CSE2010 자료구조론

Week 3: Recursion, Array, Struct, Pointer

ICT융합학부 조용우

목차

- Recursion
- Array, Struct, Pointer

Recursion

순환(Recursion)

- 알고리즘이나 함수가 수행 도중에 자기 자신을 다시 호출하여 문제를 해결하는 방법
  - 정의자체가 순환적으로 되어 있는 경우에 적합
- 함수의 순환(재귀)호출(recursive call) 사용
  - 문제 또는 사용되는 자료구조가 재귀적으로 정의되어 있을 때
  - 문제 해결이 용이, 알고리즘 정확성 증명이 용이
  - 시간, 공간 사용이 비효율적임

순환(Recursion) 예

- 팩토리얼

$$n! = \begin{cases} 1 & n = 0 \text{、} \\ n*(n-1)! & n \geq 1 \end{cases}$$

- 피보나치 수열

$$fib(n) = \begin{cases} 0 & if \quad n = 0 \\ 1 & if \quad n = 1 \\ fib(n-2) + fib(n-1) & otherwise \end{cases}$$

- 이항계수

$${}_nC_k = \begin{cases} 1 & n = 0 \quad or \quad n = k \\ {}_{n-1}C_{k-1} + {}_{n-1}C_k & otherwise \end{cases}$$

- 하노이탑, 이진탐색, ...

팩토리얼 구현

- 팩토리얼 정의

$$n! = \begin{cases} 1 & n = 0 \\ n*(n-1)! & n \geq 1 \end{cases}$$

- 팩토리얼 구현

```
int factorial(int n)
{
   if( n <= 1 )
         return(1);
   else
         return (n * factorial(n-1));
}
```

팩토리얼 호출 순서

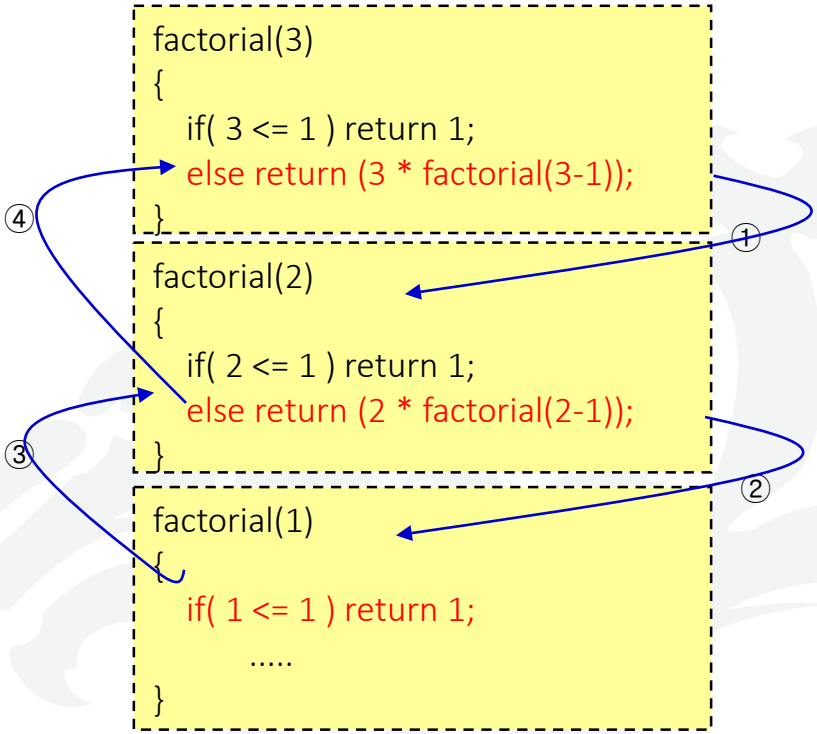
- factorial(3)?

factorial(3) = 3 * factorial(2)

= 3 * 2 * factorial(1)

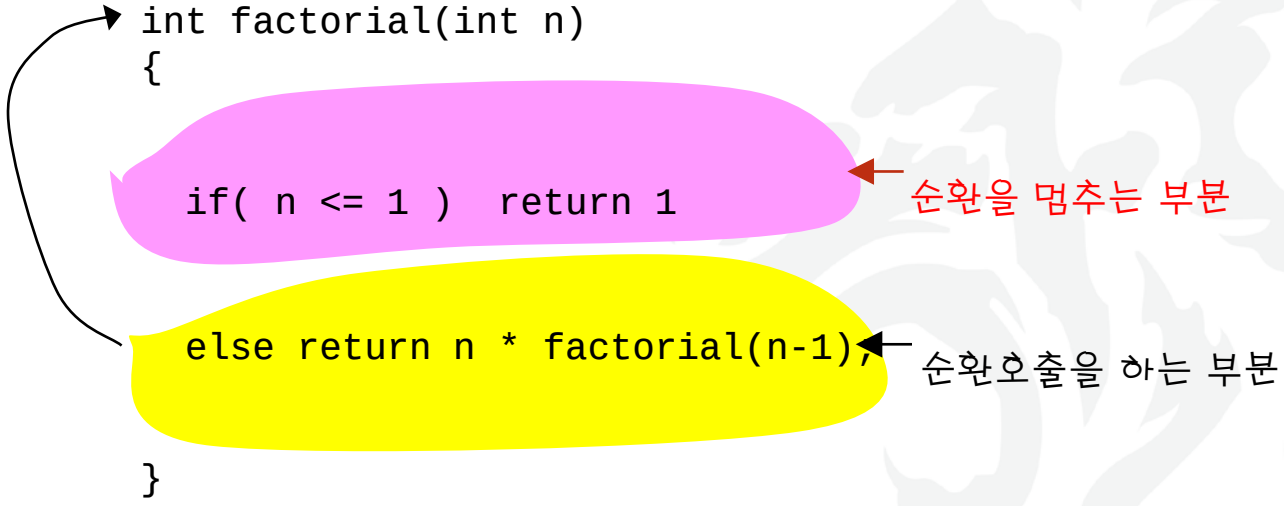
= 3 * 2 * 1

= 6



순환 알고리즘의 구조

- 순환 알고리즘은 다음의 부분을 포함
  - 순환 호출을 하는 부분
  - 순환 호출을 멈추는 부분



- 만약 순환 호출 멈추는 부분이 없다면?
  - 시스템 오류가 발생할 때까지 무한정 호출

순환 vs. 반복

- 컴퓨터에서의 되풀이
  - "순환(recursion)"
    - 순환적인 문제에서는 자연스러운 방법
    - 함수 호출의 오버헤드
  - "반복(iteration)"
    - 수행속도가 빠름
    - 순환적인 문제에 대해서는 프로그램 작성이 어려울 수 있음
- 대부분의 순환은 반복으로 바꾸어 작성할 수 있음

팩토리얼 함수의 반복구현

- 팩토리얼 함수 반복적 정의

$$n! = 1 \qquad \text{if} \quad n = 0$$

$$n! = \underbrace{n \times (n-1) \times (n-2) \times \ldots \times 1}_{\text{loop}} \qquad \text{if} \quad n > 0$$

```
int factorial_iter(int n)
{
   int i, f = 1;
   for(i=n; i>-; i--)
         f = f * i;
   return(f);
}
```

복잡도 비교

- 공간 복잡도

$$S_I(n) = \Theta(1)$$

$$S_R(n) = \Theta(n)$$

- 시간 복잡도

$$T_I(n) = n = \Theta(n)$$

$$\begin{aligned}
T_R(0) &= 0 \\
T_R(n) &= 1 + T_R(n-1) \\
&= 1 + 1 + T_R(n-2) \\
&= 1 + 1 + 1 + T_R(n-3) \\
&\cdots \\
&= \underbrace{1 + 1 + \ldots + 1}_{n} + T_R(0) \\
&= n \\
&= \Theta(n)
\end{aligned}$$

거듭제곱 프로그래밍(1)

- 숫자 x의 n제곱값을 구하는 문제: x^n
- 반복적인 방법

```
double slow_power(double x, int n)
{
   int i;
  double r = 1.0;
  for(i=0; i<n; i++)
      r = r * x;
  return(r);
}
```

거듭제곱 프로그래밍(2)

- 순환적인 방법

```
power(x, n)

if n=0
        then return 1;
else if n이 짝수
        then return power(x², n/2);
else if n이 홀수
        then return x*power(x², (n-1)/2);
```

즉 n이 짝수이면 다음과 같이 계산하는 것이다.

$$\begin{aligned} power(x, n) &= power(x^2, n / 2) \\ &= (x^2)^{n/2} \\ &= x^{2(n/2)} \\ &= x^n \end{aligned}$$

만약 n이 홀수이면 다음과 같이 계산하는 것이다.

