Computer Programming & Problem Solving

CS100

Mrs Sanga G. Chaki
Department of Computer Science and Engineering
National Institute of Technology, Goa

January, 2023

Topics

1. Two Dimensional Arrays

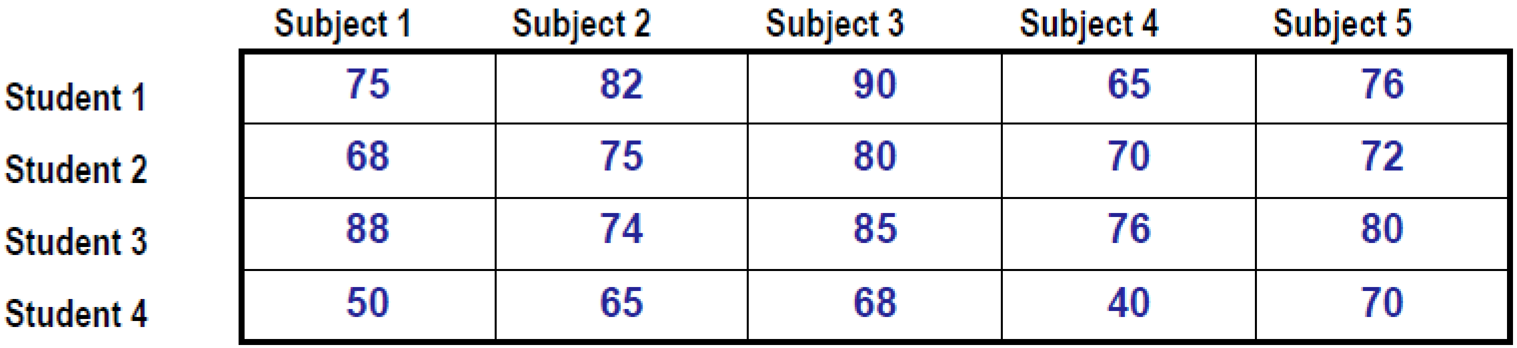
2. Pointers and 2D Arrays

    1. Pointer to array and array of pointers

3. 3D arrays

2-D Arrays/Matrix

1. We have seen that an array variable can store a list of values.
2. Many applications require us to store a *table* of values.



	Subject 1	Subject 2	Subject 3	Subject 4	Subject 5
Student 1	75	82	90	65	76
Student 2	68	75	80	70	72
Student 3	88	74	85	76	80
Student 4	50	65	68	40	70

1. The table can be regarded as a matrix consisting of 4 rows and 5 columns.
2. C allows us to define such tables of items by using two-dimensional arrays.

Declaring 2-D Arrays

General form :

```
type  array_name [row_size][column_size];
```

Examples:

```
int marks[4][5];

float sales[12][25];
```

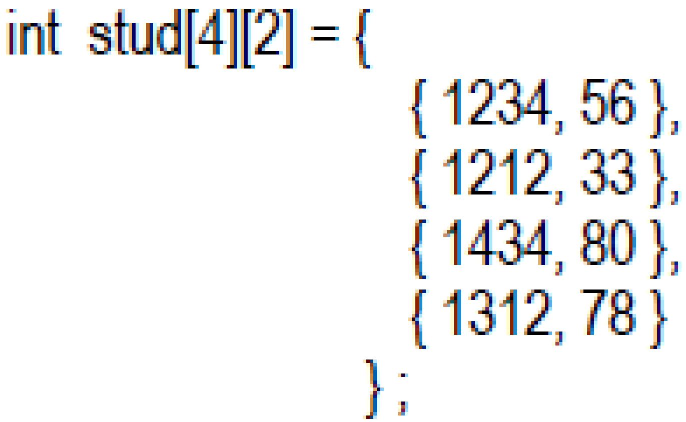
Accessing 2-D Array Elements

1. Similar to that for 1-D array, but use two indices.
   a) First indicates row, second indicates column.
   b) Both the indices should be expressions which evaluate to integer values.

   Example:

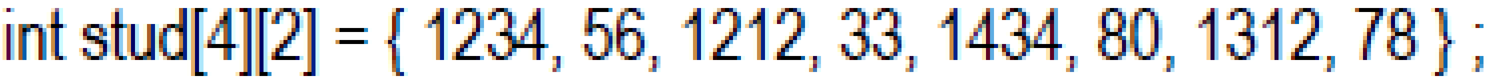
```
int x[20][100];

x[0][0] = 36;

x[4][9] = 10;

x[19][99] = 91;
```

Initializing 2-D Arrays



```
int  stud[4][2] = {
                    { 1234, 56 },
                    { 1212, 33 },
                    { 1434, 80 },
                    { 1312, 78 }
                 } ;
```

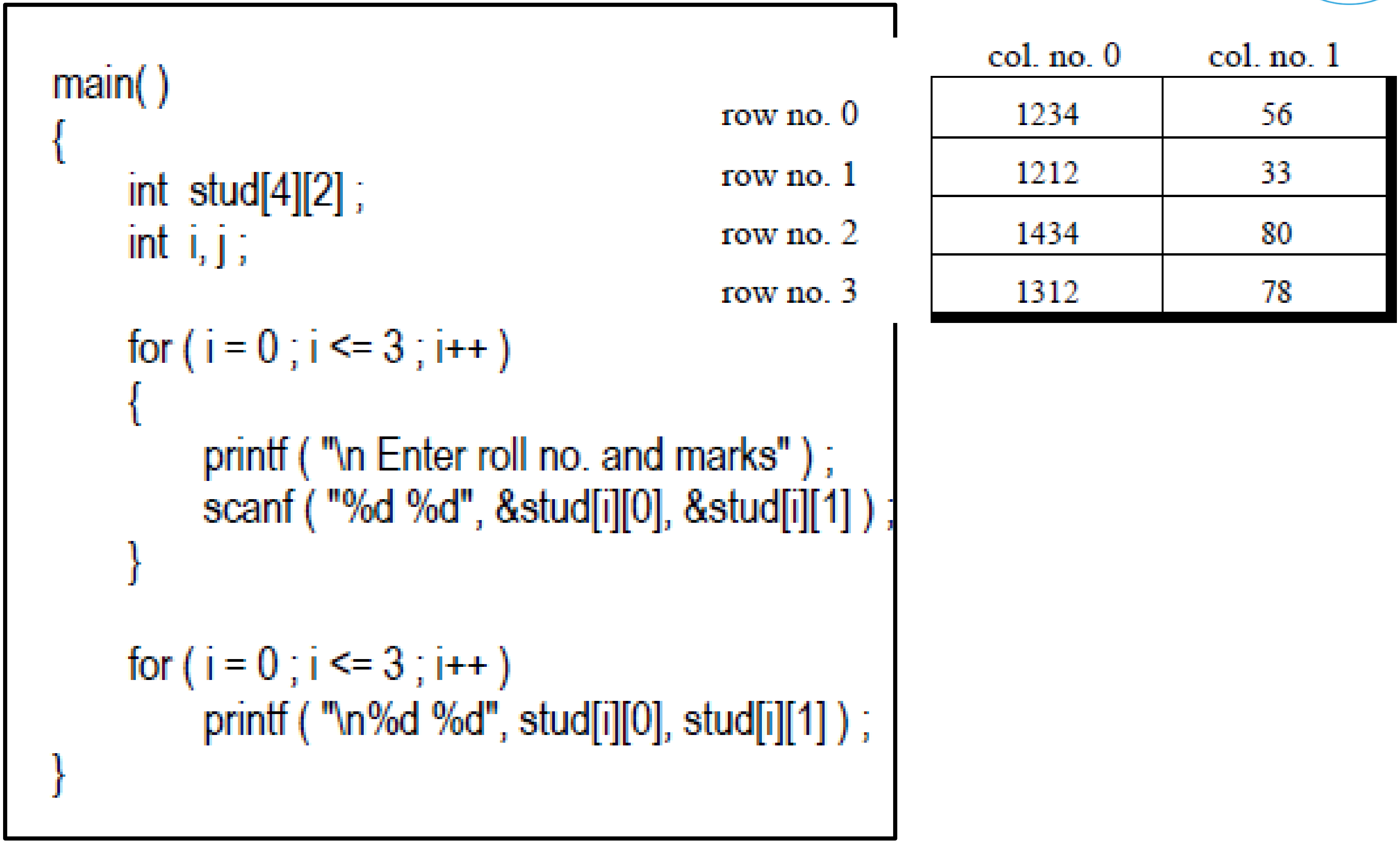
OR



```
int stud[4][2] = { 1234, 56, 1212, 33, 1434, 80, 1312, 78 } ;
```

It is necessary to mention the second (column) dimension, whereas the first dimension (row) is optional – when initializing a 2D array.

