Pointers and Two-dimensional Arrays

- A two-dimensional array can be considered as a one-dimensional array of one-dimensional arrays. For example, when you make the following declaration:

```
int table[2][4];
```

  8 locations are reserved in the memory. The first 4 locations represent the first row, the second 4 represent the second row. Thus, `table` can be considered as a one-dim array of two elements, and each element of it is a one-dim array of 4 elements.

- As you know, the name of a one-dim array represents the address of its first element. When you think `table` as a one-dim array, its name represents the address of its first element, thus the address of the first one-dim array, thus the address of its first row.

```
table ≡ &table[0]
```

- Since `table[0]` is also representing a one-dim array, it refers to the address of its first element, thus the address of the first element of the first row in two-dim `table` array.

```
table[0] ≡ &table[0][0]
```

- What about `*table`? It refers to `*(&table[0])`, which is equivalent to `table[0]`.

```
table[0] ≡ &table[0][0] ≡ *table
```

- `table[1]` represents the second row, which is also a one-dim array, so it refers to the address of the first element of the second row of the two-dim `table` array.

```
table[1] ≡ &table[1][0]
```

- As you know, we can reach to the address of the second element of a one-dim array by adding **1** to its name. So, `table+1` represents `&table[1]`, thus `*(table+1)` represents `*(&table[1])`, which is equivalent to `table[1]`.

```
table[1] ≡ &table[1][0] ≡ *(table+1)
```

- `table[0]` represents the first row, hence the address of the element `table[0][0]`. What about `table[0]+1`? It represents the address of the element `table[0][1]`.

- How can we reach to `table[0][0]`?

```
table[0] ≡ &table[0][0] ≡ *table
table[0][0] ≡ *table[0] ≡  **table ( ≡ *(*(table+0)+0) )
```

- What about `table[1][0]`?

```
table[1] ≡ &table[1][0] ≡ *(table+1)
table[1][0] ≡ *table[1] ≡  **(table+1) ( ≡ *(*(table+1)+0))
```

- If the `table` array is as follows:

3	9	2	5
7	8	12	4

a) `table[0][1]`
b) `*(table[0]+1)`
c) `*(*(table+0)+1)`

all refers to the same value: **9**.

Example: Write expressions to refer 12

a) `table[1][2]`
b) `*(table[1]+2)`
c) `*(*(table+1)+2)`
d) `*(*table+6)`

- So, we can write it generally as

```
table[m][n] ≡ *(*(table+m)+n)

*(table+m)      =  address of table[m][0]
*(table+m)+n    =  address of table[m][n]
*(*(table+m)+n) =  content of table[m][n]
```

Example: Write a function `out_array` that displays the elements of a 2-dimensional array. Use pointer notation.

```
void out_array(int ar[][MAX], int row)
{    int k, j;
     for (k = 0; k < row; k++)
     {   for (j = 0; j < MAX; j++)
             printf("%3d", *(*(ar+k)+j));     // ar[k][j]
         printf("\n");
     }
}
```

- Since `ar` can also be considered as a one-dim array whose elements are one-dim arrays with `MAX` integers, and since a one-dim array can be represented as a pointer, we can also declare the `ar` array as `int(*ar)[MAX]` in the formal parameter list.

- The parantheses are important. If we omit them, `int *ar[MAX]` means an array of `MAX` elements whose each element is a pointer to an integer.