예제 1. 2차원 배열의 활용식

CODE XPERT
임베디드 전문가 그룹 철학

@ 2차원 배열의 활용식 : a[m][n] = *(*(a+m)+n) ➔ 교환법칙 성립

@ 다음 수식을 모두 * 연산식으로 바꾸고 다시 모두 [ ] 연산식으로 변환하라

- ◆ *a, **a, ***a
- ◆ a[0], a[0][0], a[1][2][3]
- ◆ *(*(*(a-1)+2)-3), *(a[1][2]+2)
- ◆ (*(a+2))[2], *(a+2)[2], *(*(a+1)[-2]+3)

@ 다음 코드의 인쇄 결과를 예측하라

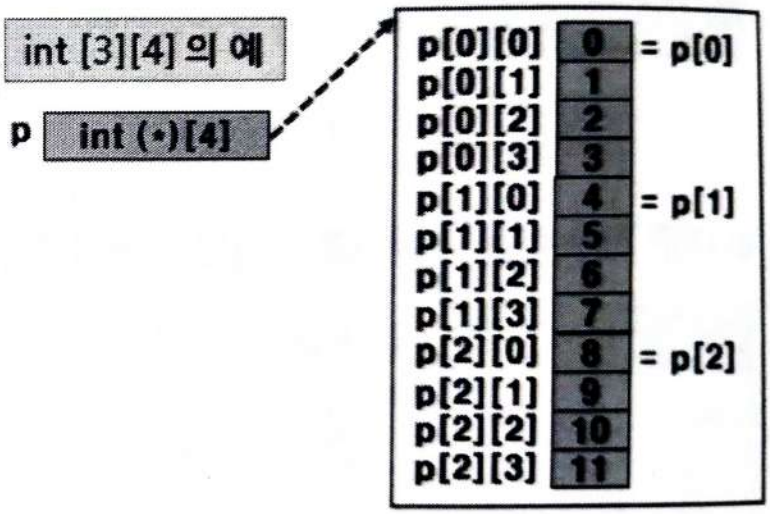
```
int a[3][4] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};

void main(void)
{
  printf("%d\n", a[2][3]);
  printf("%d\n", *(*(a+2)+3));
  printf("%d\n", (a+1)[1][3]);
  printf("%d\n", ((a+1)[1]+2)[1]);
  printf("%d\n", 3[a[2]]);
  printf("%d\n", 3[2[a]]);
}
```

예제 3. 등가포인터를 이용한 동적 배열

@ 2차원 배열의 동적 배열은 등가 포인터를 활용한다

- 한 번의 할당, 한번의 해제 그리고 딱 필요한 메모리만 할당 받는다
- 할당 받은 메모리가 선형적이므로 배열처럼 사용할 경우 문제가 발생하지 않는다

```
int (*p)[4];

x = 3;

p = malloc(x * sizeof(int [4]));

for(i=0; i<x; i++)
{
   for(j=0; j<4; j++)
   {
      p[i][j] = i*4+j;
   }
}

free(p);
```



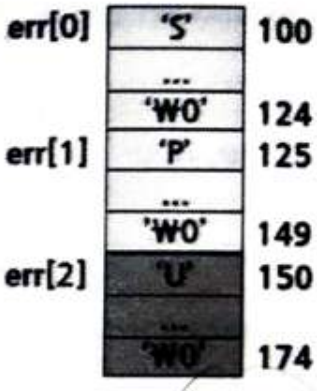
예제 5. 이차원 배열과 포인터 배열

CODE XPERT
임베디드 전문기 그룹 월택

@ 다음 string pool을 구현한 코드에서 속도, 메모리 측면에서 유리한 것은?