What happens here?



```
main( )
{
    int  stud[4][2] ;
    int  i, j ;

    for ( i = 0 ; i <= 3 ; i++ )
    {
        printf ( "\n Enter roll no. and marks" ) ;
        scanf ( "%d %d", &stud[i][0], &stud[i][1] ) ;
    }

    for ( i = 0 ; i <= 3 ; i++ )
        printf ( "\n%d %d", stud[i][0], stud[i][1] ) ;
}
```

	col. no. 0	col. no. 1
row no. 0	1234	56
row no. 1	1212	33
row no. 2	1434	80
row no. 3	1312	78

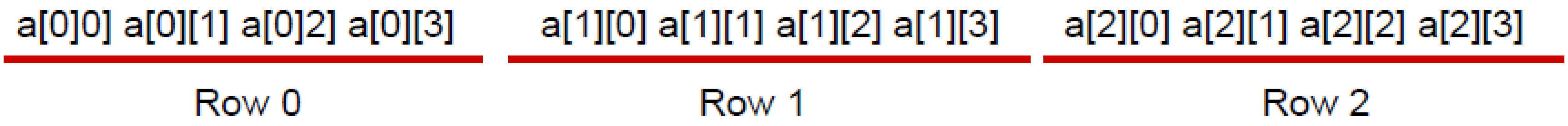
Memory Map of 2D Array

1. **Memory doesn’t contain rows and columns.**
2. **In memory whether it is a one-dimensional or a two-dimensional array the array elements are stored in one continuous chain**

s[0][0]	s[0][1]	s[1][0]	s[1][1]	s[2][0]	s[2][1]	s[3][0]	s[3][1]
1234	56	1212	33	1434	80	1312	78
65508	65510	65512	65514	65516	65518	65520	65522

Memory Map of 2D Array

1. **Starting from a given memory location, the elements are stored row-wise in consecutive memory locations**



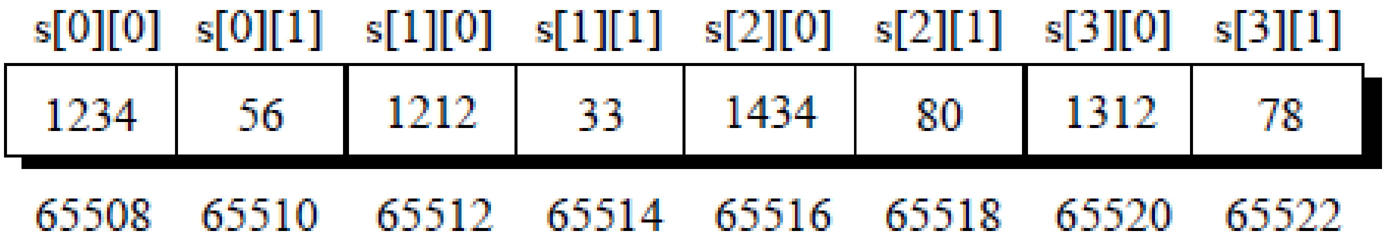
a[0]0] a[0][1] a[0]2] a[0][3] a[1][0] a[1][1] a[1][2] a[1][3] a[2][0] a[2][1] a[2][2] a[2][3]

Row 0 Row 1 Row 2

Memory Map of 2D Array

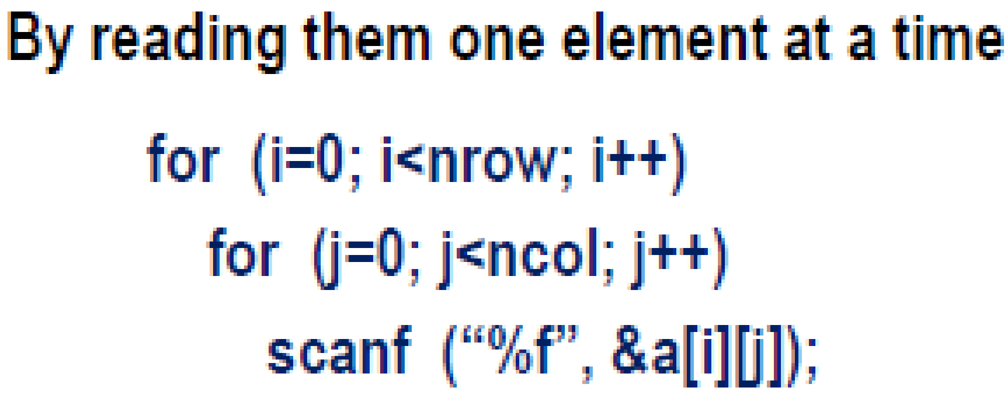
- x: starting address of the array in memory
- c: number of columns
- k: number of bytes allocated per array element

- a[i][j] is allocated memory location at address $x + (i * c + j) * k$



s[0][0]	s[0][1]	s[1][0]	s[1][1]	s[2][0]	s[2][1]	s[3][0]	s[3][1]
1234	56	1212	33	1434	80	1312	78
65508	65510	65512	65514	65516	65518	65520	65522

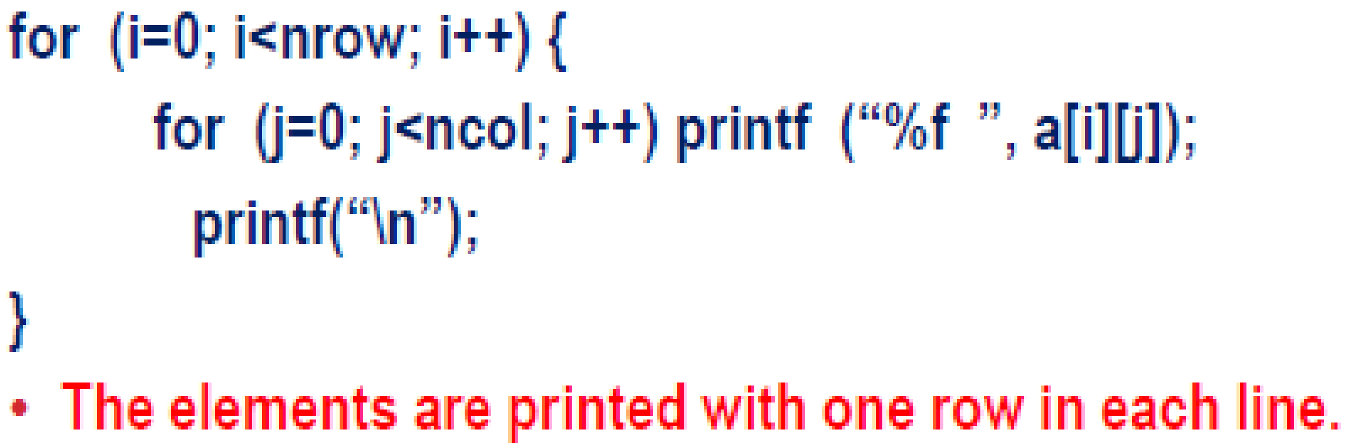
Reading Elements into 2D Array



By reading them one element at a time

```
for  (i=0; i<nrow; i++)
   for  (j=0; j<ncol; j++)
      scanf  (“%f”, &a[i][j]);
```

Printing Elements from 2D Array



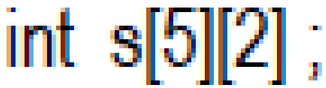
```
for  (i=0; i<nrow; i++) {
        for  (j=0; j<ncol; j++) printf  (“%f  ”, a[i][j]);
          printf(“\n”);
}
```

- The elements are printed with one row in each line.