$$\begin{aligned} power(x, n) &= x \cdot power(x^2, (n-1) / 2) \\ &= x \cdot (x^2)^{(n-1)/2} \\ &= x \cdot x^{n-1} \\ &= x^n \end{aligned}$$

```
double power(double x, int n)
{
        if( n==0 ) return 1;
        else if ( (n%2)==0 )
                return power(x*x, n/2);
        else return x*power(x*x, (n-1)/2);
}
```

시간 복잡도 비교

- 순환적인 방법
  - 만약 n이 2의 거듭 제곱인 2^k 이라고 가정하면 다음과 같이 문제의 크기가 줄어듬
  - $n = 2^k$ -> $k=\log n$

$$\underbrace{2^k \rightarrow 2^{k-1} \rightarrow \cdots 2^2 \rightarrow 2^1 \rightarrow 2^0}_{k=\log n}$$

- 반복적인 방법과의 비교

	반복적인 함수 slow_power	순환적인 함수 power
시간복잡도	O(n)	O(logn)

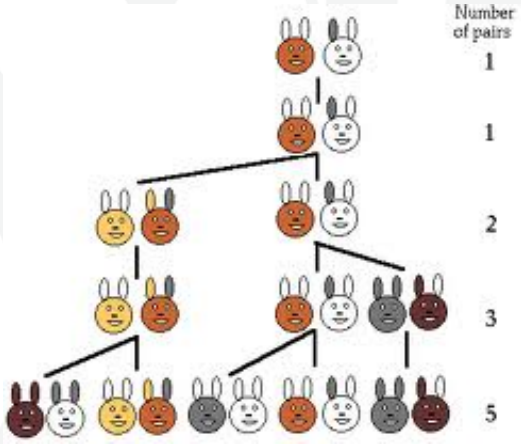
이 경우에, 순환적인 방법이 반복적인 방법보다 시간 복잡도 측면에서 더 효율적임!

토끼 번식 문제

문제	**토끼장에 바로 막 태어난 암수 토끼 한쌍을 넣었다. 다음 가정 하에 일년 후 이 토끼장에는 몇쌍이 있겠는가?**
가정	**(1) 토끼 한쌍은 새로 태어난지 한달후 성숙해진다.** **(2) 토끼 한쌍은 성숙해진지 한달후부터 매달 새로운 암수 토끼 한쌍을 낳는다.** **(3) 토끼는 죽지 않는다.**

- 여섯달 후의 토끼의 쌍 수는?
  - F_i : i번째 달의 토끼 쌍수
  - r_i : i번째 태어난 토끼 쌍
  - R_i : 성숙한 r_i

$F_0 = 0$
$F_1 = 1$ (최초의 토끼 한 쌍 r_1)
$F_2 = 1$ (R_1)
$F_3 = 2$ (R_1, ($R_1 \rightarrow$)r_2)
$F_4 = 3$ (R_1, R_2, ($R_1 \rightarrow$)r_3)
$F_5 = 5$ (R_1, R_2, R_3, ($R_1 \rightarrow$)r_4, ($R_2 \rightarrow$)r_5)
$F_6 = 8$ (R_1, R_2, R_3, R_4, R_5, ($R_1 \rightarrow$)r_6, ($R_2 \rightarrow$)r_7, ($R_3 \rightarrow$)r_8)



피보나치 수열

- i번째 달의 토끼 쌍의 수

  $= (i-1)$ 번째 달의 토끼 쌍 + i 번째 달에 새로 태어난 토끼 쌍

  $= (i-1)$ 번째 달의 토끼 쌍 + $(i-1)$ 번째 달의 성숙한 토끼 쌍

  $= (i-1)$ 번째 달의 토끼 쌍 + $(i-2)$ 번째 달의 토끼 쌍

$$F_i = F_{i-1} + F_{i-2}$$ 피보나치 수열

- 피보나치 수열 정의

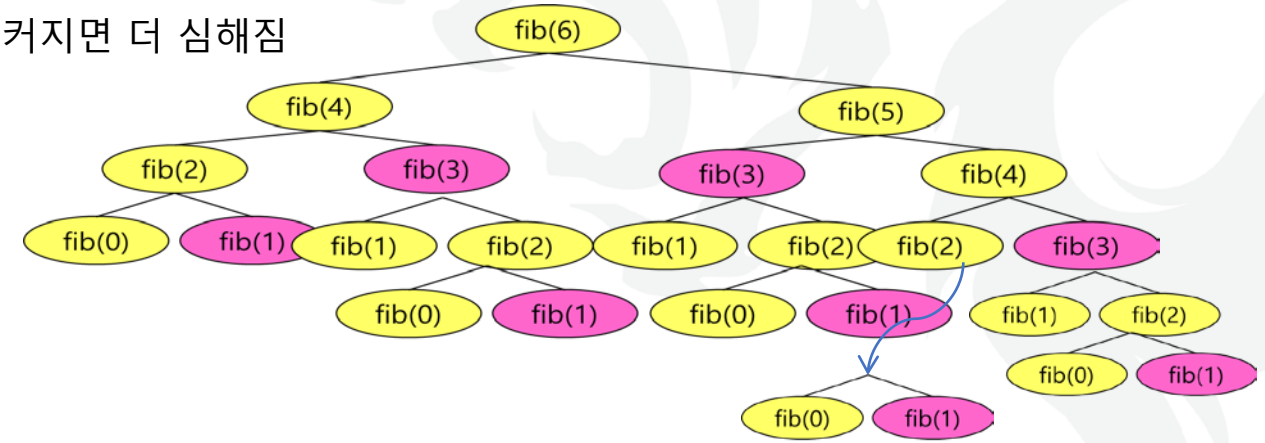
$$fib(n)\begin{cases} 0 & n=0 \\ 1 & n=1 \\ fib(n-2)+fib(n-1) & otherwise \end{cases}$$

피보나치 수열의 순환적 계산 방법

- 순환적인 방법

```
int fib(int n)
{
   if( n==0 ) return 0;
   if( n==1 ) return 1;
   return (fib(n-1) + fib(n-2));
}
```

- 비효율성
  - 같은 항이 중복해서 계산됨
  - 예를 들어 fib(6)을 호출하게 되면 fib(3)이 3번이나 중복되어서 계산됨
  - 이러한 현상은 n이 커지면 더 심해짐



피보나치 수열의 반복적 계산

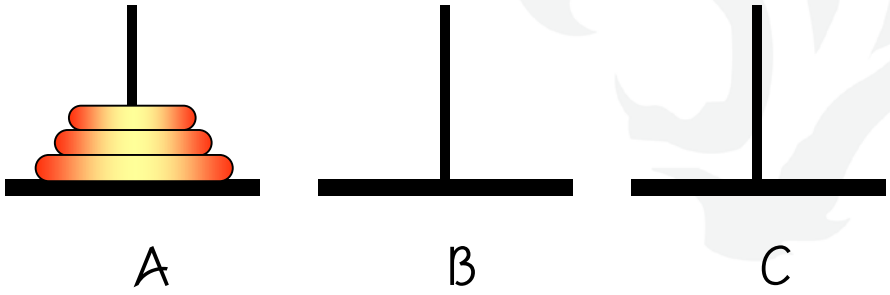
- 반복 구조를 사용한 구현

```
int fib_iter(int n)
{
        if( n < 2 ) return n;
        else {
                int i, tmp, current=1, last=0;
                for(i=2;i<=n;i++){
                        tmp = current;
                        current += last;
                        last = tmp;
                }
                return current;
        }
}
```

