18ES611 Embedded System Programming

Sarath tv

One-Dimensional Arrays

int vector[5];

- The internal representation of an array has no information about the number of elements it contains.
- The array name simply references a block of memory. Using the sizeof operator with an array will return the number of bytes allocated to the array.
  - How to determine the number of elements????
  - printf("%d**\n**", **sizeof**(vector)/**sizeof**(**vector[0]**));

When a one-dimensional array is passed to a function, the array's address is passed by value. This makes the transfer of information more efficient since **we are not passing the entire array** and having to allocate **memory in** the **stack** for it.

Normally, this means the **array's size must be passed**.

from the function's perspective all it has, is the **address** of an **array** with no indication of its size

Using malloc to Create a One-Dimensional Array

*If we allocate memory from the heap and assign the address to a pointer, there is no reason we cannot use **array subscripts** with the **pointer** and **treat** this memory as an **array**.*

```
int *pv = (int*) malloc(5 * sizeof(int));
        for(int i=0; i<5; i++) {
             pv[i] = i+1;
         printf("%d \n",pv[i]);
                }
```

Instead of

```
for(int i=0; i<5; i++) {
     *(pv+i) = i+1;
           }
```

MD_4

`*(pv+i)` instead of `*pv+1`.

The dereference operator has higher precedence than the plus operator,

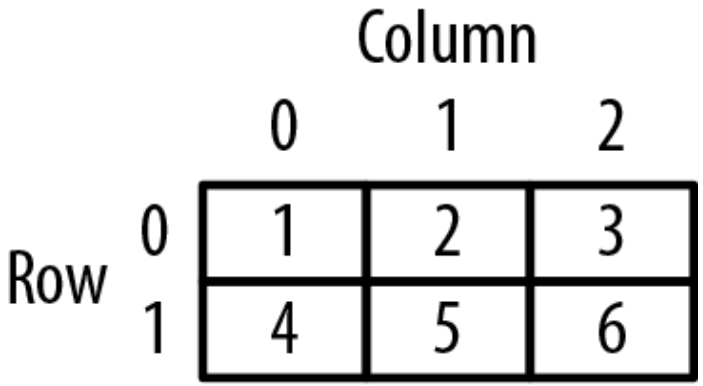
The second expression's pointer is dereferenced, giving us the value referenced by the pointer. We then add i to this integer value.

We need to force the addition to be performed first, followed by the dereference operation, in order for it to work correctly.

Two-Dimensional Arrays

- Two-dimensional arrays use **rows** and **columns** to identify array elements.
- This type of array needs to be **mapped** to the **one-dimension address space** of main memory.
- In C this is achieved by using a row-column ordering sequence. The **array's first row** is placed in memory **followed** by the **second row**, then the **third row**, and this ordering **continues** until the **last row** is placed in memory.
- **int** matrix[2][3] = {{1,2,3},{4,5,6}};

matrix[0][0] 100	1
matrix[0][1] 104	2
matrix[0][2] 108	3
matrix[1][0] 112	4
matrix[1][1] 116	5
matrix[1][2] 120	6



- A Two-dimensional array is treated as an ***array of arrays***.

- **int** matrix[2][3] = {{1,2,3},{4,5,6}};

```
for (int i = 0; i < 2; i++) {
    printf("&matrix[%d]: %p sizeof(matrix[%d]): %d\n", i,
           &matrix[i], i, sizeof(matrix[i]));
}
```

- when we access the *array using only **one subscript***, we get *a **pointer to** the corresponding **row***.

- The following output **assumes** the array is located at address 100. The size is 12 because each row has three elements of four bytes each:

```
&matrix[0]: 100 sizeof(matrix[0]): 12
&matrix[1]: 112 sizeof(matrix[1]): 12
```

*****NOTE- We used array notations only for this*****

#ifdef MD_1

int arr[5];

Array of integers-each mem block will contain an integer

float arr[5];

Array of float-each mem block will contain an **float**

char arr[5];

Array of char -each mem block will contain an char

int* arr[5];

Array of pointer to integers-each mem block will contain an pointer to int

Using a One-Dimensional Array of Pointers

The following sequence declares an array of integer pointers, allocates memory for each element, and initializes this memory to the array's index.

```
int* arr[5];
for(int i=0; i<5; i++) {
arr[i] = (int*)malloc(sizeof(int));
*arr[i] = i;
}
```

We used arr[i] to reference the pointer and *arr[i] to assign a value to the location referenced by the pointer.

equivalent pointer notation for the loop's body:

```
*(arr+i) = (int*)malloc(sizeof(int));
**(arr+i) = i;
```

using two levels of indirection in the second type statement.