Pointers and 2-D Arrays

Pointers and 2-D Arrays

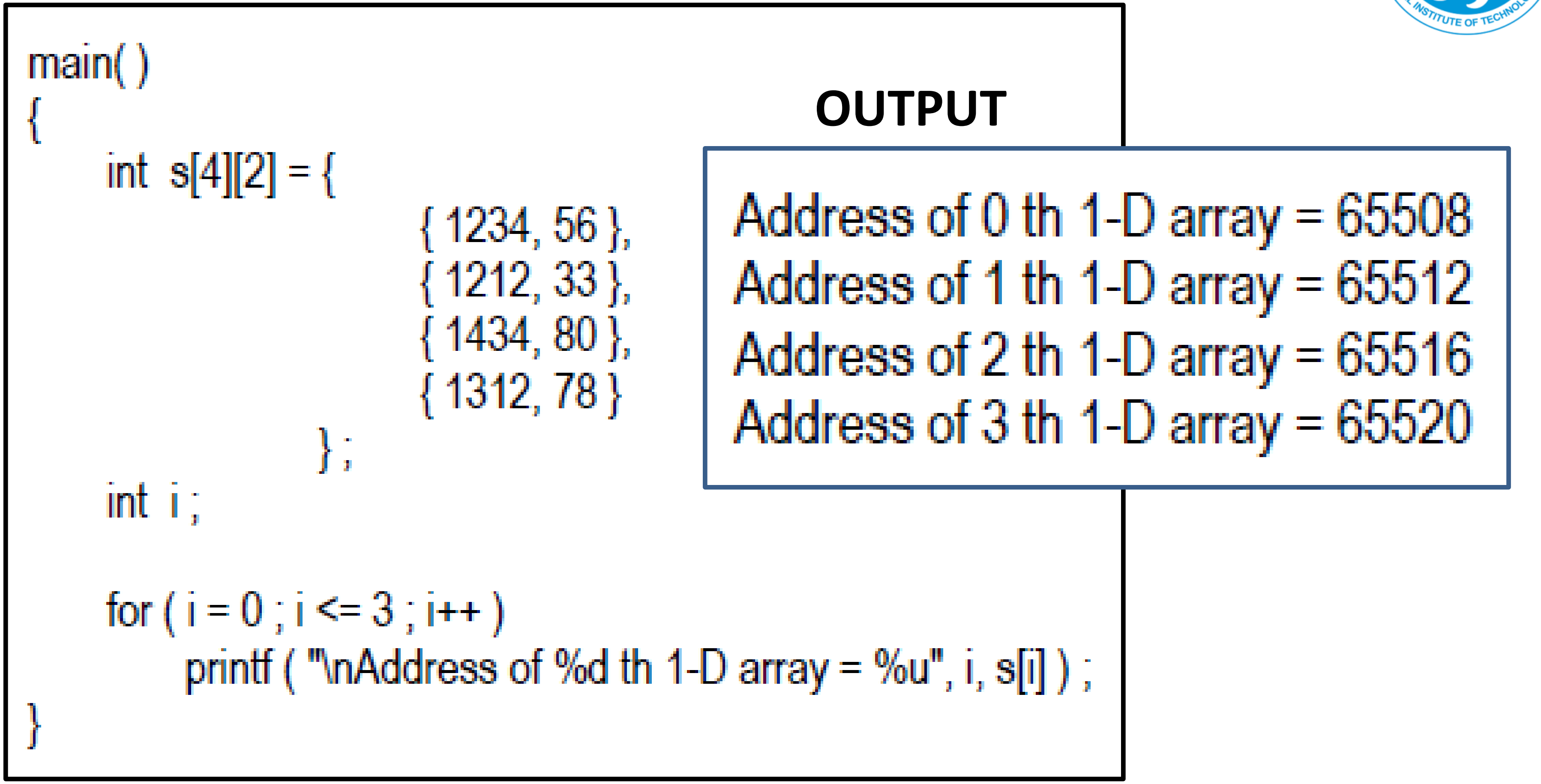
1. C language treats parts of arrays as arrays
2. Each row of a two-dimensional array can be thought of as a one-dimensional array.
3. This is utilized to access array elements of a two-dimensional array using pointers.



```
int  s[5][2] ;
```

1. This can be thought of as an array of 5 elements, each of which is a 1D array containing 2 integers

How to access 2-D Arrays using Pointers

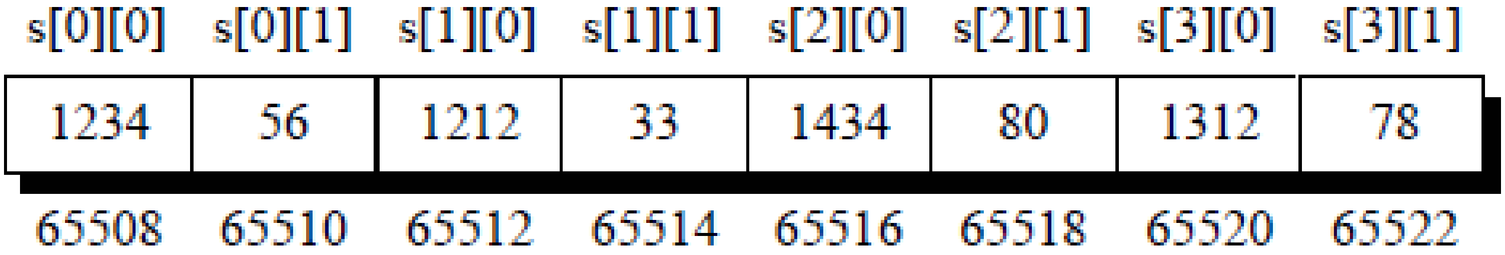


```
main( )
{
    int  s[4][2] = {
                        { 1234, 56 },
                        { 1212, 33 },
                        { 1434, 80 },
                        { 1312, 78 }
                    } ;
    int  i ;

    for ( i = 0 ; i <= 3 ; i++ )
        printf ( "\nAddress of %d th 1-D array = %u", i, s[i] ) ;
}
```

OUTPUT

```
Address of 0 th 1-D array = 65508
Address of 1 th 1-D array = 65512
Address of 2 th 1-D array = 65516
Address of 3 th 1-D array = 65520
```

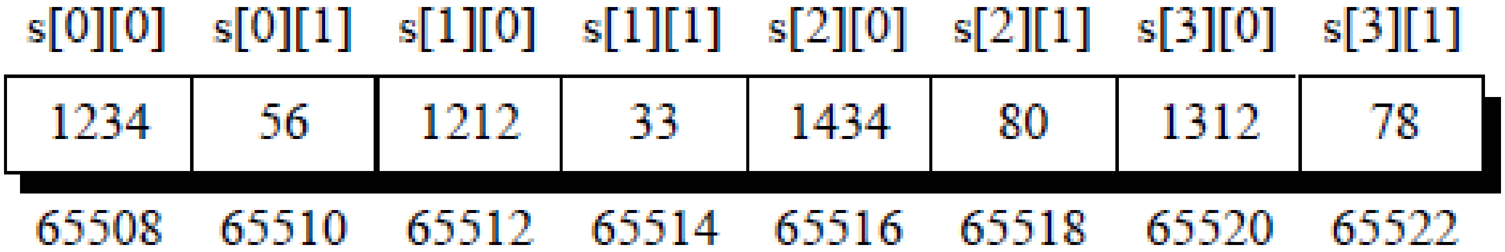
How to access 2-D Arrays using Pointers



s[0][0]	s[0][1]	s[1][0]	s[1][1]	s[2][0]	s[2][1]	s[3][0]	s[3][1]
1234	56	1212	33	1434	80	1312	78
65508	65510	65512	65514	65516	65518	65520	65522

1. s[0] gives the address of zeroth 1-D array
2. s[1] gives the address of first 1-D array
3. So we are able to reach each one-dimensional array.
4. What remains is to be able to refer to individual elements of a one-dimensional array

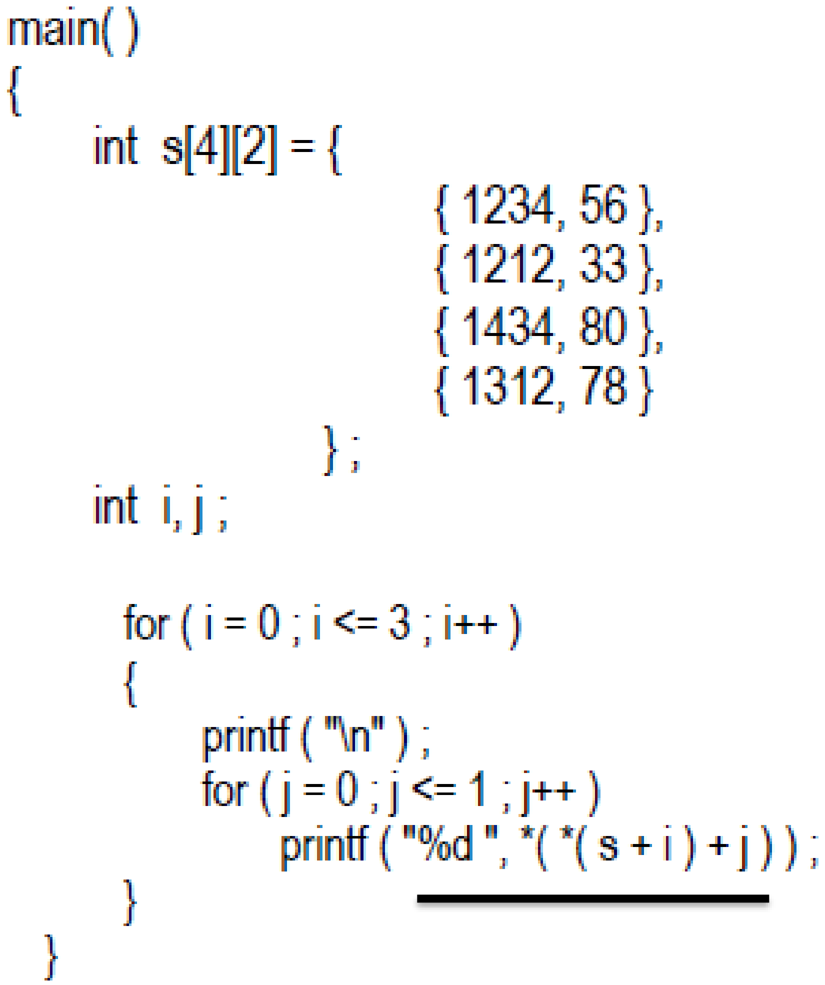
How to access 2-D Arrays using Pointers



s[0][0]	s[0][1]	s[1][0]	s[1][1]	s[2][0]	s[2][1]	s[3][0]	s[3][1]
1234	56	1212	33	1434	80	1312	78
65508	65510	65512	65514	65516	65518	65520	65522