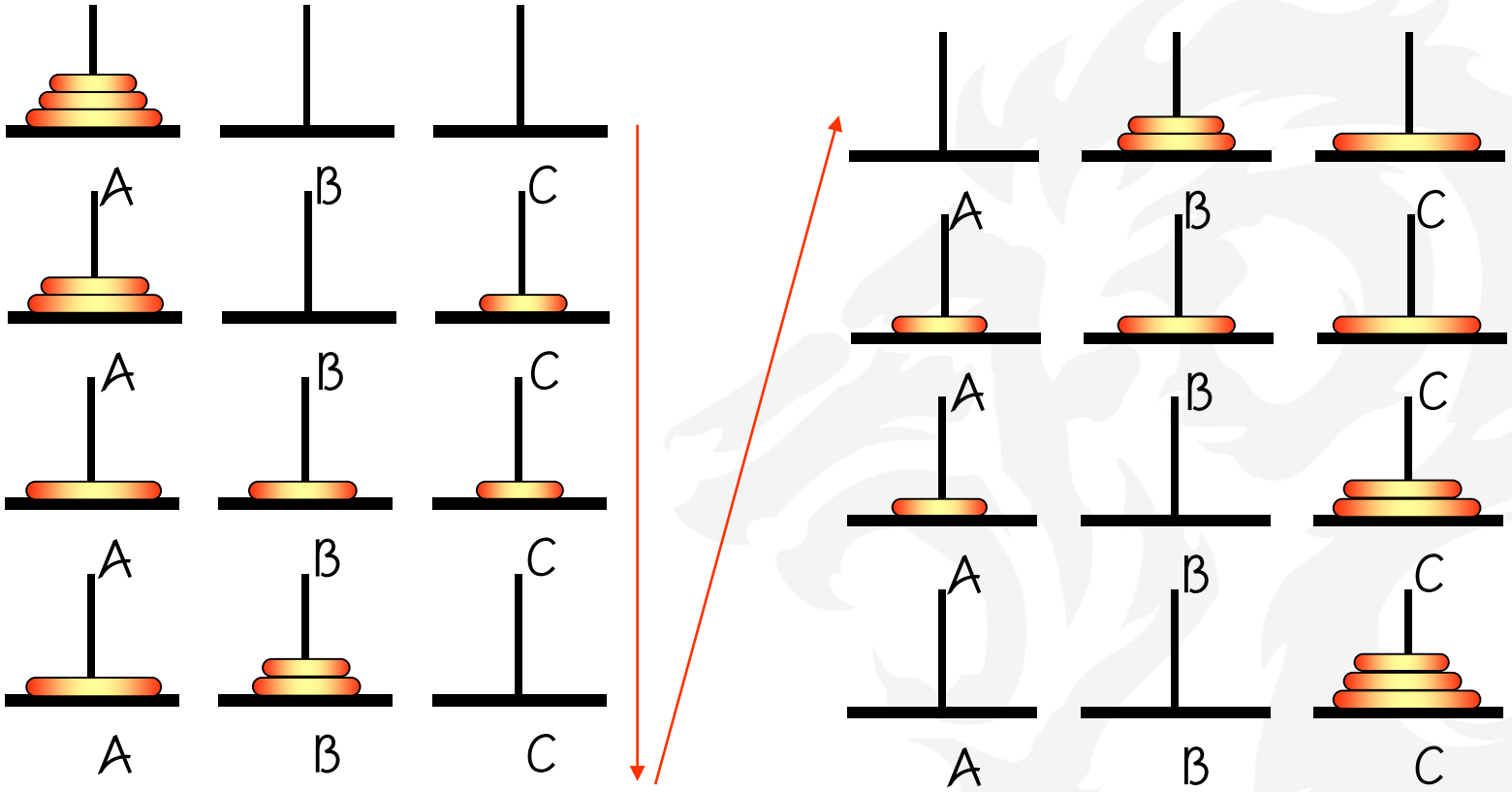
이 경우에, 반복적인 방법이 순환적인 방법보다 더 효율적임!

하노이탑 문제

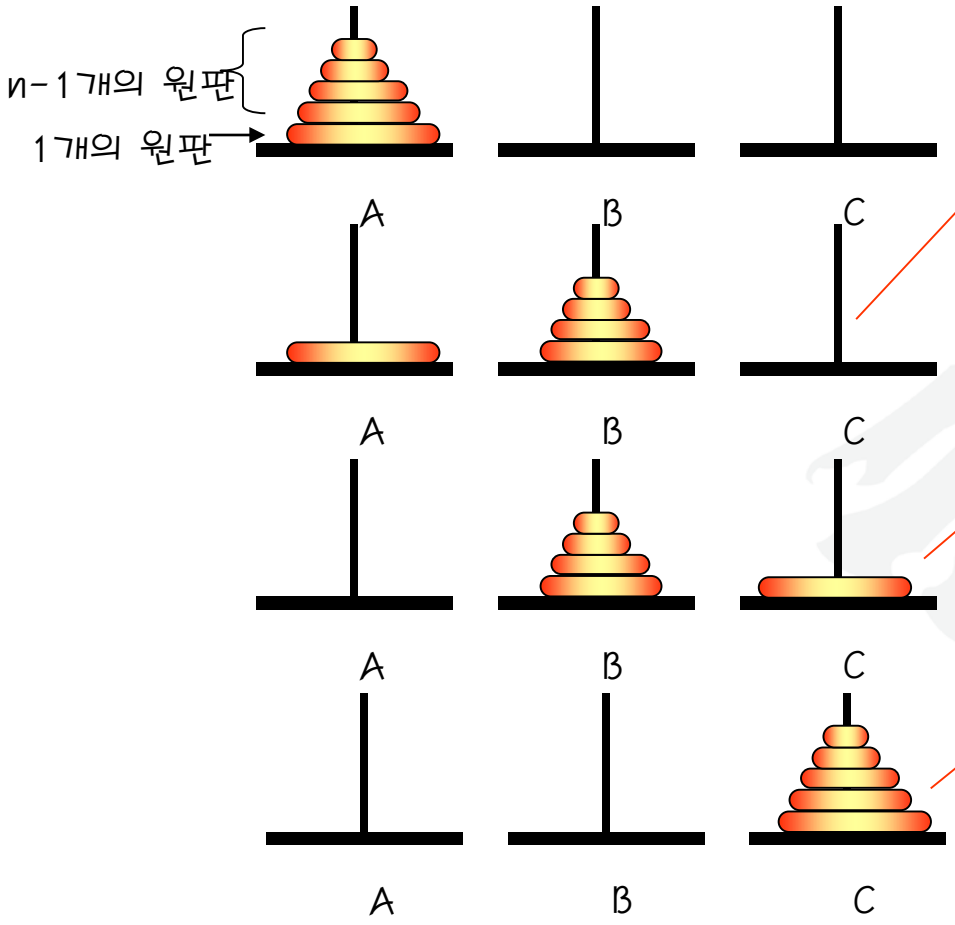
- 하노이탑 문제: 막대 A에 쌓여있는 원판 n개를 막대 C로 옮기는 것
- 다음의 조건을 지켜야 함
  - 한 번에 하나의 원판만 이동할 수 있음
  - 맨 위에 있는 원판만 이동할 수 있음
  - 크기가 작은 원판 위에 큰 원판이 쌓일 수 없음
  - 중간의 막대를 임시적으로 이용할 수 있으나 앞의 조건들을 지켜야 함



하노이탑 문제: n=3인 경우



하노이탑 문제: 일반적인 경우



C를 임시버퍼로 사용하여 A에 쌓여있는 n-1개의 원판을 B로 옮김

A의 가장 큰 원판을 C로 옮김

A를 임시버퍼로 사용하여 B에 쌓여있는 n-1개의 원판을 C로 옮김

하노이탑 문제: 순환방법 도입

- 어떻게 n-1개의 원판을 A에서 B로, 또 B에서 C로 이동하는가?
  - (힌트) 우리의 원래 문제는 n개의 원판을 A에서 C로 옮기는 것임
- 작성하고 있는 함수의 파라미터를 n-1로 바꾸어 순환 호출!

```
// 막대 from에 쌓여있는 n개의 원판을 막대 tmp를 사용하여 막대 to로 옮긴다.
void hanoi_tower(int n, char from, char tmp, char to)
{
   if (n==1){
      from에서 to로 원판을 옮긴다.
   }
   else{
      (1) hanoi_tower(n-1, from, to, tmp); //from의 맨 밑에 있는 원판을 제외한
     나머지 원판들을 tmp로 옮긴다.
      (2) from에 있는 한 개의 원판을 to로 옮긴다.
      (3) hanoi_tower(n-1, tmp, from, to); // tmp의 원판들을 to로 옮긴다.
   }
}
```

하노이탑 문제의 구현

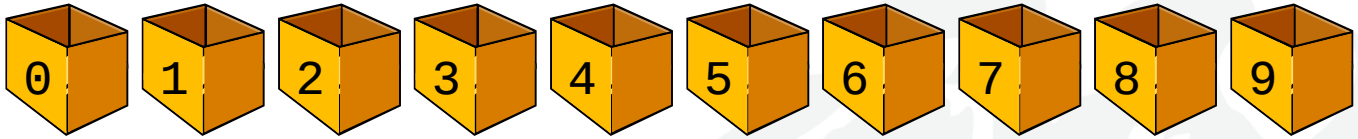
- 기본 아이디어: n-1개의 원판을 A에서 B로 옮기고, n번째 원판을 A에서 C로 옮긴 다음, n-1개의 원판을 B에서 C로 옮김

```
#include <stdio.h>
void hanoi_tower(int n, char from, char tmp, char to)
{
  if( n==1 ) printf("원판 1을 %c 에서 %c으로 옮긴다.\n",from,to);
    else {
       hanoi_tower(n-1, from, to, tmp);
            printf("원판 %d을 %c에서 %c으로 옮긴다.\n",n, from, to);
       hanoi_tower(n-1, tmp, from, to);
  }
}
main()
{
    hanoi_tower(4, 'A', 'B', 'C');
}
```