MD_5 MD_6

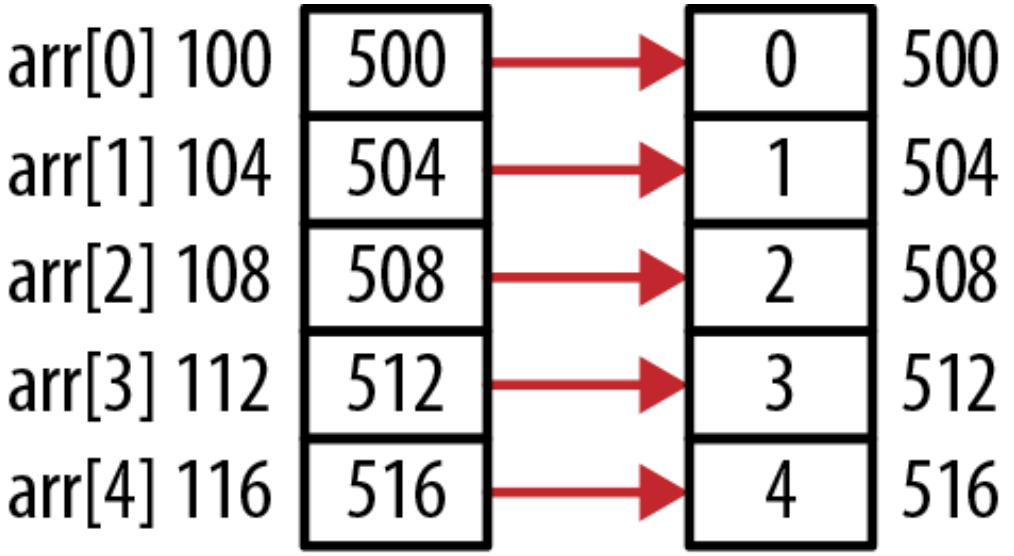
The subexpression **(arr+i)** represents the **address** of the **array's ith** element.

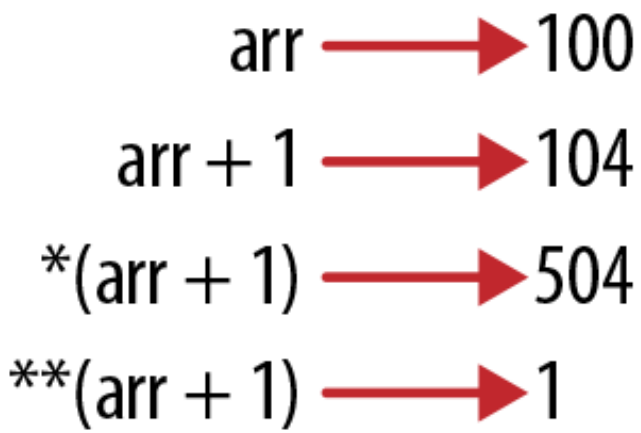
We need to **modify the content of this address** so we use the subexpression *(arr+i).

The allocated memory is assigned to this location in the first statement.

Dereferencing this subexpression a second time, as we do in the second statement, returns the allocated memory's location.

We then assign the variable i to it.





Pointers to array[first element/full array]

```
#include<stdio.h>

int main()
{
int arr[5] = { 1, 2, 3, 4, 5 };
int *ptr = arr;

printf("%p\n", ptr);
return 0;
}
```

In this program, we have a pointer **ptr** that points to the 0th element of the **array**.