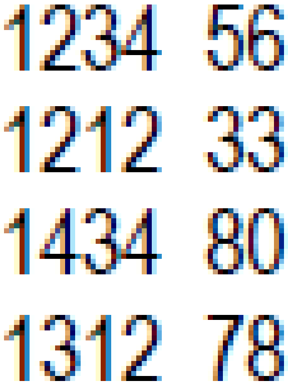
1. Say we want to reach s[2][1].
2. This is equivalent to
   a) * (s[2] + 1) or
   b) * (* (s + 2) + 1)
3. Hint: Remember how we access elements of 1-D array using pointers

How to access 2-D Arrays using Pointers



```
main( )
{
    int  s[4][2] = {
                        { 1234, 56 },
                        { 1212, 33 },
                        { 1434, 80 },
                        { 1312, 78 }
                   } ;
    int  i, j ;

     for ( i = 0 ; i <= 3 ; i++ )
     {
         printf ( "\n" ) ;
         for ( j = 0 ; j <= 1 ; j++ )
             printf ( "%d ", *( *( s + i ) + j ) ) ;
     }
}
```

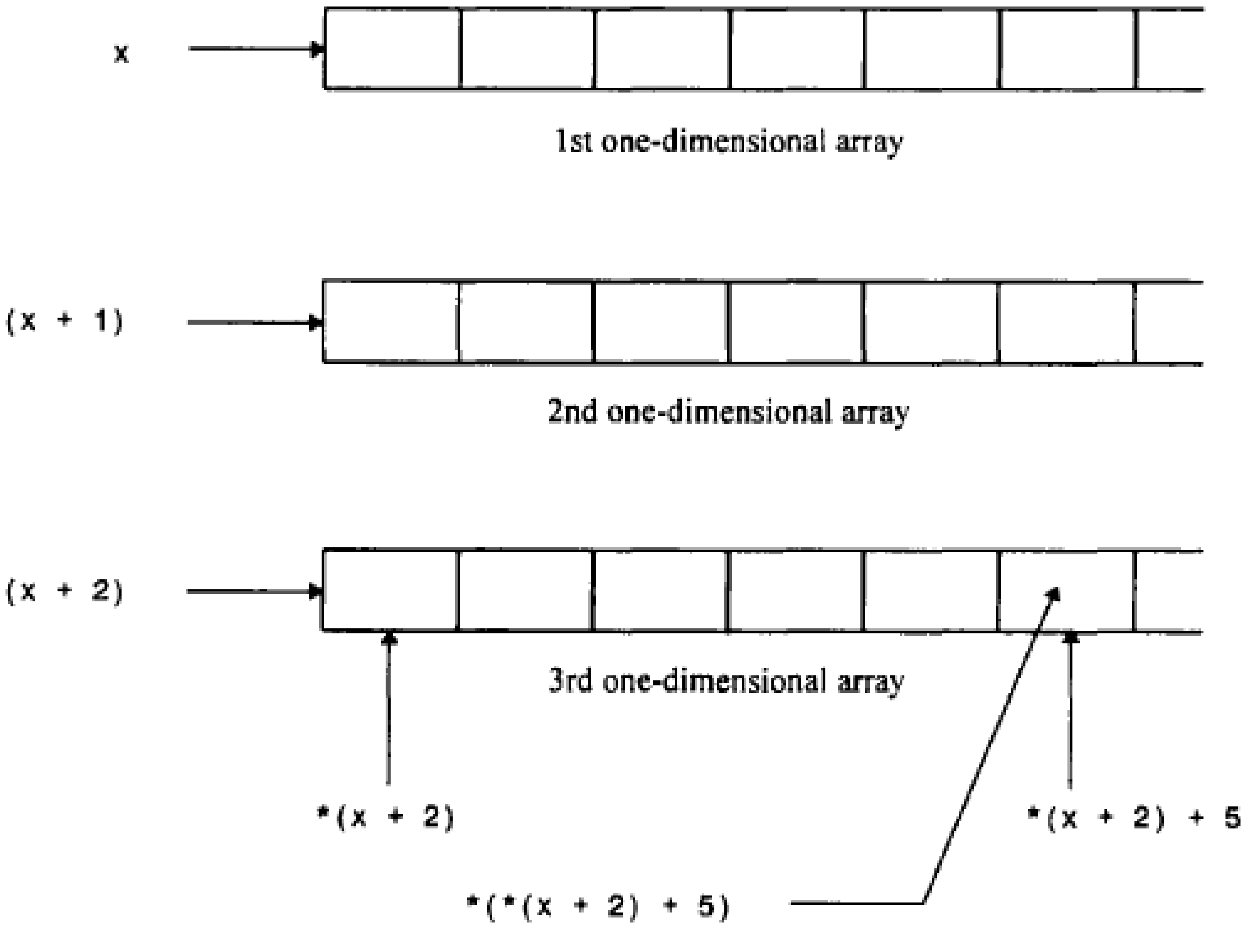
What will this print?



```
1234  56
1212  33
1434  80
1312  78
```

How to access 2-D Arrays using Pointers

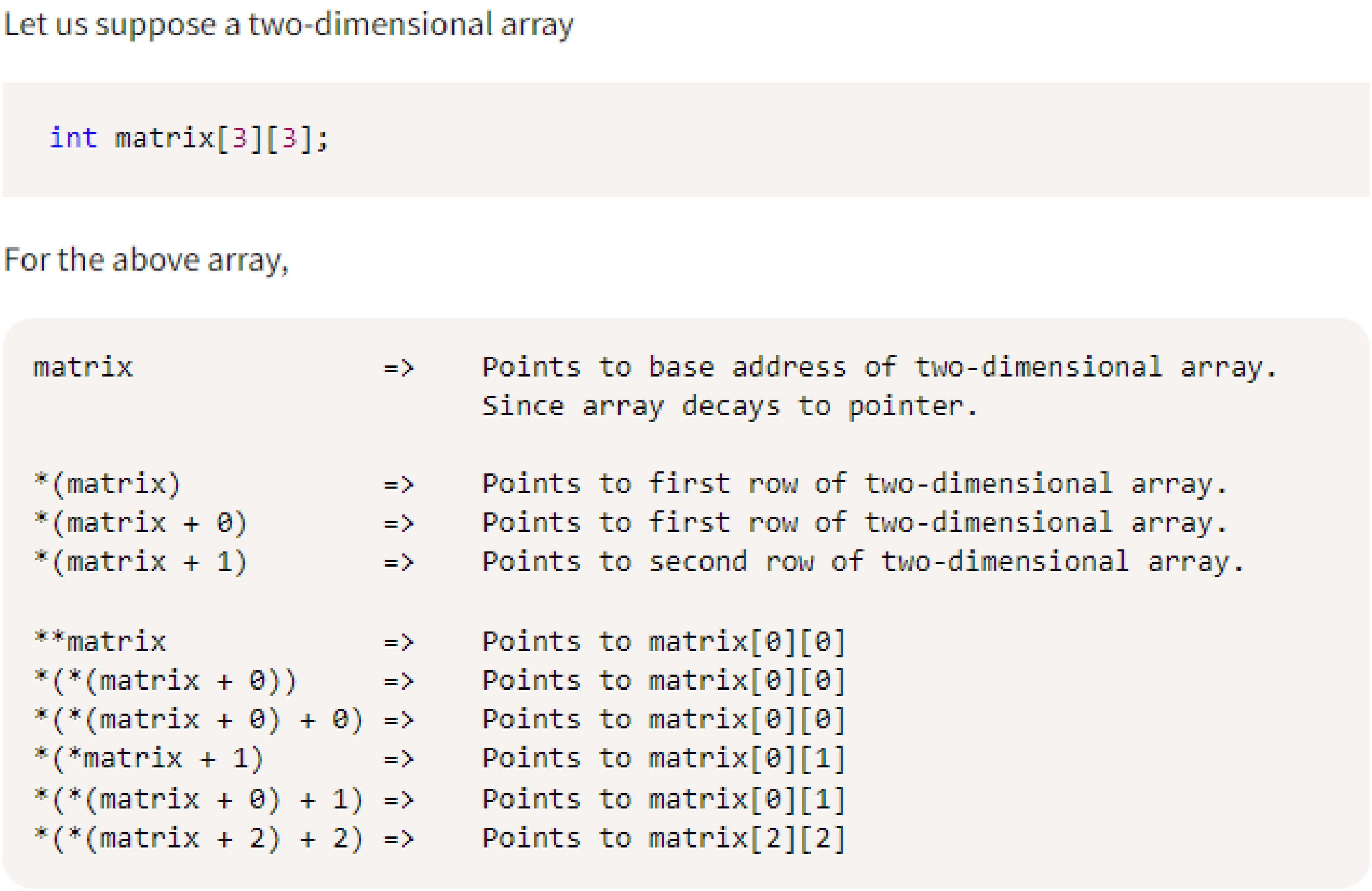
In a nutshell



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)