Array, Struct, Pointer

배열

- 동일한 데이터 타입의 데이터를 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 ADT

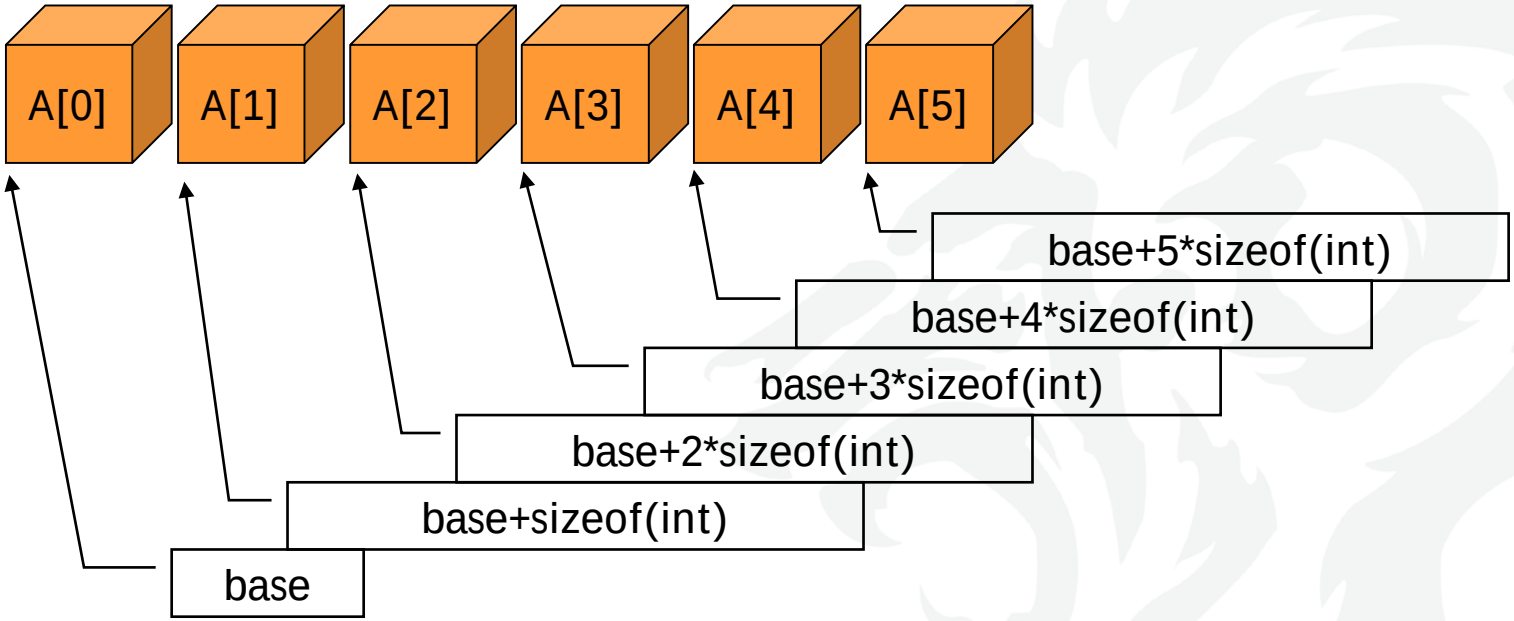
- 배열 ADT

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.

1차원 배열

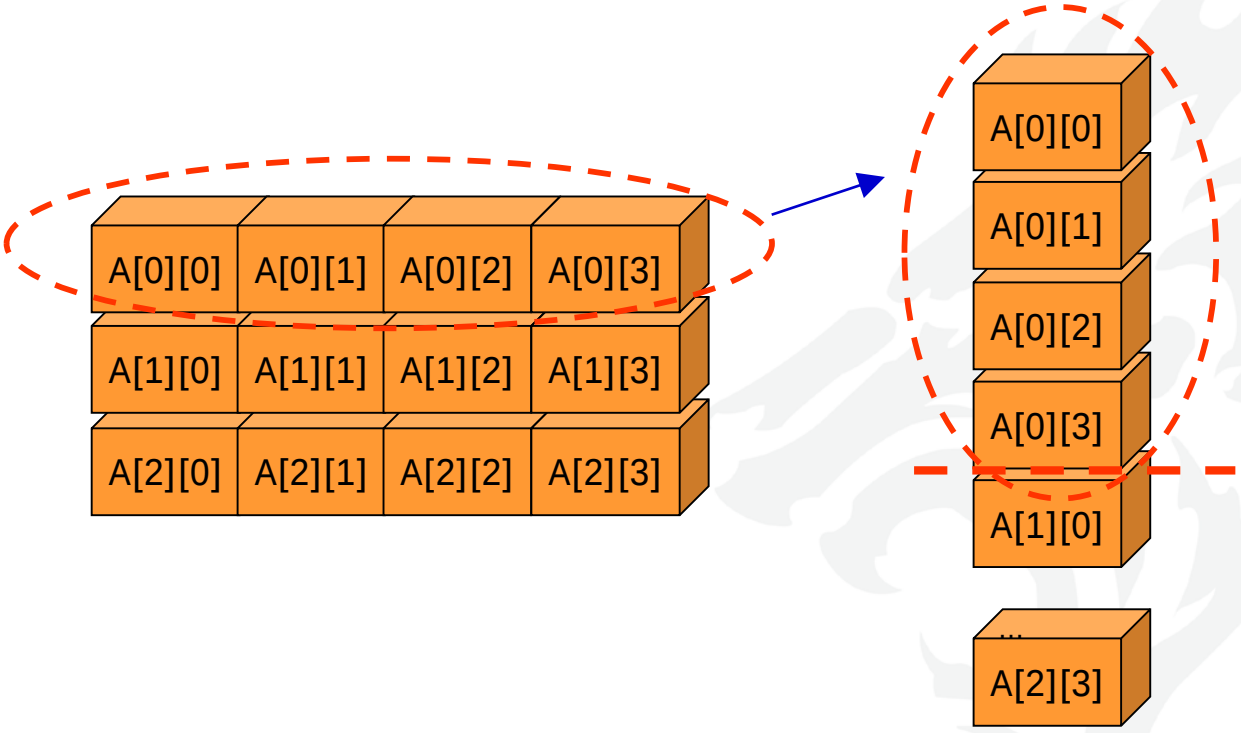
- int A[6];



- 배열 인덱스는 0부터 시작
- base: 메모리상의 기본 주소
- 배열은 메모리의 연속된 위치에 구현 되기때문에, A[0]의 주소가 base가 됨

2차원 배열

- int A[3][4];



실제 메모리 안에서의 위치

배열의 응용: 다항식

- 다항식의 일반적인 형태

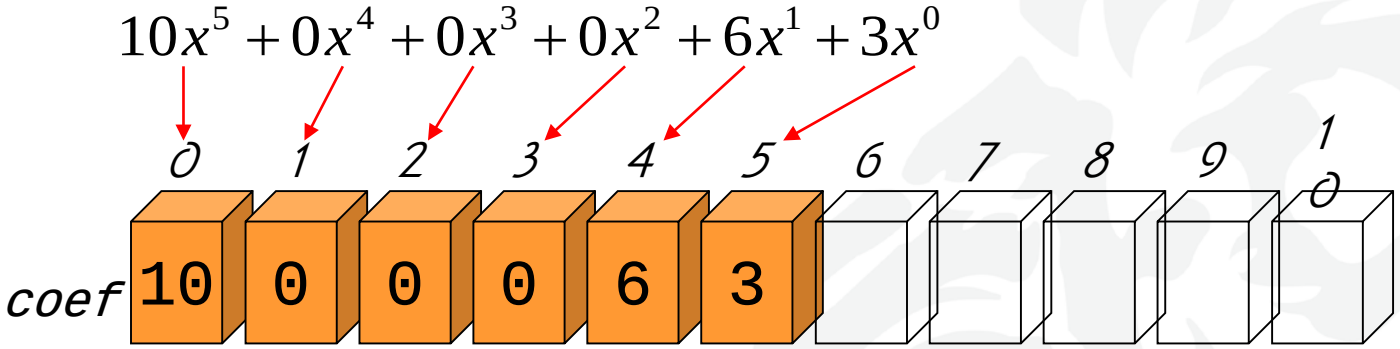
$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

- 프로그램에서 다항식을 처리하려면 다항식을 위한 자료구조가 필요함
  - 어떤 자료구조를 사용해야 다항식의 덧셈, 뺄셈 ,곱셈, 나눗셈 등의 연산을 할 때 편리하고 효율적일까?