Similarly, we can also declare a pointer that can point to whole array instead of only one element of the array.

data_type (*var_name)[size_of_array];

int (*ptr)[10];

Here **ptr** is pointer that can point to an array of 10 integers.

The pointer that points to the 0th element of array and the pointer that points to the whole array are totally different.

```
// C program to understand difference between
// pointer to an integer and pointer to an
// array of integers.
#include<stdio.h>

int main()
{
        // Pointer to an integer
        int *p;

        // Pointer to an array of 5 integers
        int (*ptr)[5];
        int arr[5];
        p = arr;
        ptr = &arr;

        printf("p = %p, ptr = %p\n", p, ptr);

        p++;
        ptr++;

        printf("p = %p, ptr = %p\n", p, ptr);

        return 0;
}
```

// C program to illustrate sizes of pointer of array

size_p2arr

```
#include<stdio.h>
int main()
{
      int arr[] = { 3, 5, 6, 7, 9 };
      int *p = arr;
      int (*ptr)[5] = &arr;

      printf("p = %p, ptr = %p\n", p, ptr);
      printf("*p = %d, *ptr = %p\n", *p, *ptr);

      printf("sizeof(p) = %d, sizeof(*p) = %d\n", sizeof(p), sizeof(*p));
      printf("sizeof(ptr) = %d, sizeof(*ptr) = %d\n", sizeof(ptr),sizeof(*ptr));
      return 0;
}
```

//size_p2arr

Pointers and Multidimensional Arrays

- Parts of **Multidimensional Arrays** Can Be Treated As **Subarrays**.
- Each **row** of a **two-dimensional** array can be treated **as a one-dimensional array**.
- This behavior affects how we use pointers when dealing with multidimensional arrays.

- Create a two-dimensional array

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

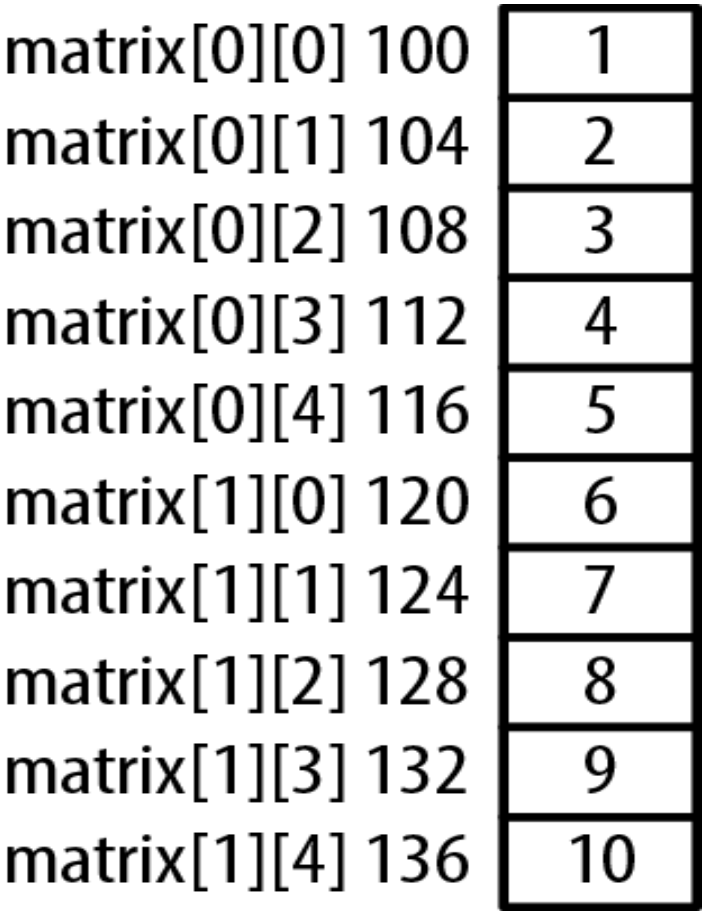
- //Print Addresses and their corresponding values

```
for(int i=0; i<2; i++) {
for(int j=0; j<5; j++) {
printf("matrix[%d][%d] Address: %p Value: %d\n",
i, j, &matrix[i][j], matrix[i][j]);
}}
```