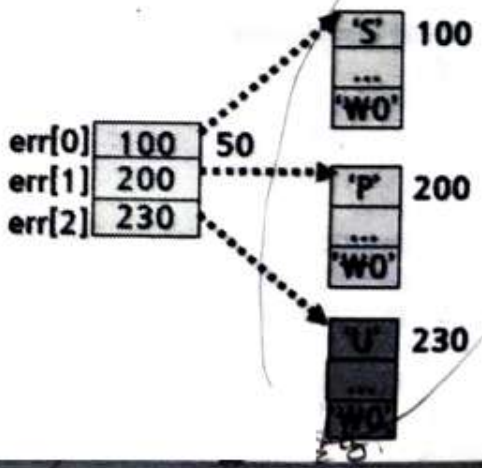
● 이차원 배열 : 배열 안에 문자열을 1차원 배열로 저장한다

①
```
char err[3][25] = {"Speed Error!",
                   "Position Error!",
                   "Unknown Command Error!"};

void main(void)
{
   printf("%s\n", err[1]);
}
```



● 포인터배열 : 배열 안에 문자열 시작주소를 저장한다

②
```
char *err[3] = {  "Speed Error!",
                  "Position Error!",
                  "Unknown Command Error!"};

void main(void)
{
   printf("%s\n",    err[1]);
}
```



배열의 등가포인터

CODE XPERT
임베디드 전문가 그룹 월택

@ 배열 등가포인터 : 배열을 대입할 때 경고가 발생하지 않는 포인터

- 따라서 배열과 동등하게 쓰일 수 있다 단, void * 는 등가 포인터가 될 수 없다
- 등가 포인터 만드는 방법 : 정의식 a == &a[0]에서 &a[0] 타입의 포인터를 만든다
  - 즉, 배열의 첫 요소 a[0]의 타입을 구하고 거기에 1순위 포인터를 추가하면 된다

배열	배열 첫 요소	배열명의 타입	등가 포인터
int a[4];	int	int *	int *p
int *a[4];	int *	int **	int **p
int a[2][3];	int [3]	int (*)[3]	int (*p)[3]

- 배열 등가포인터는 &, sizeof가 사용된 경우를 제외하고는 배열과 동일하게 쓰인다

```
int a[2][3];
int (*p)[3];
p=a;
```
➡
```
p[2][3] == a[2][3];
f(p) == f(a);
```
➡
```
&p != &a;
sizeof(p) != sizeof(a);
```

매번 이렇게 불편하게 등가포인터를 만들어야 하는가? 쉬운 방법은 없을까?

타입 분석 연습

CODEXPERT
임베디드 전문가 그룹 웹덱

@ 변수들의 modifier 우선 순위와 메모리 구조를 분석하고 질문에 답하라

```
int *p;

int a[4];

char **p;

int *a[4];

int (*p)[4]
```

(1) *p의 타입, (2) p+1, (3) sizeof(a)

배열이면 반드시 포인터 타입을 구하고 포인터는 가리키는 것의 타입을 구한다

C에서 사용하는 모든 연산자의 우선순위는 다음과 같다

순위	Operator Type	종류	결합
1	Unary: Postfix	() [] -> . (postfix)++,--	L->R
2	Unary: Prefix	(prefix)++,-- (sign)+,- ! ~ sizeof & * (type cast)	L<-R
3	Binary: Multiply 계열	* / %	L->R
4	Binary: Add 계열	+ -	L->R
5	Binary: Shift	<< >>	L->R
6	Binary: Relational	< <= > >=	L->R
7	Binary: Equality	== !=	L->R
8	Binary: Bit AND	&	L->R
9	Binary: Bit X-OR	^	L->R
10	Binary: Bit OR	|	L->R
11	Binary: Logical AND	&&	L->R
12	Binary: Logical OR	||	L->R
13	Ternary: Conditional	? :	L<-R
14	Assignment	= += -= *= /= %= &= ^= |= <<= >>=	L<-R
15	Sequential Evaluation	,	L->R

연산자 우선순위 3법칙

CODE XPERT
임베디드 전문기 그룹 협력

◉ 연산자들은 다음과 같은 우선순위 규정에 따라서 계산된다

단항 우선

- 단항 연산자
- 이항 연산자
- 삼항 연산자
- 대입 연산자
- , 연산자

```
p = *a * *b;
p = a + ++b;
p=a, a=b, b=0;
```