- 배열을 사용한 2가지 방법
  - 다항식의 모든 항을 배열에 저장
  - 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} };
```

장점: 다항식의 각종 연산이 간단해짐
단점: 대부분의 항의 계수가 0이면 공간의 낭비가 심함

다항식 덧셈 연산 #1 (1)

```
#include <stdio.h>
#define MAX(a,b) (((a)>(b))?(a):(b))
#define MAX_DEGREE 101
typedef struct {                    // 다항식 구조체 타입 선언
    int degree;                     // 다항식의 차수
    float coef[MAX_DEGREE];         // 다항식의 계수
} polynomial;
```

다항식 덧셈 연산 #1 (2)

```
// C = A+B 여기서 A와 B는 다항식이다.
polynomial poly_add1(polynomial A, polynomial B)
{
    polynomial C;                          // 결과 다항식
    int Apos=0, Bpos=0, Cpos=0;     // 배열 인덱스 변수
    int degree_a=A.degree;
    int degree_b=B.degree;
    C.degree = MAX(A.degree, B.degree); // 결과 다항식 차수
    while( Apos<=A.degree && Bpos<=B.degree ){
        if( degree_a > degree_b ){ // A항 > B항
          C.coef[Cpos++]= A.coef[Apos++];
          degree_a--;
        }
```

다항식 덧셈 연산 #1 (3)

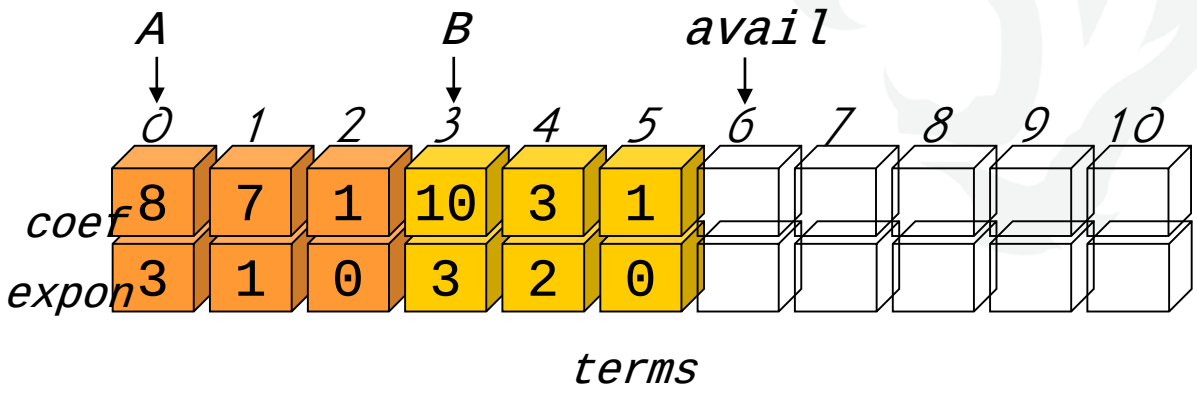
```
        else if( degree_a == degree_b ){ // A항 == B항
                C.coef[Cpos++]=A.coef[Apos++]+B.coef[Bpos++];
                degree_a--; degree_b--;
              }
        else {                      // B항 > A항
                C.coef[Cpos++]= B.coef[Bpos++];
                degree_b--;
              }
        }
        return C;
}
// 주함수
main()
{
        polynomial a = { 5, {3, 6, 0, 0, 0, 10} };
        polynomial b = { 4, {7, 0, 5, 0, 1} };
        polynomial c;
        c = poly_add1(a, 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))

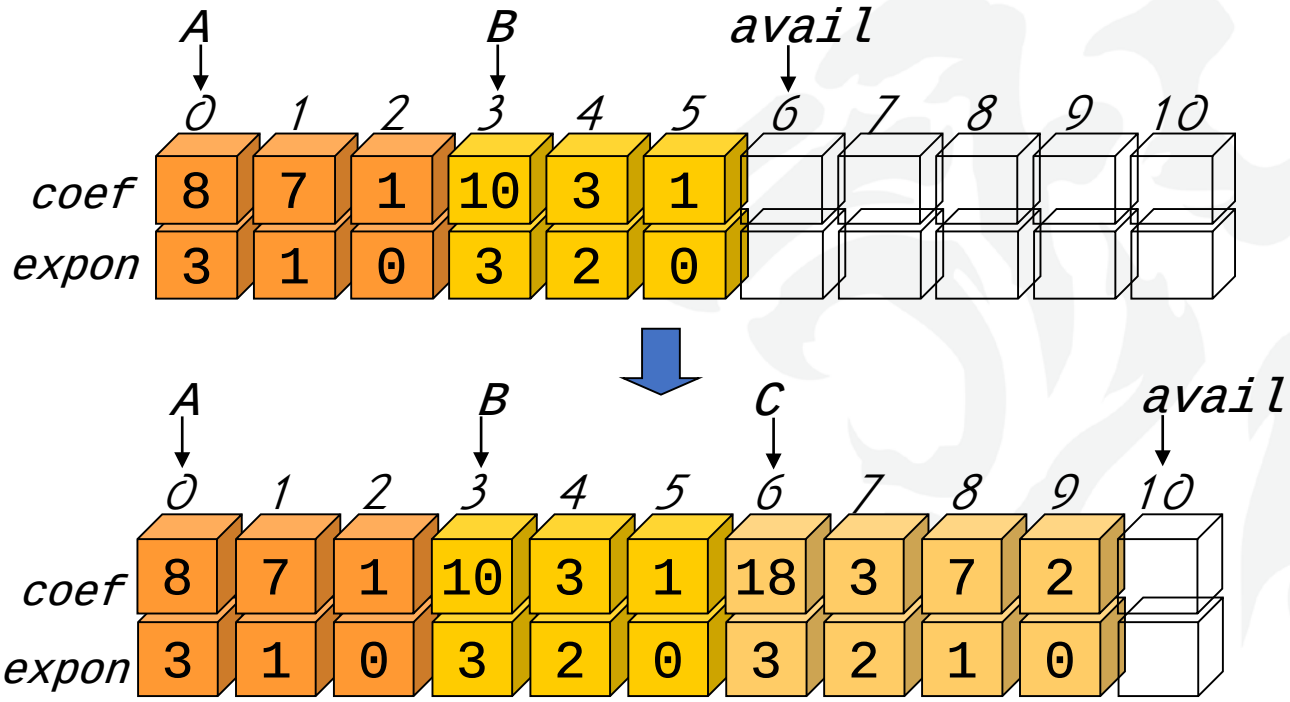
```
struct {
        float coef;
        int expon;
} terms[MAX_TERMS]={ {10,5}, {6,1}, {3,0} };
```

- 하나의 배열로 여러 개의 다항식을 나타낼 수 있음



다항식 표현 방법 #2 (계속)

- 장점: 메모리 공간의 효율적인 이용
- 단점: 다항식의 연산들이 복잡해짐
  - 예: 다항식의 덧셈 $A=8x^3+7x+1$, $B=10x^3+3x^2+1$, $C=A+B$



다항식 덧셈 연산 #2 (1)

```
#define MAX_TERMS 101
struct {
    float coef;
    int expon;
} terms[MAX_TERMS]={ {8,3}, {7,1}, {1,0}, {10,3}, {3,2},{1,0} };
int avail=6;
// 두 개의 정수를 비교
char compare(int a, int b)
{
    if( a>b ) return '>';
    else if( a==b ) return '=';
    else return '<';
}
```

다항식 덧셈 연산 #2 (2)

```
// 새로운 항을 다항식에 추가한다.
void attach(float coef, int expon)
{
    if( avail>MAX_TERMS ){
        fprintf(stderr, "항의 개수가 너무 많음\n");
        exit(1);
    }
    terms[avail].coef=coef;
    terms[avail++].expon=expon;
}
```

다항식 덧셈 연산 #2 (3)

```
// C = A + B
poly_add2(int As, int Ae, int Bs, int Be, int *Cs, int *Ce)
{
      float tempcoef;
      *Cs = avail;
      while( As <= Ae && Bs <= Be )
       switch(compare(terms[As].expon,terms[Bs].expon)){
       case '>':        // A의 차수 > B의 차수
                attach(terms[As].coef, terms[As].expon);
                As++;                    break;
       case '=':        // A의 차수 == B의 차수
                tempcoef = terms[As].coef + terms[Bs].coef;
                if( tempcoef )
                 attach(tempcoef,terms[As].expon);
                As++; Bs++;              break;
       case '<':        // A의 차수 < B의 차수
                attach(terms[Bs].coef, terms[Bs].expon);
                Bs++;                    break;
      }
```

다항식 덧셈 연산 #2 (4)

```
// A의 나머지 항들을 이동함
    for(;As<=Ae;As++)
        attach(terms[As].coef, terms[As].expon);
    // B의 나머지 항들을 이동함
    for(;Bs<=Be;Bs++)
        attach(terms[Bs].coef, terms[Bs].expon);
    *Ce = avail -1;
}
//
void main()
{
    int Cs, Ce;
    poly_add2(0,2,3,5,&Cs,&Ce);
}
```