MD_2

```
matrix[0][0]  Address: 100  Value: 1
matrix[0][1]  Address: 104  Value: 2
matrix[0][2]  Address: 108  Value: 3
matrix[0][3]  Address: 112  Value: 4
matrix[0][4]  Address: 116  Value: 5
matrix[1][0]  Address: 120  Value: 6
matrix[1][1]  Address: 124  Value: 7
matrix[1][2]  Address: 128  Value: 8
matrix[1][3]  Address: 132  Value: 9
matrix[1][4]  Address: 136  Value: 10
```

Two-dimensional array memory allocation



Relate a 2D array to a pointer to full array

Create a matrix- 4x3

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

Print address of matrix

```
printf("%d\n",&matrix);
```

Print using just array name

```
printf("%d\n",matrix);
```

Create a pointer to array and initialize to the martix

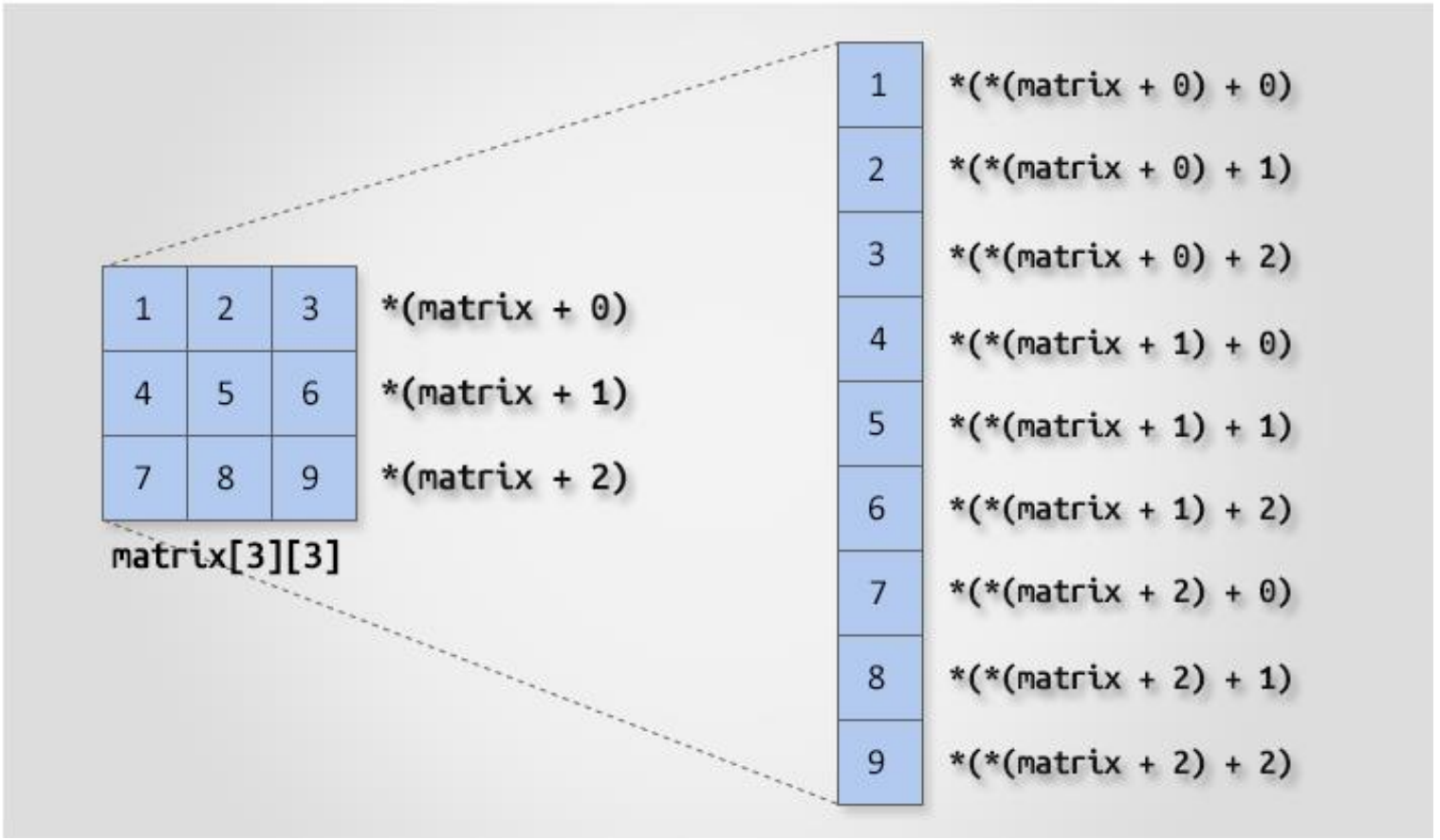
```
int (*ptr)[3]=&matrix;
```

Print the address of each matrix element.

Using & operation and pointer notation

MD_9

Two dimensional array access using pointer in image.



Pointers and Multi dimensional Array

- Suppose **x** is a one-dimensional, 10-element array of integers. It is possible to define **x as** a pointer variable rather than an array. Thus, we can write

```
int *x;
```

rather than

```
int  x[10] ;
```

- x is *not automatically assigned a memory* block when it is defined as a pointer variable, though a block of memory large enough to store **10** integer quantities will be reserved in advance when **x** is defined **as** an array.
- To assign sufficient memory for **x,** we can make use of the library function **malloc,** as follows.
  - `x = (int *) malloc(10 * sizeof(int));`

POINTERS AND MULTIDIMENSIONAL ARRAYS

- Since a one-dimensional array can be represented in terms of a pointer (the array name) and an offset (the subscript),
- A Multidimensional array can also be represented with **an** equivalent pointer notation. we can define a two-dimensional array as a pointer to a group of contiguous one-dimensional arrays.
  - **`data- type (*ptvar) [expression 2];`**

  Instead of
  - *data- type array_name[expression l]* [*expression* 2];
- This concept can be generalized to higher-dimensional arrays; that is,
  - **`data- type (*ptvar)[expression 2]. . [ expression n];`**

  replaces
  - data- type *array_name* [expression 1][expression 21 . . [expression n] ;