How to access 2-D Arrays using Pointers

In a nutshell



Let us suppose a two-dimensional array

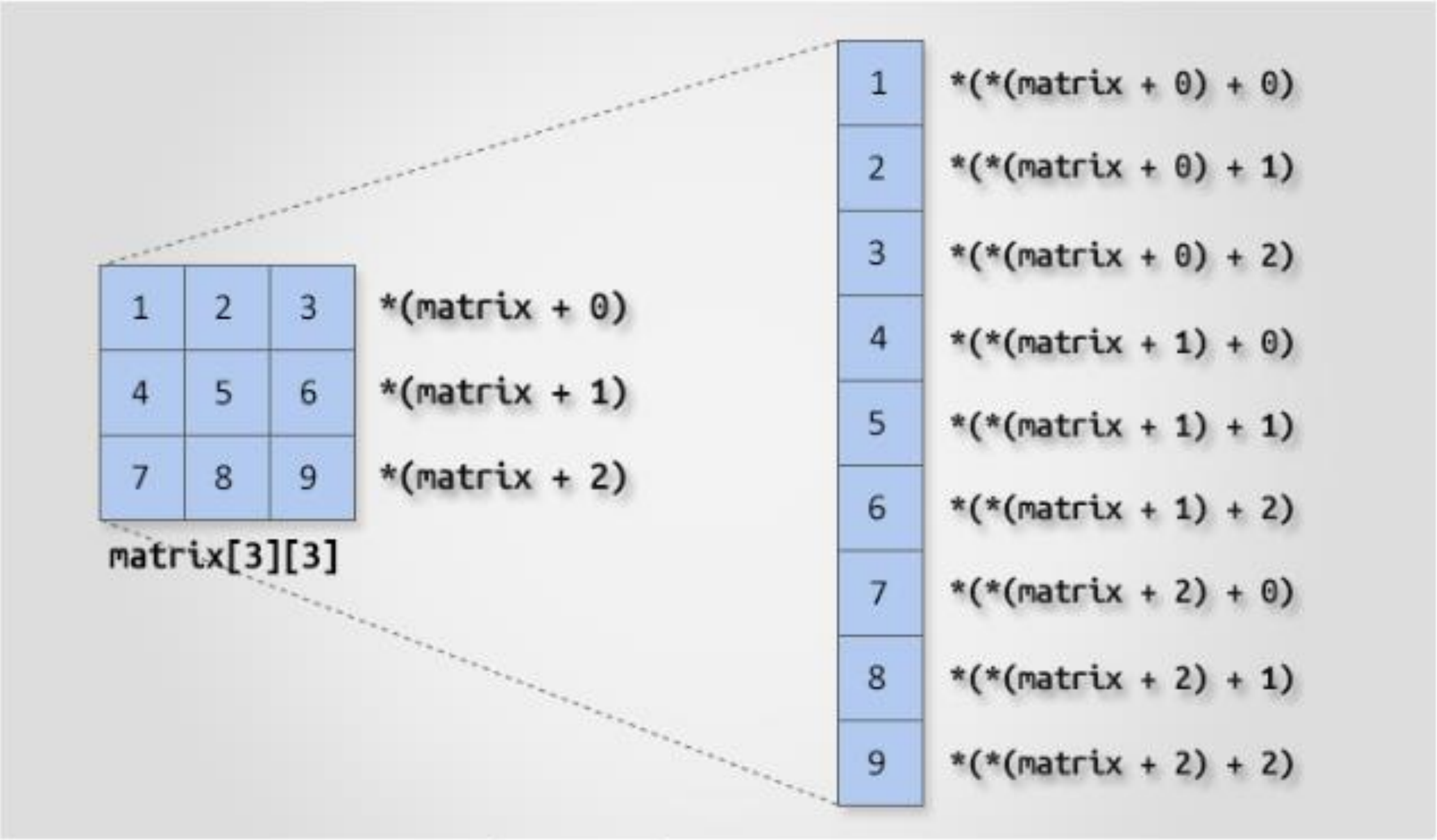
```
int matrix[3][3];
```

For the above array,

```
matrix                =>    Points to base address of two-dimensional array.
                            Since array decays to pointer.

*(matrix)             =>    Points to first row of two-dimensional array.
*(matrix + 0)         =>    Points to first row of two-dimensional array.
*(matrix + 1)         =>    Points to second row of two-dimensional array.

**matrix              =>    Points to matrix[0][0]
*(*(matrix + 0))      =>    Points to matrix[0][0]
*(*(matrix + 0) + 0)  =>    Points to matrix[0][0]
*(*matrix + 1)        =>    Points to matrix[0][1]
*(*(matrix + 0) + 1)  =>    Points to matrix[0][1]
*(*(matrix + 2) + 2)  =>    Points to matrix[2][2]
```