후치 우선

- 후치 단항
- 전치 단항

```
p = *a[4];
p = ++*b();
p = (*a)[4];
```

근치 우선

- 가까운 단항
- 먼 단항

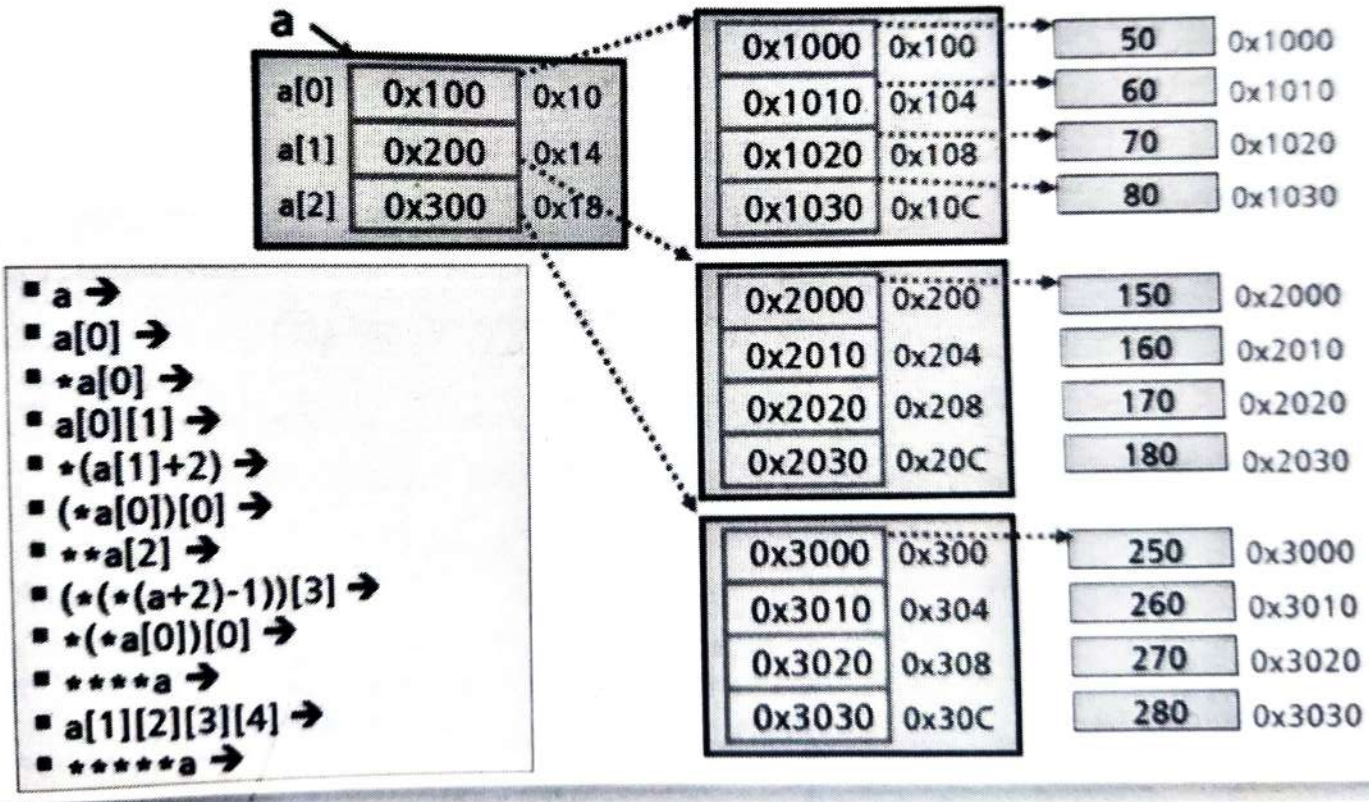
```
p = *--*++a;
p = a[3]++;
```

```
*--*b[3](3,4)
*--(*b)[3](3,4)
*--(*b[3])(3,4)
*--((*b)[3])(3,4)
```

int *(*a[3])[4]의 메모리 분석

@ a 배열에서 주소 겹 수와 예상되는 값을 구하라

● 단, 메모리에 없거나 알 수 없는 값일 경우 "0x1234 번지의 int" 처럼 구할 것



- a →
- a[0] →
- *a[0] →
- a[0][1] →
- *(a[1]+2) →
- (*a[0])[0] →
- **a[2] →
- (*(*(a+2)-1))[3] →
- *(*a[0])[0] →
- ****a →
- a[1][2][3][4] →
- *****a →

함수 등가포인터 선언과 실행

CODEXPERT
임베디드 전문가 그룹 웹택

함수 등가포인터 ➔ 함수 이름 대신 (*p)를 넣는다

- add 함수의 함수 등가포인터는 원론적으로 다음과 같이 사용한다

```
int (*p)(int, int);
p = &add;
(*p)(3,4);
```