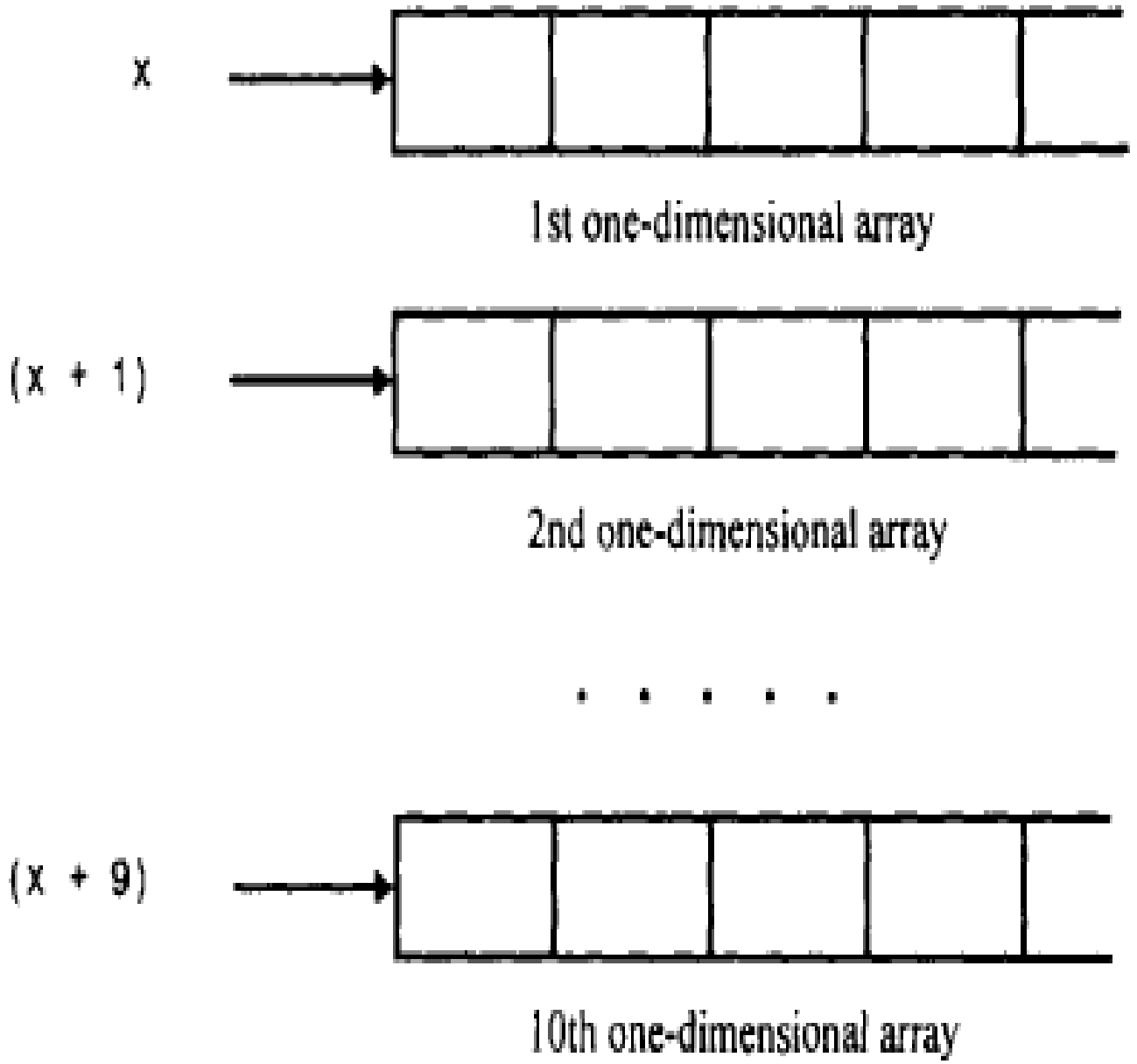
Suppose **x** is a two-dimensional integer array having 10 rows and 20 columns.
We can declare **x** as

```
int (*x)[20];
```

rather than

```
int x[10][20];
```

In the first declaration, **x** is defined to be a pointer to a group of contiguous, one-dimensional, 20-element integer arrays.
Thus, **x** points to the first 20-element array, which is actually the first row (i.e., row 0) of the original two-dimensional array.
Similarly, **(x + 1)** points to the second 20-element array, which is the second row (row 1) of the original two dimensional array, and so on,



- Suppose **x** is a two-dimensional integer array having 10 rows and **20** columns, **as** declared in the previous example. The item in row **2,** column ***5*** can be accessed by writing either
  - `* ( * ( x + 2) + 5)`