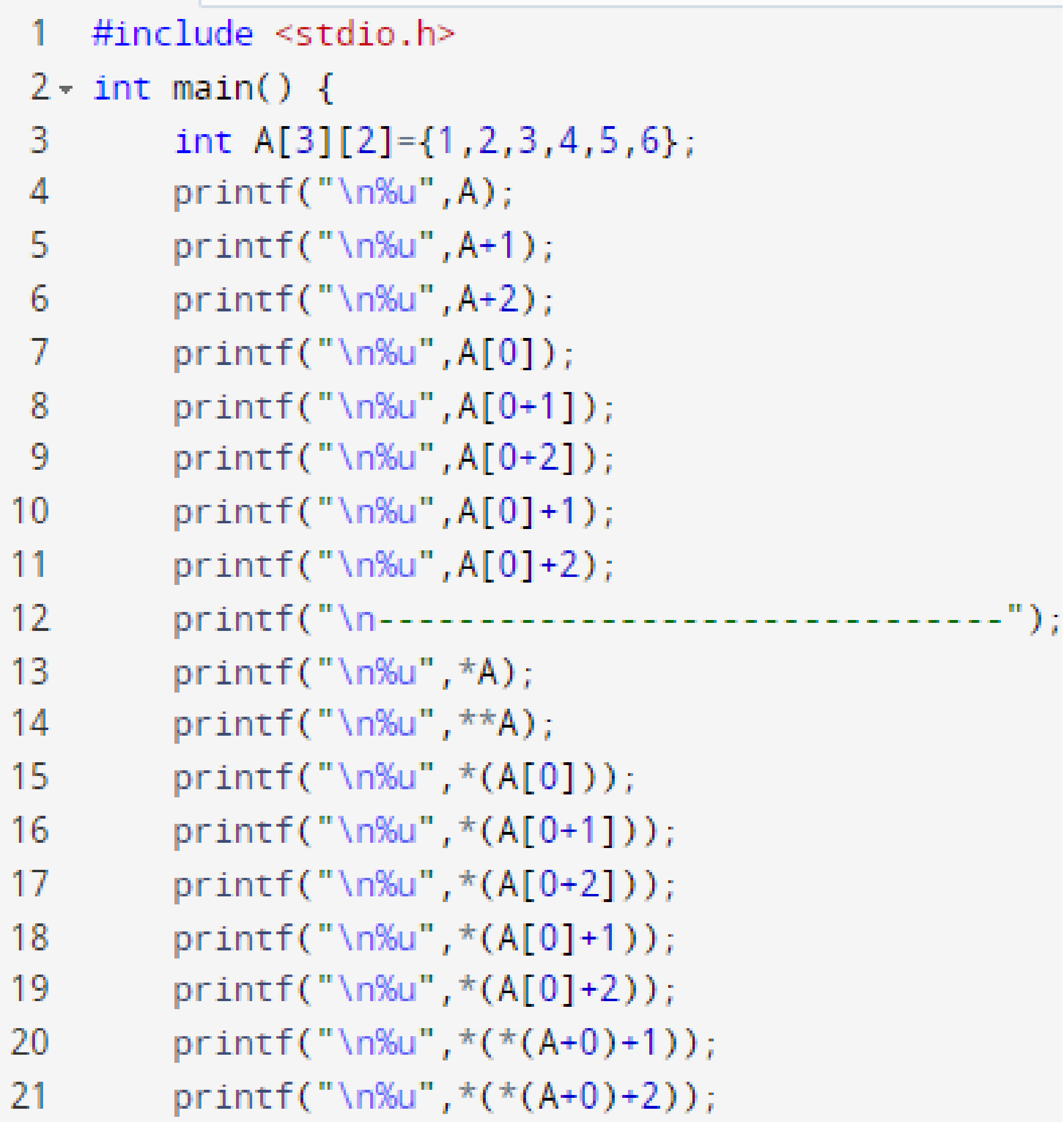
How to access 2-D Arrays using Pointers

In a nutshell

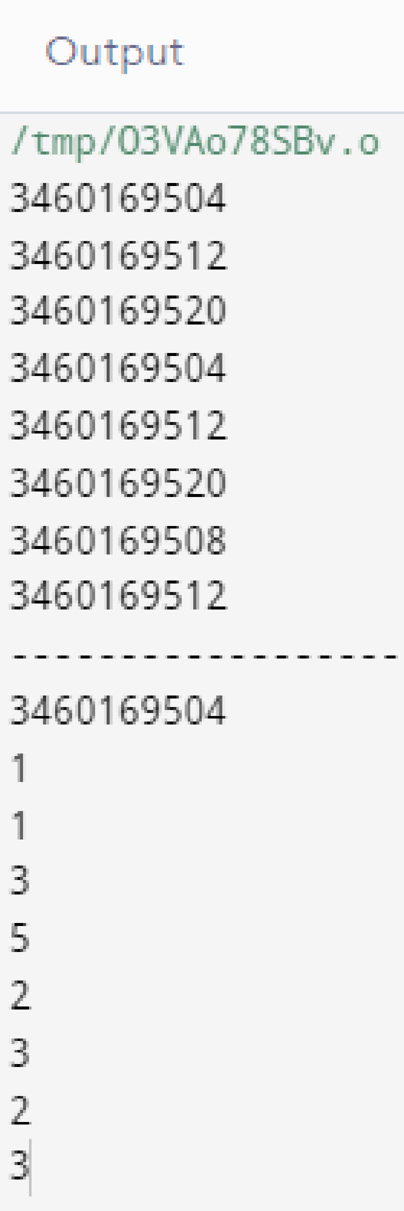


Two dimensional array access using pointer

Pointer and 2D array



```c
#include <stdio.h>
int main() {
    int A[3][2]={1,2,3,4,5,6};
    printf("\n%u",A);
    printf("\n%u",A+1);
    printf("\n%u",A+2);
    printf("\n%u",A[0]);
    printf("\n%u",A[0+1]);
    printf("\n%u",A[0+2]);
    printf("\n%u",A[0]+1);
    printf("\n%u",A[0]+2);
    printf("\n-------------------------------");
    printf("\n%u",*A);
    printf("\n%u",**A);
    printf("\n%u",*(A[0]));
    printf("\n%u",*(A[0+1]));
    printf("\n%u",*(A[0+2]));
    printf("\n%u",*(A[0]+1));
    printf("\n%u",*(A[0]+2));
    printf("\n%u",*(*(A+0)+1));
    printf("\n%u",*(*(A+0)+2));
```

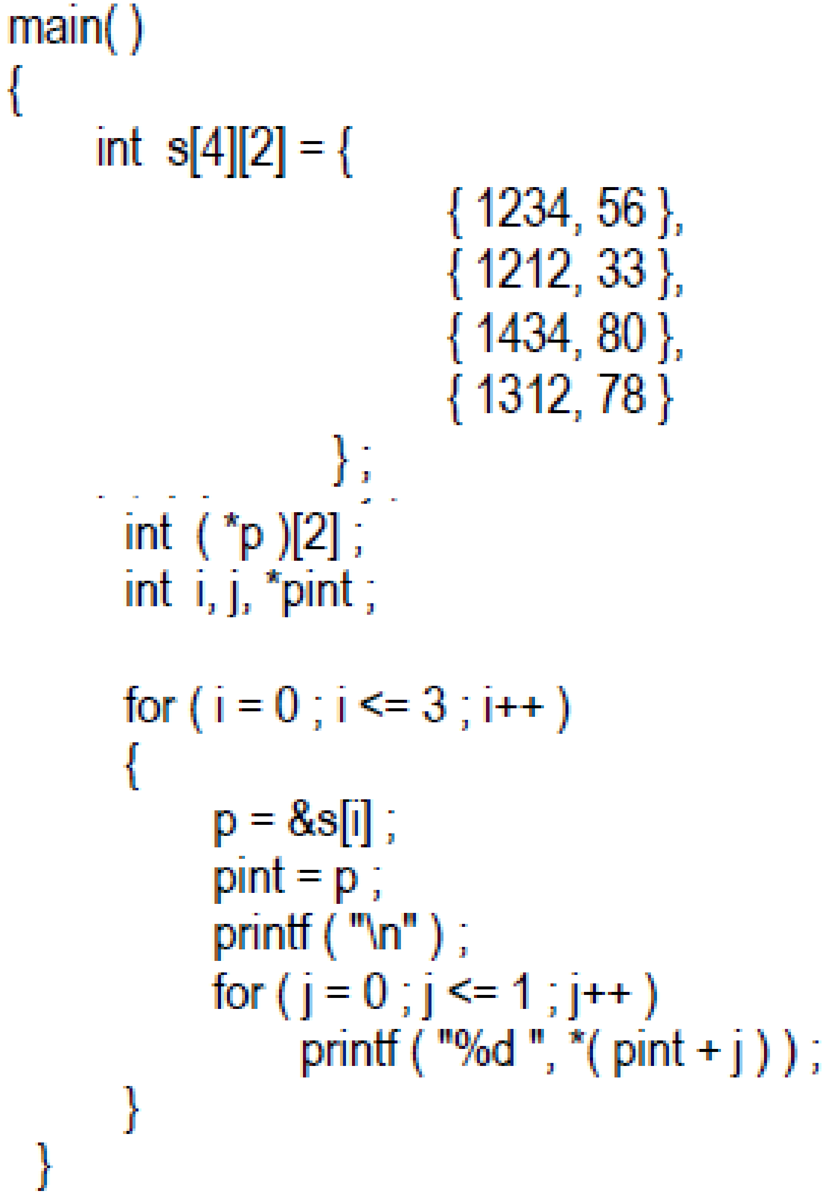


```
/tmp/03VAo78SBv.o
3460169504
3460169512
3460169520
3460169504
3460169512
3460169520
3460169508
3460169512
------------------
3460169504
1
1
3
5
2
3
2
3
```

Pointer to an Entire Array

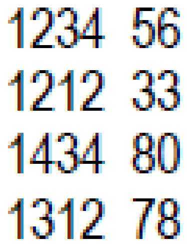
1. Till now we were accessing 2-D array elements.
2. Now we want to see the usage of pointer to a 2-D array
3. Similar to pointers to 1-D arrays

Pointer to an Entire Array



```
main( )
{
    int  s[4][2] = {
                        { 1234, 56 },
                        { 1212, 33 },
                        { 1434, 80 },
                        { 1312, 78 }
                   } ;
     int  ( *p )[2] ;
     int  i, j, *pint ;

     for ( i = 0 ; i <= 3 ; i++ )
     {
          p = &s[i] ;
          pint = p ;
          printf ( "\n" ) ;
          for ( j = 0 ; j <= 1 ; j++ )
               printf ( "%d ", *( pint + j ) ) ;
     }
 }
```

