음, 포인터를 한칸 감소.
(*p)++   // 포인터가 가리키는 값을 증가.
```

```
int a;    // 정수 변수 선언
int *p;   // 정수 포인터 선언
int **pp;         // 정수 포인터의 포인터 선언
p = &a;// 변수 a와 포인터 p를 연결
pp = &p;          // 포인터 p와 포인터의 포인터 pp를 연결
```

디양한 포인터

- 포인터의 종류

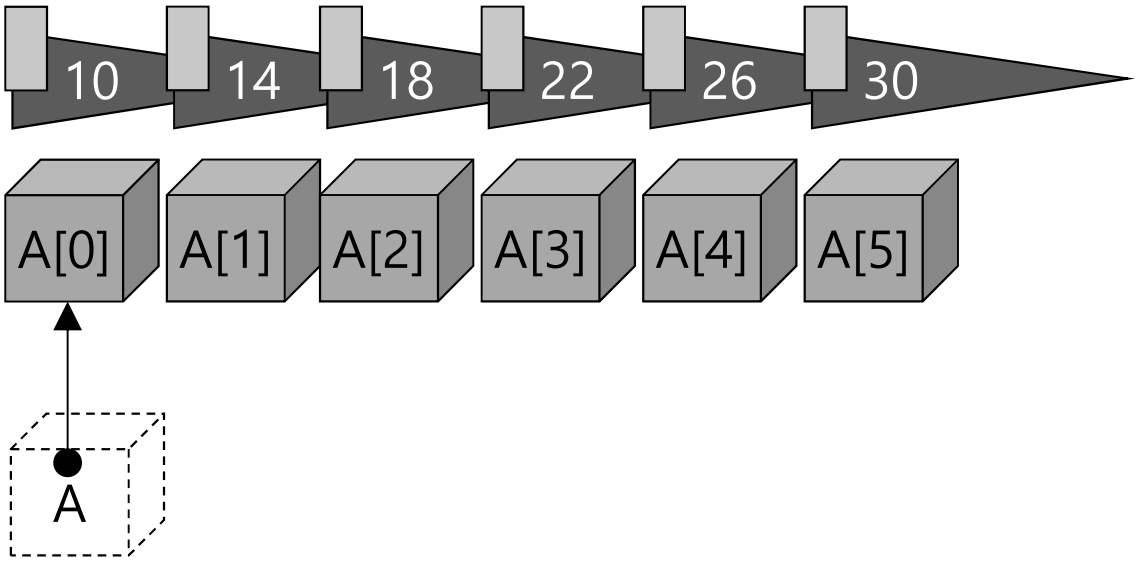
```
void *p;            // p는 아무것도 가리키지 않는 포인터
int *pi;            // pi는 정수 변수를 가리키는 포인터
float *pf;          // pf는 실수 변수를 가리키는 포인터
char *pc;           // pc는 문자 변수를 가리키는 포인터
int **pp;           // pp는 포인터를 가리키는 포인터
struct test *ps; // ps는 test 타입의 구조체를 가리키는 포인터
void (*f)(int) ;   // f는 함수를 가리키는 포인터
```

- 포인터의 형변환: 필요할 때마다 형 변환 가능