희소 행렬

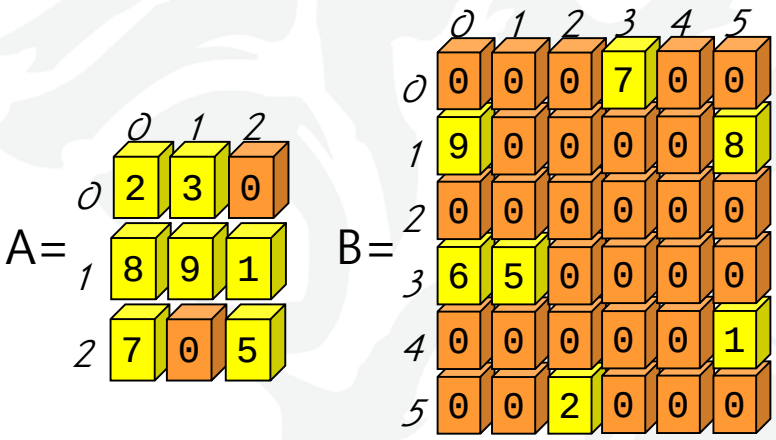
- 배열을 이용하여 행렬(matrix)을 표현하는 2가지 방법
  - (1) 2차원 배열을 이용하여 배열의 전체 요소를 저장하는 방법
  - (2) 0이 아닌 요소들만 저장하는 방법
- 희소행렬: 대부분의 항들이 0인 배열

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

희소 행렬 표현 방법 #1

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

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



희소 행렬 덧셈 연산 #1 (1)

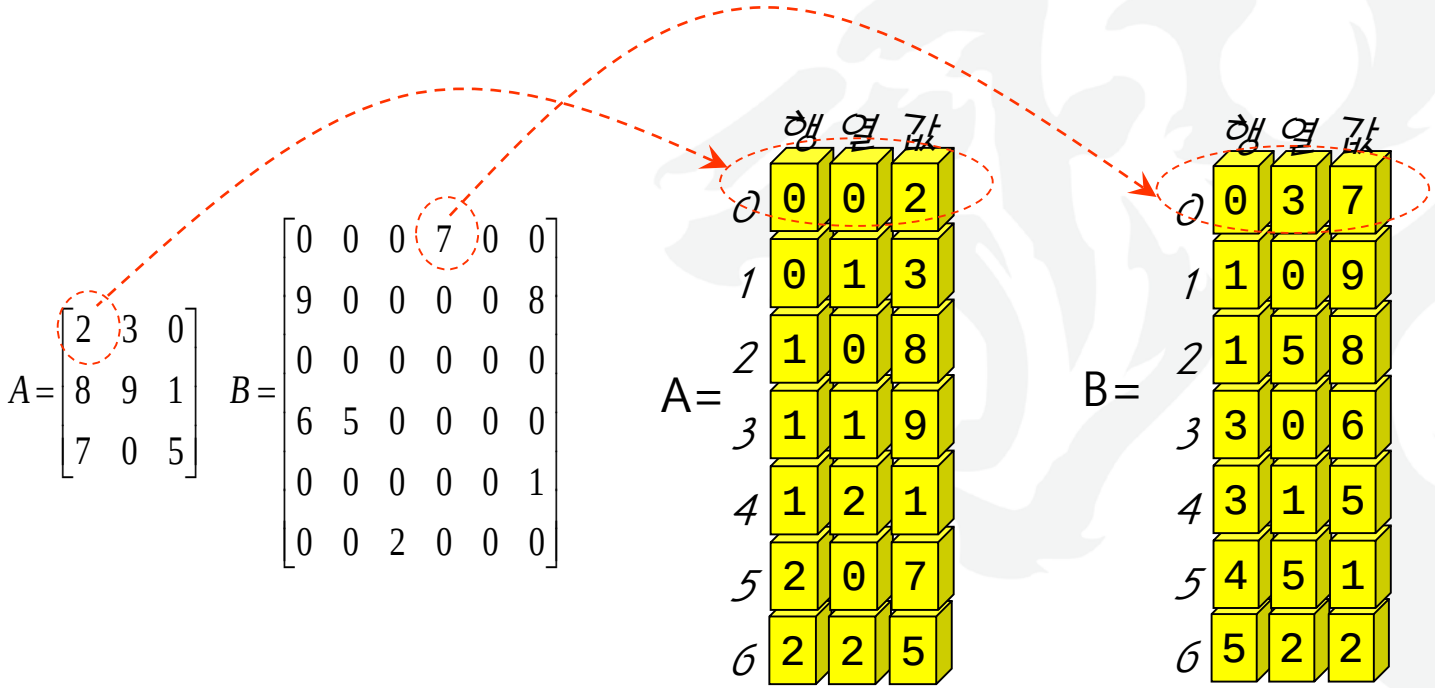
```
#include <stdio.h>
#define ROWS 3
#define COLS 3
// 희소 행렬 덧셈 함수
void sparse_matrix_add1(int A[ROWS][COLS],
        int B[ROWS][COLS], int C[ROWS][COLS]) // C=A+B
{
    int r,c;
    for(r=0;r<ROWS;r++)
        for(c=0;c<COLS;c++)
            C[r][c] = A[r][c] + B[r][c];
}
```

희소 행렬 덧셈 연산 #1 (2)

```
main()
{
    int array1[ROWS][COLS] = {    { 2,3,0 },
                                  { 8,9,1 },
                                  { 7,0,5 } };
    int array2[ROWS][COLS] = {    { 1,0,0 },
                                  { 1,0,0 },
                                  { 1,0,0 } };
    int array3[ROWS][COLS];
    sparse_matrix_add1(array1,array2,array3);
}
```

희소 행렬 표현 방법 #2

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



희소 행렬 덧셈 연산 #2 (1)

```
#define ROWS 3
#define COLS 3
#define MAX_TERMS 10
typedef struct {
    int row;
    int col;
    int value;
} element;
typedef struct SparseMatrix {
    element data[MAX_TERMS];
    int rows;    // 행의 개수
    int cols;    // 열의 개수
    int terms;   // 항의 개수
} SparseMatrix;
```

희소 행렬 덧셈 연산 #2 (2)

```
// 희소 행렬 덧셈 함수
// c = a + b
SparseMatrix sparse_matrix_add2(SparseMatrix a, SparseMatrix b)
{
    SparseMatrix c;
    int ca=0, cb=0, cc=0; // 각 배열의 항목을 가리키는 인덱스
    // 배열 a와 배열 b의 크기가 같은지를 확인
    if( a.rows != b.rows || a.cols != b.cols ){
        fprintf(stderr,"희소행렬 크기에러\n");
        exit(1);
    }
    c.rows = a.rows;
    c.cols = a.cols;
    c.terms = 0;
```

희소 행렬 덧셈 연산 #2 (3)

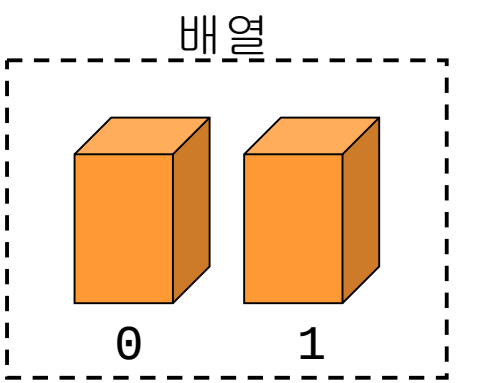
```
while( ca < a.terms && cb < b.terms ){
        // 각 항목의 순차적인 번호를 계산한다.
                int inda = a.data[ca].row * a.cols + a.data[ca].col;
                int indb = b.data[cb].row * b.cols + b.data[cb].col;
        if( inda < indb) {
                // a 배열 항목이 앞에 있으면
                c.data[cc++] = a.data[ca++];
        }
        else if( inda == indb ){
                // a와 b가 같은 위치
                if( (a.data[ca].value+b.data[cb].value)!=0){
                   c.data[cc].row = a.data[ca].row;
                   c.data[cc].col = a.data[ca].col;
                   c.data[cc++].value = a.data[ca++].value +
                                b.data[cb++].value;
                }
                else {
                   ca++; cb++;
                }
        }
        else   // b 배열 항목이 앞에 있음
                c.data[cc++] = b.data[cb++];
    }
```

희소 행렬 덧셈 연산 #2 (4)

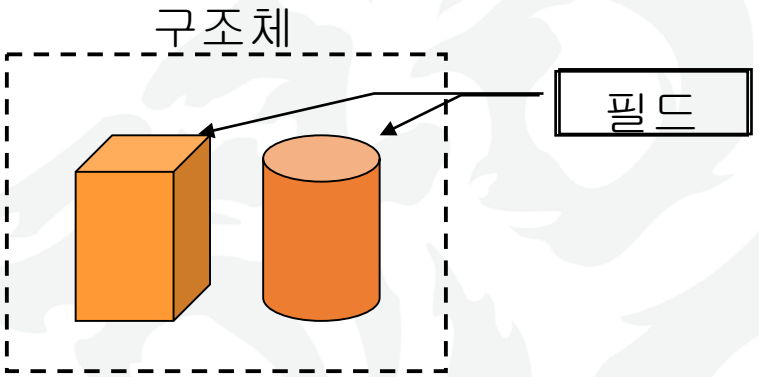
```
// 배열 a와 b에 남아 있는 항들을 배열 c로 옮긴다.
    for(; ca < a.terms; )
        c.data[cc++] = a.data[ca++];
    for(; cb < b.terms; )
        c.data[cc++] = b.data[cb++];
    c.terms = cc;
    return c;
}
// 주함수
main()
{
    SparseMatrix m1 = {     {{ 1,1,5 },{ 2,2,9 }}, 3,3,2 };
    SparseMatrix m2 = {     {{ 0,0,5 },{ 2,2,9 }}, 3,3,2 };
    SparseMatrix m3;
    m3 = sparse_matrix_add2(m1, m2);
}
```