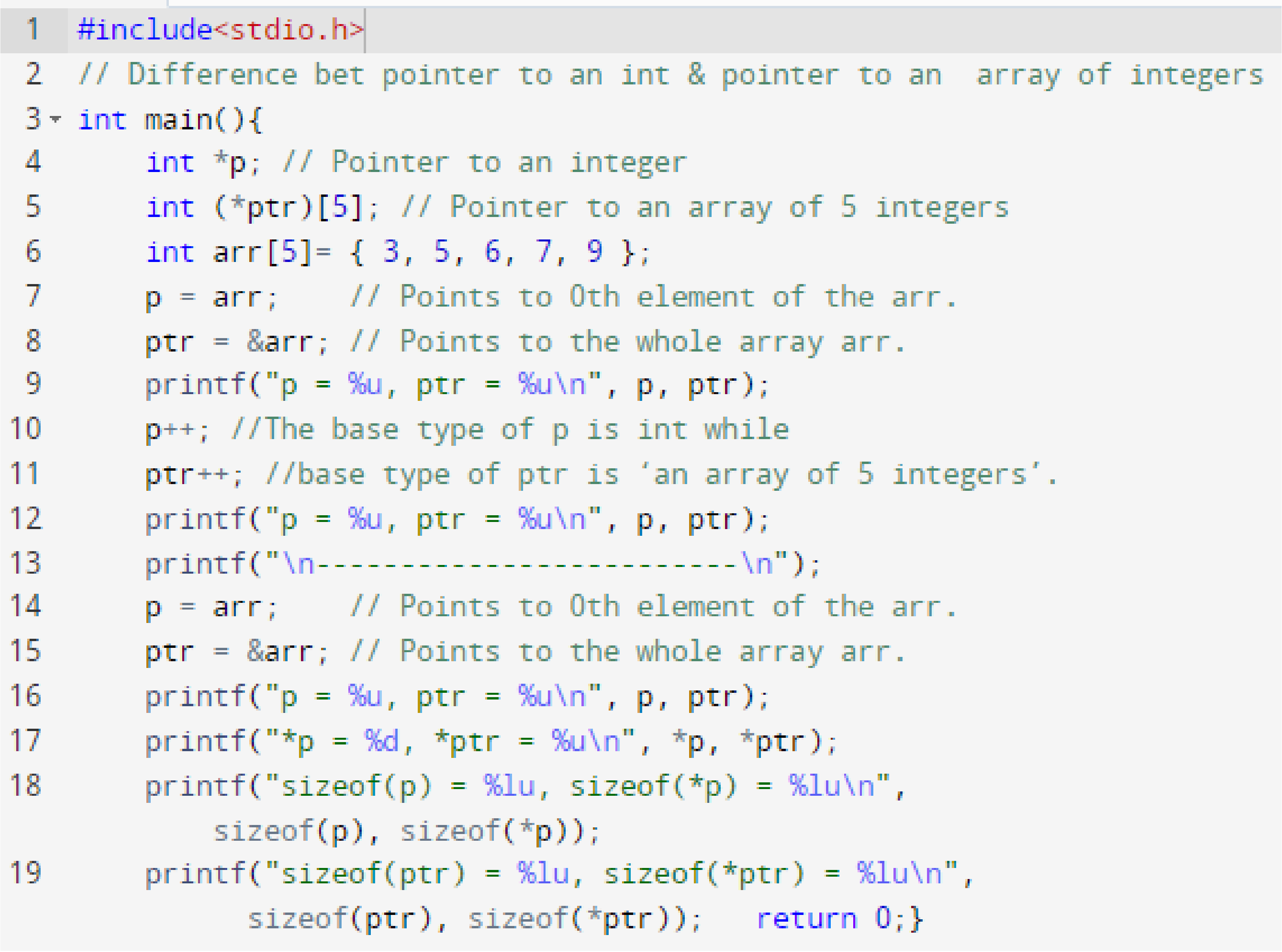
What will this print?



```
1234  56
1212  33
1434  80
1312  78
```

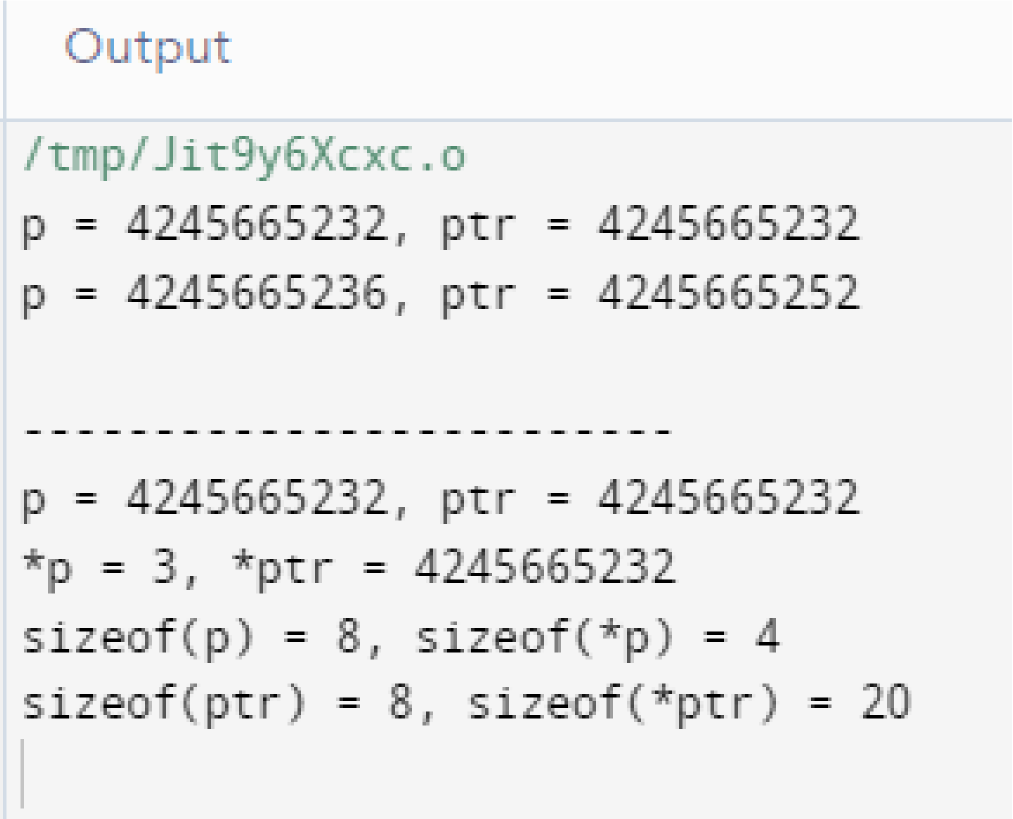
1. Here p is a pointer to an array of two integers.
2. In the outer for loop each time we store the address of a new one-dimensional array.
3. in the inner for loop using the pointer pint we have printed the individual elements of the 1-D array to which p is pointing

Pointer to an Entire Array



```c
#include<stdio.h>
// Difference bet pointer to an int & pointer to an  array of integers
int main(){
    int *p; // Pointer to an integer
    int (*ptr)[5]; // Pointer to an array of 5 integers
    int arr[5]= { 3, 5, 6, 7, 9 };
    p = arr;    // Points to 0th element of the arr.
    ptr = &arr; // Points to the whole array arr.
    printf("p = %u, ptr = %u\n", p, ptr);
    p++; //The base type of p is int while
    ptr++; //base type of ptr is ‘an array of 5 integers’.
    printf("p = %u, ptr = %u\n", p, ptr);
    printf("\n-------------------------\n");
    p = arr;    // Points to 0th element of the arr.
    ptr = &arr; // Points to the whole array arr.
    printf("p = %u, ptr = %u\n", p, ptr);
    printf("*p = %d, *ptr = %u\n", *p, *ptr);
    printf("sizeof(p) = %lu, sizeof(*p) = %lu\n",
        sizeof(p), sizeof(*p));
    printf("sizeof(ptr) = %lu, sizeof(*ptr) = %lu\n",
          sizeof(ptr), sizeof(*ptr));   return 0;}
```

Pointer to an Entire Array



Output

```
/tmp/Jit9y6Xcxc.o
p = 4245665232, ptr = 4245665232
p = 4245665236, ptr = 4245665252


----------------------------
p = 4245665232, ptr = 4245665232
*p = 3, *ptr = 4245665232
sizeof(p) = 8, sizeof(*p) = 4
sizeof(ptr) = 8, sizeof(*ptr) = 20
```