```
void *p;
pi=(int *) p;
```

배열과 포인터

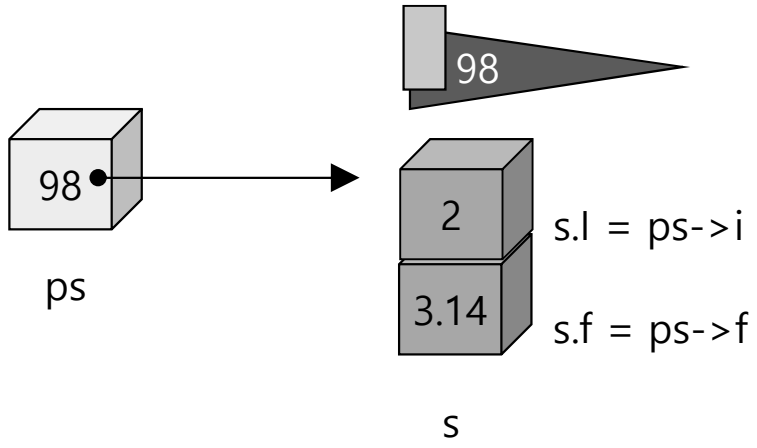
- 배열의 이름: 사실상의 포인터와 같은 역할



- 컴파일러가 배열의 이름을 배열의 첫번째 주소로 대치

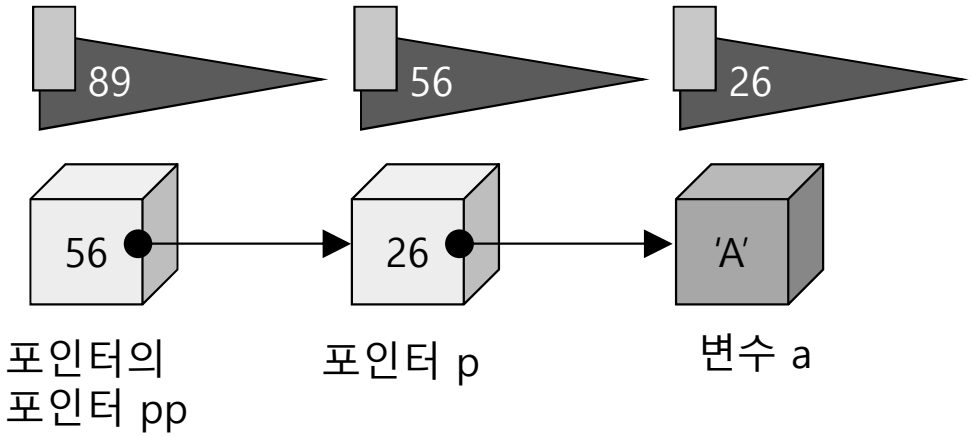
구조체의 포인터

- 구조체의 요소에 접근하는 연산자: ->




```
main( )  {
        struct {
                int     i;
                float f;
        } s, *ps;
        ps = &s;
        ps->i = 2;
        ps->f = 3.14;
}
```

포인터의 포인터

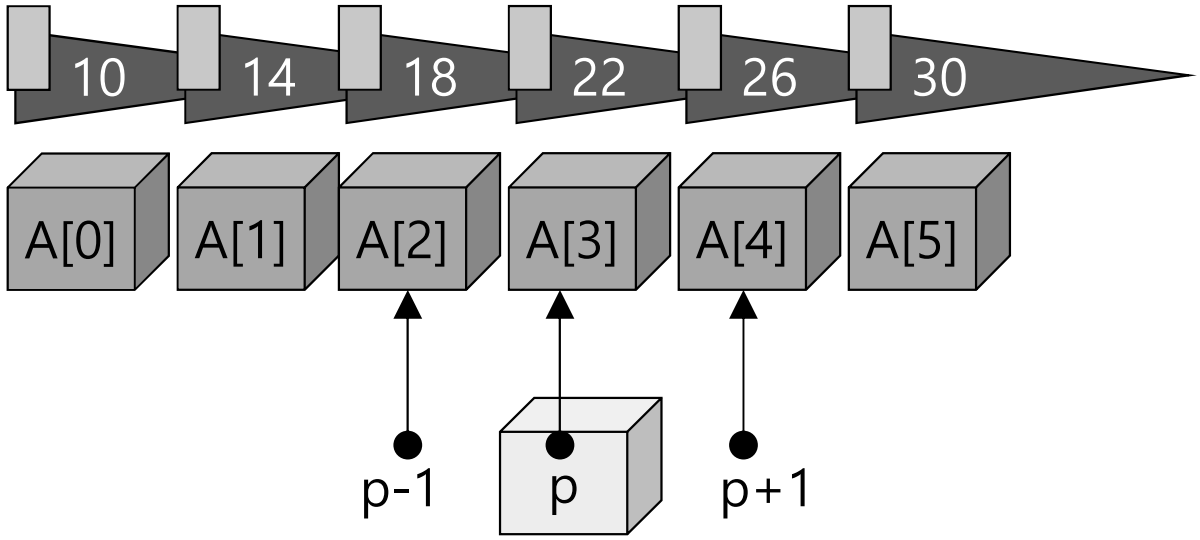