⇨ int a = 10; int *p; 일 때 p = a; 가 아니라 p = &a;를 대입한다
함수 포인터도 함수 주소를 대입해야 옳으므로 p = &add가 옳다
포인터 깡통이론을 적용하면 p가 가리키는 함수는 (*p)가 된다
따라서 포인터 p가 가리키는 함수는 (*p)(3,4); 로 실행해야 한다

- 그러나 p = add; 는 add가 함수가 아니라 주소로 사용되는 경우이다
  - add는 양면성에 의하여 p = add하면 &add가 대입되고 대치법을 적용할 수가 있다

```
int (*p)(int, int);
p = add;
p(3,4);
```

⇨ int (*p)(int a, int b); 처럼 parameter 이름을 넣어도 무관하다
고로 parameter가 대입하는 함수 parameter와 달라도 무관하다
(*p)(3,4), (**p)(3,4) 모두 함수 주소 앞의 *는 가짜 ➔ p(3,4)

- 함수 등가포인터는 int (*p)( )와 같이 parameter를 미지정으로 하는 경우가 있다
  - 이 경우 p에는 parameter 무관하게 리턴이 int이기만 하면 어떤 함수든지 대입이 가능하다
  - 편리성과 활용도는 넓을 수 있으나 타입을 확정할 수 없으므로 실수할 우려가 있다

예제 5. 함수 Lookup table

CODE XPERT 임베디드 전문가 그룹 일택

@ switch에 의한 함수 호출 엔진을 개선한 다음 코드를 완성하라

● 배열에 호출할 함수들의 주소를 저장한 후 원하는 함수를 꺼내서 실행한다

```
int add(int a, int b)
{
  return a+b;
}

int sub(int a, int b)
{
  return a-b;
}

int mul(int a, int b)
{
  return a*b;
}

int get_key(void)
{
  return rand() % 3;
}
```

```
    fa[3]       = {add, sub, mul};

int op(int a, int b)
{
  return fa[get_key()](a,b);
}

void main(void)
{
  printf("%d\n", op(3, 4));
    ...

  printf("%d\n", op(3, 4));
}
```

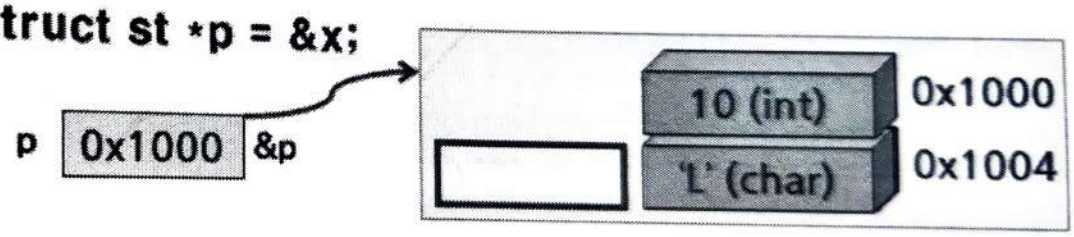
구조체 포인터

@ 구조체는 함수로 넘기면 call by value가 되어 속도, 메모리 측면에서 불리!

- 구조체는 구조체 자체보다 &x를 넘겨서 call by address 방식으로 전달한다
- 다음 구조체 포인터가 가리키는 구조체의 멤버 b를 'L'로 바꾸는 수식을 작성하라
  - -> 연산자를 사용하지 않는 방법으로 수식을 구하라
  - -> 연산자를 사용하여 수식을 구하라. -> 연산자도 역시 후치, 단항 1순위 연산자다

```
struct st
{
    int    a;
    char b;
}x={10,'k'};
```

```
struct st *p = &x;
```



CPP

```cpp
// C++ program to demonstrate that when vectors
// are passed to functions without &, a copy is
// created.
#include <bits/stdc++.h>
using namespace std;

// The vect here is a copy of vect in main()
void func(vector<int> vect) { vect.push_back(30); }

int main()
{
    vector<int> vect;
    vect.push_back(10);
    vect.push_back(20);

    func(vect);

    // vect remains unchanged after function
    // call
    for (int i = 0; i < vect.size(); i++)
        cout << vect[i] << " ";

    return 0;
}
```

Output

```
10 20
```