Passing 2-D Array to a Function

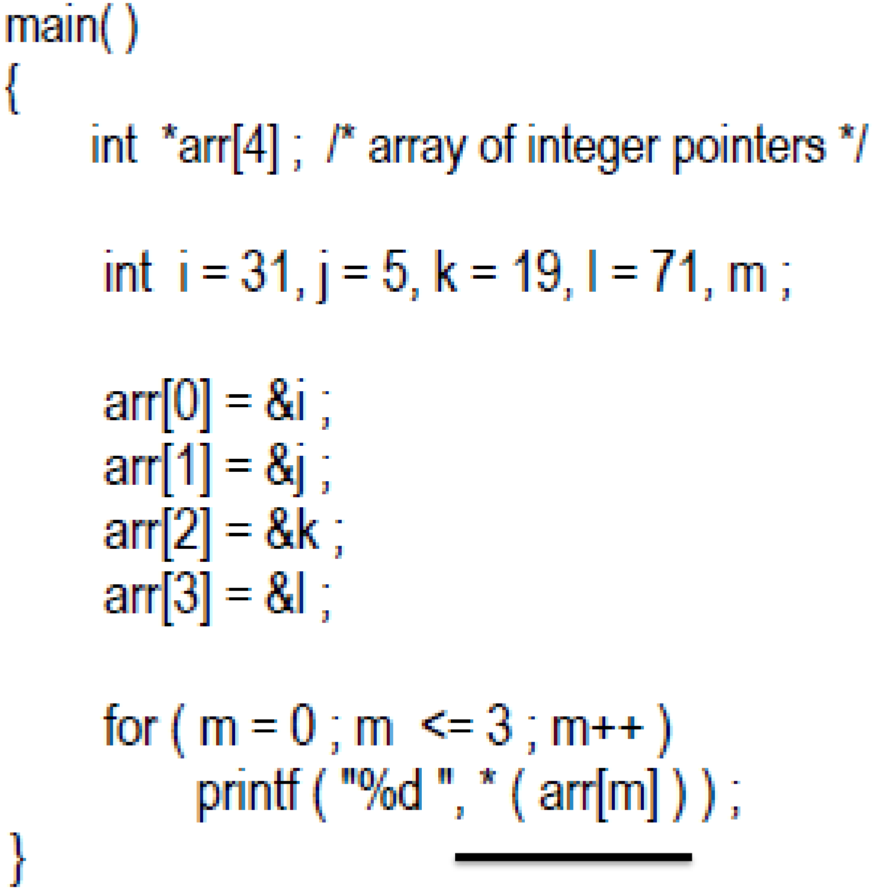
Three ways:

1. Just pass the base address
2. Pass the addresses of the 1-D sub-arrays as pointers
3. Pass the addresses of the 1-D sub-arrays as the more familiar expression using indices.

Array of Pointers

1. The way there can be an array of integers or an array of floats, similarly there can be an array of pointers.
2. A collection of addresses.
3. The addresses present in the array of pointers can be addresses of isolated variables or addresses of array elements or any other addresses.
4. All rules that apply to an ordinary array apply to the array of pointers as well

Array of Pointers

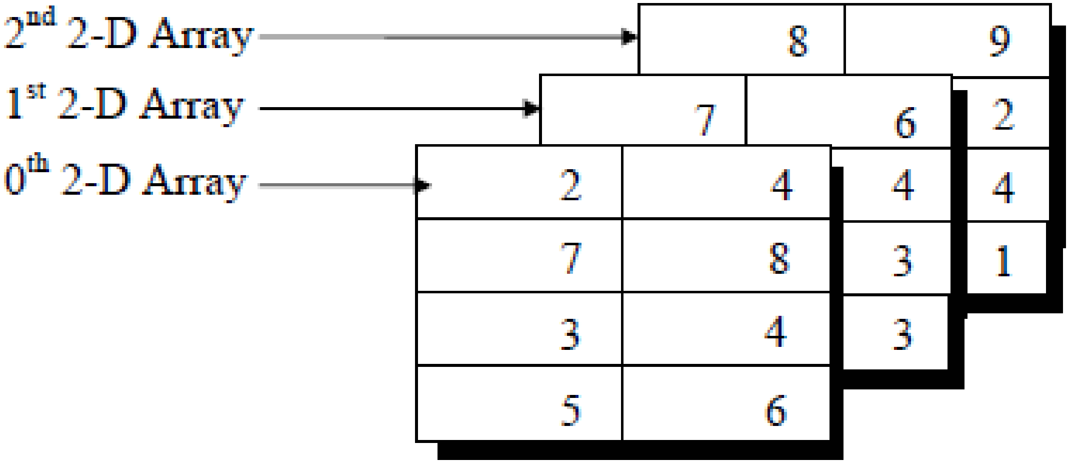


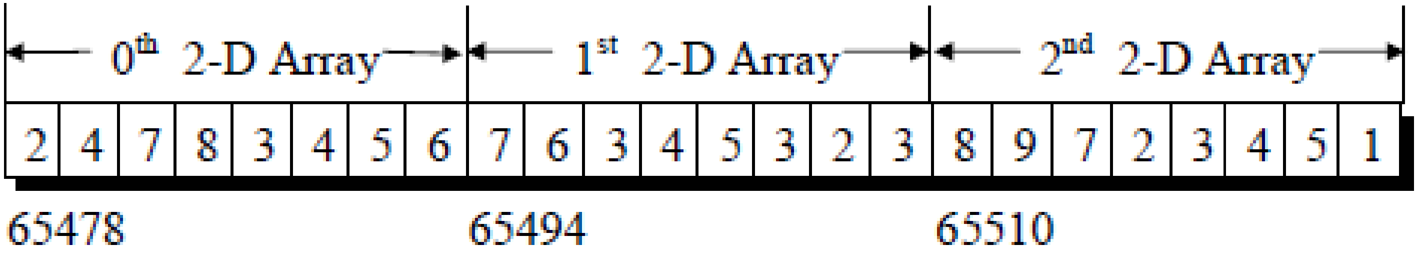
```
main( )
{
     int  *arr[4] ;  /* array of integer pointers */

      int  i = 31, j = 5, k = 19, l = 71, m ;

      arr[0] = &i ;
      arr[1] = &j ;
      arr[2] = &k ;
      arr[3] = &l ;

      for ( m = 0 ; m  <= 3 ; m++ )
           printf ( "%d ", * ( arr[m] ) ) ;
}
```

What is the output?

3-D Array

1. **An array of arrays of arrays.**

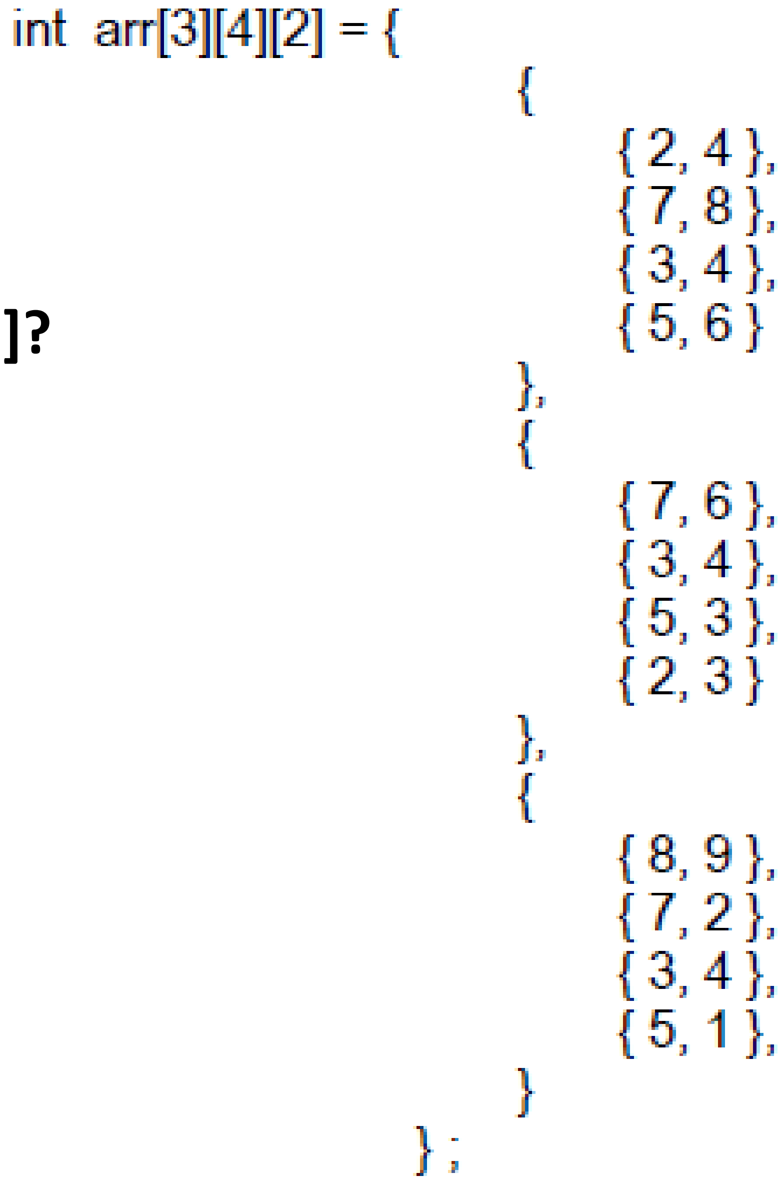




2	4	7	8	3	4	5	6	7	6	3	4	5	3	2	3	8	9	7	2	3	4	5	1

3-D Array - Initializing

1. So, what is arr[2][3][1]?



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