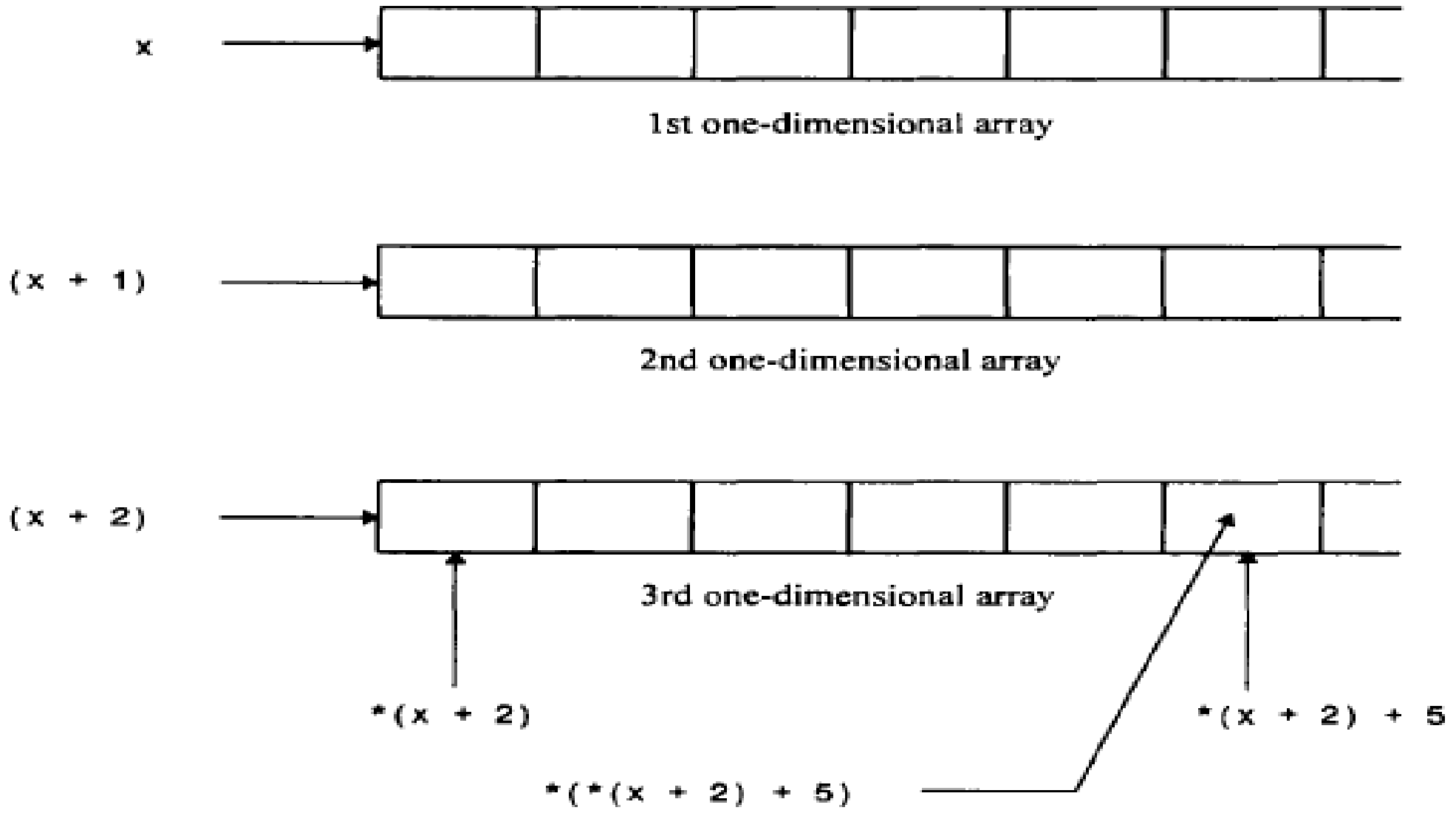
  or
  - `X[2][5]`

First, note that `(x + 2)` is a pointer to row **2.** Therefore the object of this pointer,

- `(x+2)`, refers to the entire row. Since row **2** is a one-dimensional array, `*(x+2)` is actually a pointer to the first element in row **2.**
- We now add ***5*** to this pointer. Hence, `(* (x + 2) + 5)` is a pointer to element **5** (i.e., the sixth

element) in row **2.**

The object of this pointer, `* ( * (x + 2) + 5)`, therefore refers to the item in column ***5*** of row **2,** which is `x [ 2][5]`.

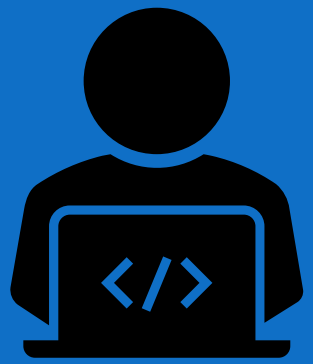

x
1st one-dimensional array
(x + 1)
2nd one-dimensional array
(x + 2)
3rd one-dimensional array
*(x + 2)
*(x + 2) + 5
((x + 2) + 5)

Dynamically Allocating a Two-Dimensional Array

- `int *matrix = (int *)malloc(rows * columns * sizeof(int));`

Note

- **Array and Pointers – how they can be used.**
- **2D array can be seen through**
  - Array of pointer concept .
  - Pointer to whole array concept.



THANK YOU!!!!!