구조체

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



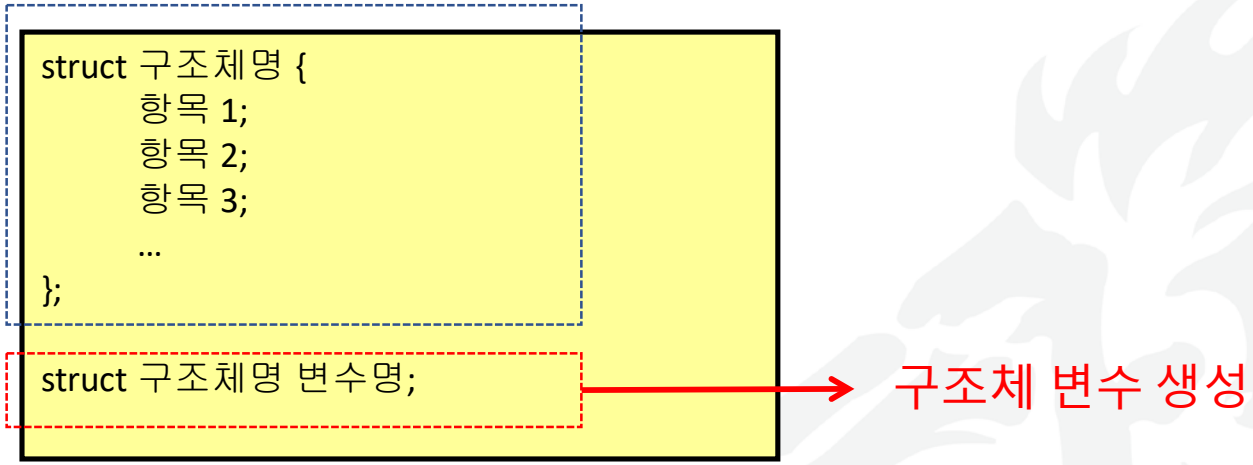

```
char carray[100];
```




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

구조체 형식

- 구조체 형식 정의



- typedef를 이용해 구조체를 만들면 기존 C의 자료형에는 없었던 새로운 타입으로 선언할 수 있음
  - 구조체 개념이 발전해 C++의 클래스(class)가 됨

구조체 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
```

- 항목연산자(membership operator): "."

```
   strcpy(a.name, "tom");
   a.age = 20;
   a.height = 180.5;
```

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
}
```

자체 참조 구조체

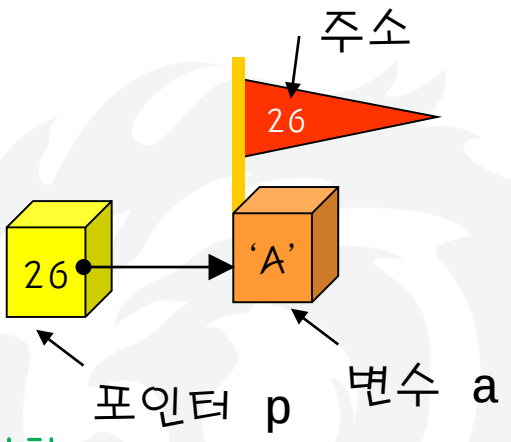
- 자체 참조 구조체(self-referential structure)
  - 항목 중에서 자기 자신을 가리키는 포인터가 한 개 이상 존재하는 구조체
  - 연결 리스트나 트리에 많이 등장

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

포인터

- 포인터 변수: 다른 변수의 주소를 가지고 있는 변수

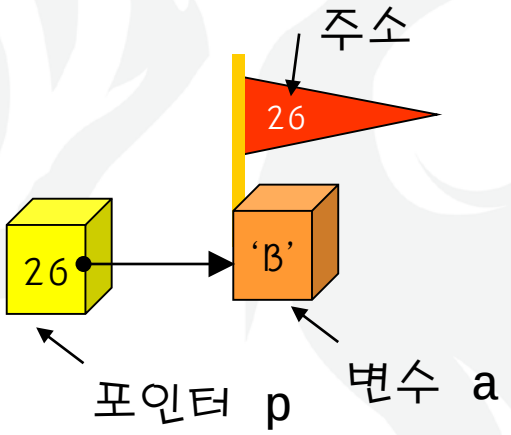
```
char a='A';
char *p;
p = &a;
```



- *p와 변수 a가 동일한 메모리 위치를 참조함
- *p와 변수 a는 동일한 객체를 가리키므로 전적으로 동일함
- *p값을 변경하면 변수 a값도 바뀜

- 포인터가 가리키는 내용의 변경: * 연산자 사용

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



다양한 포인터

- 다양한 포인터 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 int를 매개변수로 갖고 반환값을 갖지 않는 함수를 가리키는 포인터
```

- 포인터의 형변환: 필요할 때마다 형변환하는 것이 가능함

```
void *p;
pi=(int *) p; //p를 정수 포인터로 변경하여 pi로 대입
```

함수의 매개 변수로서의 포인터

- 함수안에서 매개변수로 전달된 포인터를 이용하여 외부 변수의 값 변경 가능

```
void swap(int *px, int *py)
{
        int tmp;
        tmp = *px;
        *px = *py;
        *py = tmp;
}
main()
{
        int a=1,b=2;
        printf("swap을 호출하기 전: a=%d, b=%d\n", a,b);
        swap(&a, &b);
        printf("swap을 호출한 다음: a=%d, b=%d\n", a,b);
}
```

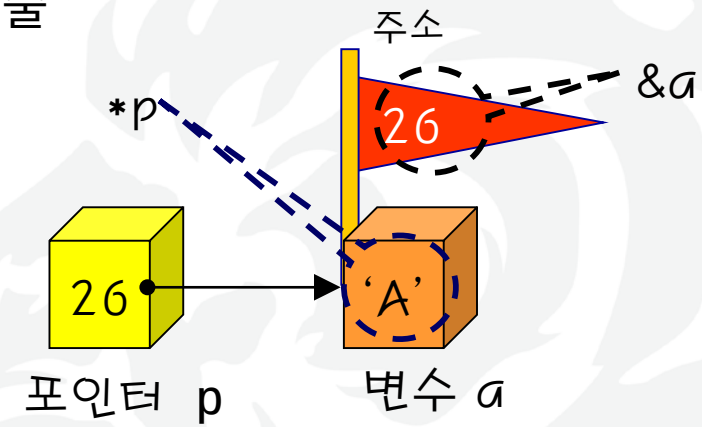
함수의 매개 변수로서의 포인터(계속)

- 만약 다음과 같다면?

```
void swap(int px, int py)
{
        int tmp;
        tmp = px;
        px = py;
        py = tmp;
}
main()
{
        int a=1,b=2;
        printf("swap을 호출하기 전: a=%d, b=%d\n", a,b);
        swap(a, b);
        printf("swap을 호출한 다음: a=%d, b=%d\n", a,b);
}
```

포인터 관련 연산

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

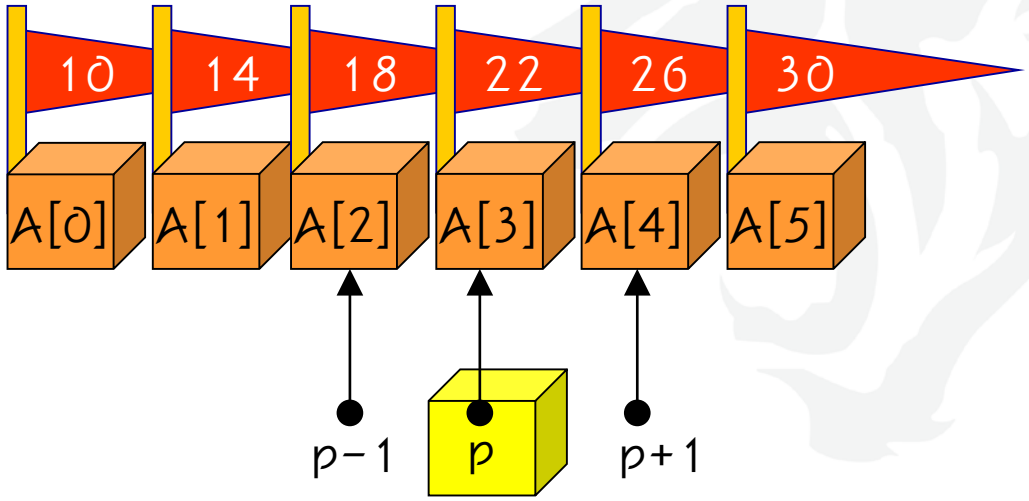