```
int a;          // 정수 변수 변수 선언
int *p;         // 정수 포인터 선언
int **pp;       // 정수 포인터의 포인터 선언
p = &a;         // 변수 a와 포인터 p를 연결
pp = &p;        // 포인터 p와 포인터의 포인터 pp를 연결
```

포인터 연산

- 포인터에 대한 사칙연산: 포인터가 가리키는 객체단위로 계산

```
p       // 포인터
p+1     // 포인터 p가 가리키는 객체의 바로 뒤 객체
p-1     // 포인터 p가 가리키는 객체의 바로 앞 객체
```



포인터 사용시 주의할 점

- 포인터가 아무것도 가리키고 있지 않을 때는 NULL로 설정
  - int *pi=NULL;
- 초기화가 안된 상태에서 사용 금지

```
main() {
        char *pc;       // 포인터 pi는 초기화가 안되어 있음
        *pc = 'E';      // 위험한 코드
}
```

- 포인터 타입간의 변환시에는 명시적인 타입 변환 사용

```
int *pi;
float *pf;
pf = (float *)pi;
```

동적 메모리 할당

- 프로그램이 메모리를 할당받는 방법
  - 정적 메모리(static memory)
  - 동적 메모리(dynamic memory)

- 정적 메모리 할당
  - 메모리의 크기는 프로그램이 시작하기 전에 결정
  - 프로그램의 수행 도중에 그 크기가 변경될 수는 없다.
  - 만약 처음에 결정된 크기보다 더 큰 입력이 들어온다면 처리하지 못할 것이고 더 작은 입력이 들어온다면 남은 메모리 공간은 낭비될 것이다.
  - (예) 변수나 배열의 선언
  - int buffer[100];
    char name[] = “data structure";

- **동적 메모리 할당**
  - 프로그램의 실행 도중에 메모리를 할당받는 것
  - 필요한 만큼만 할당을 받고 또 필요한 때에 사용하고 반납
  - 메모리를 매우 효율적으로 사용가능

동적 메모리 할당

- 전형적인 동적 메모리 할당 코드

```
main( ) {
    int *pi;
    pi = (int *)malloc(sizeof(int));        // 동적 메모리 할당
    …
    …                                       // 동적 메모리 사용
    …
    free(pi);                               // 동적 메모리 반납
}
```

- 동적 메모리 할당 관련 라이브러리 함수
  - malloc(size) // 메모리 할당
  - free(ptr) // 메모리 할당 해제
  - sizeof(var) // 변수나 타입의 크기 반환(바이트 단위)

동적 메모리 할당 라이브러리

- malloc(int size)
  - size 바이트 만큼의 메모리 블록을 할당

```
(char *)malloc(100) ;   /* 100 바이트로 50개의 정수를 저장 */
(int *)malloc(sizeof(int));/* 정수 1개를 저장할 메모리 확보*/
(struct Book *)malloc(sizeof(struct Book))/* 하나의 구조체 생성 */
```

- free(void ptr)
  - ptr이 가리키는 할당된 메모리 블록을 해제
- sizeof 키워드
  - 변수나 타입의 크기 반환(바이트 단위)

```
size_t  i = sizeof( int );                                  // 4
struct AlignDepends {
    char c;
    int i;
};
size_t size = sizeof(struct AlignDepends);                  // 8
int  array[] = { 1, 2, 3, 4, 5 };
size_t  sizearr = sizeof( array ) / sizeof( array[0] );     // 20/4=5
```

동적 메모리 할당 예제

```
struct Example {
        int number;
        char name[10];
};
void main()
{
        struct Example *p;

        p=(struct Example *)malloc(2*sizeof(struct Example));
        if(p==NULL){
          fprintf(stderr, "can't allocate memory\n") ;
          exit(1) ;
        }
        p->number=1;
        strcpy(p->name,"Park");
        (p+1)->number=2;
        strcpy((p+1)->name,"Kim");
        free(p);
}
```