```
 p          // 포인터
*p          // 포인터가 가리키는 값
*p++        // 포인터가 가리키는 값을 가져온 다음, 포인터를 한칸 증가
*p--        // 포인터가 가리키는 값을 가져온 다음, 포인터를 한칸 감소
(*p)++      // 포인터가 가리키는 값을 증가
```

포인터 관련 연산(계속)

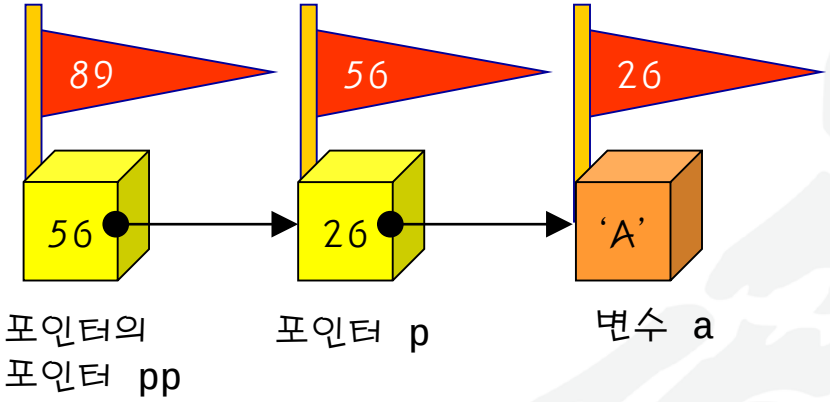
- 포인터에 대한 사칙연산
  - 포인터가 가리키는 객체단위로 계산됨

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



포인터의 포인터

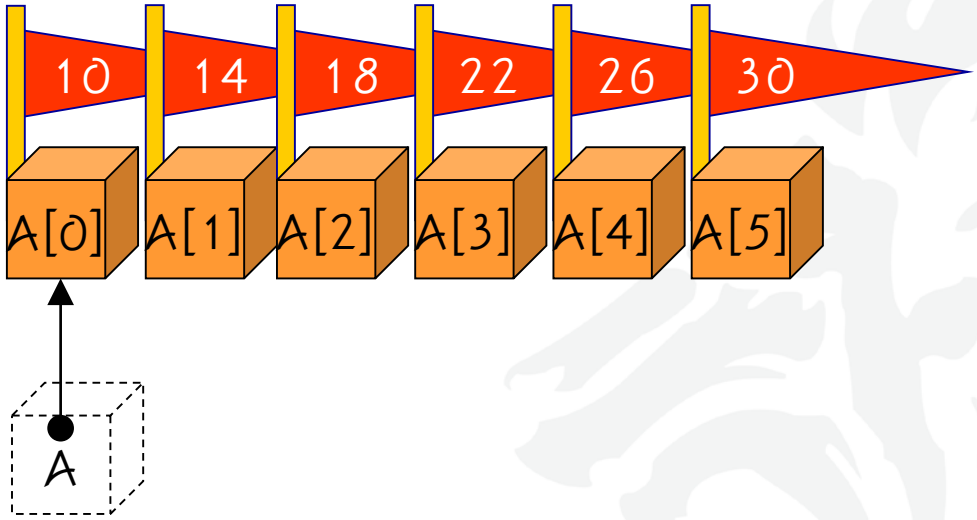
- 포인터도 하나의 변수이므로 포인터의 포인터 선언이 가능함




```
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
```

배열과 포인터

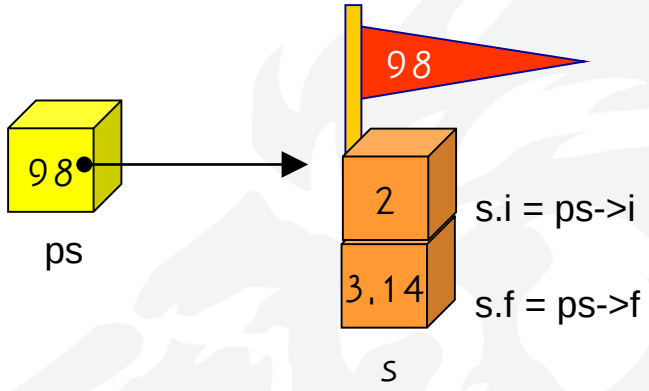
- 배열의 이름: 사실상의 포인터와 같은 역할



컴파일러가 배열의 이름을 배열의 첫 번째 주소로 대치

구조체와 포인터

- 구조체의 요소(멤버)에 접근하는 연산자: ->




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

ps->i 는 (*ps).i와 동일한 효과를 가짐

포인터 사용시 주의점

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

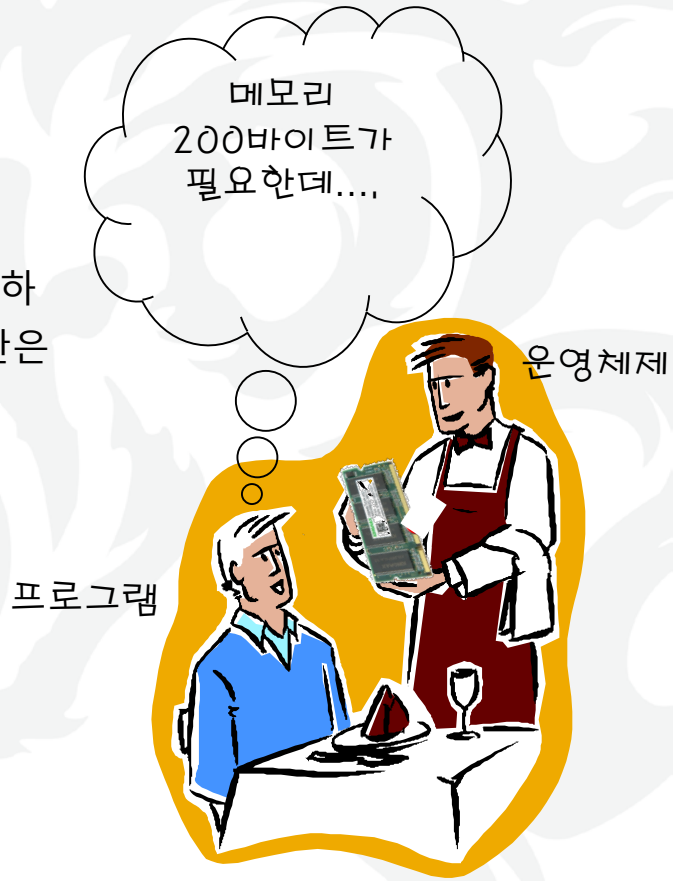
```
main()
{
        char *pc;     // 포인터 pi는 초기화가 안되어 있음
        *pc = 'E';    // 위험한 코드
}
```

- 포인터 타입간의 변환시에는 명시적인 타입 변환 사용

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

메모리 할당

- 프로그램이 메모리를 할당받는 방법
  - 정적 메모리
  - 동적 메모리 할당
- 정적 메모리 할당
  - 메모리의 크기는 프로그램이 시작하기 전에 결정
  - 프로그램의 수행 도중에 그 크기가 변경될 수는 없음
  - 만약 처음에 결정된 크기보다 더 큰 입력이 들어온다면 처리하지 못할 것이고 더 작은 입력이 들어온다면 남은 메모리 공간은 낭비될 것임
  - 예: 변수나 배열의 선언
    - int buffer[100]; char name[] = "data structure";
- 동적 메모리 할당
  - 프로그램의 실행 도중에 메모리를 할당 받는 것
  - 필요한 만큼만 할당을 받고 또 필요한 때에 사용하고 반납
  - 메모리를 매우 효율적으로 사용가능



동적 메모리 할당

- 동적 메모리 할당(dynamic memory allocation)
  - 프로그램이 실행 도중에 동적으로 메모리를 할당 받는 것
- 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 함수
  - malloc(size): 메모리 할당
  - free(ptr): 메모리 해제
  - sizeof(var): 변수나 타입의 크기 반환(바이트단위)

동적 메모리 할당 관련 함수

- malloc(int size): size만큼 메모리 블록 할당

```
(char *)malloc(100) ;    /* 100 바이트 할당 */
(int *)malloc(sizeof(int));/* 정수 1개를 저장할 메모리 확보*/
(struct Book *)malloc(sizeof(struct Book))/* Book 구조체 메모리 할당 */
```

- free(ptr): 메모리 해제: ptr이 가리키는 할당된 메모리 블록을 해제
- sizeof(var): 변수나 타입의 크기 반환(바이트단위)

```
size_t  i = sizeof( int );                          // 4
struct AlignDepends {
    char c;
    int i;
};
size_t size = sizeof(struct AlignDepends);          // 5? 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);
}
```

Week 3: Recursion, Array, Struct, Pointer