Multidimensional Arrays

Scanned with CamScanner

Two-Dimensional Arrays

The two-dimensional array is also called a matrix. Let us see how to create this array and work with it. Here is a sample program that stores roll number and marks obtained by a student side-by-side in a matrix.

```
# include <stdio.h>
int main( )
{
    int  stud[ 4 ][ 2 ] ;
    int  i, j ;

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

    return 0 ;
}
```

There are two parts to the program—in the first part, through a **for** loop, we read in the values of roll no. and marks, whereas, in the second part through another **for** loop, we print out these values.

Look at the **scanf( )** statement used in the first **for** loop:

```
scanf ( "%d %d", &stud[ i ][ 0 ], &stud[ i ][ 1 ] ) ;
```


Scanned with CamScanner

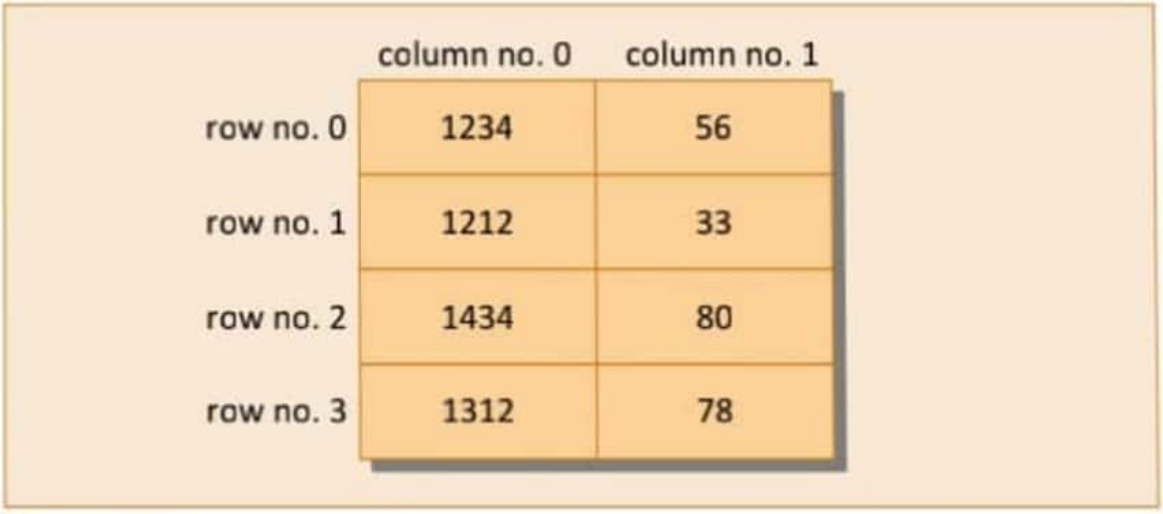


Figure 14.1

Thus, 1234 is stored in **stud[0][0]**, 56 is stored in **stud[0][1]** and so on. The above arrangement highlights the fact that a two- dimensional array is nothing but a collection of a number of one- dimensional arrays placed one below the other.

In our sample program, the array elements have been stored row-wise and accessed row-wise. However, you can access the array elements column-wise as well. Traditionally, the array elements are being stored and accessed row-wise; therefore we would also stick to the same strategy.

Initializing a Two-Dimensional Array

How do we initialize a two-dimensional array? As simple as this...

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

or even this would work...

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

of course, with a corresponding loss in readability.

It is important to remember that, while initializing a 2-D array, it is necessary to mention the second (column) dimension, whereas the first dimension (row) is optional.

Scanned with CamScanner

Thus the declarations,

```
int arr[ 2 ][ 3 ] = { 12, 34, 23, 45, 56, 45 } ;
int arr[   ][ 3 ] = { 12, 34, 23, 45, 56, 45 } ;
```

are perfectly acceptable,

whereas,

```
int arr[ 2 ][   ] = { 12, 34, 23, 45, 56, 45 } ;
int arr[   ][   ] = { 12, 34, 23, 45, 56, 45 } ;
```

would never work.

Memory Map of a Two-Dimensional Array

Let us reiterate the arrangement of array elements in a two-dimensional array of students, which contains roll nos. in one column and the marks in the other.

The array arrangement shown in Figure 14.1 is only conceptually true. This is because memory doesn't contain rows and columns. In memory, whether it is a one-dimensional or a two-dimensional array, the array elements are stored in one continuous chain. The arrangement of array elements of a two-dimensional array in memory is shown in Figure 14.2:

Figure 14.2

We can easily refer to the marks obtained by the third student using the subscript notation as shown below.

```
printf ( "Marks of third student = %d", stud[ 2 ][ 1 ] ) ;
```

Can we not refer to the same element using pointer notation, the way we did in one-dimensional arrays? Answer is yes. Only the procedure is slightly difficult to understand. So, read on...

Scanned with CamScanner

Pointers and Two-Dimensional Arrays

The C language embodies an unusual but powerful capability—it can treat parts of arrays as arrays. More specifically, each row of a two-dimensional array can be thought of as a one-dimensional array. This is a very important fact if we wish to access array elements of a two-dimensional array using pointers.

Thus, the declaration,

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

can be thought of as setting up an array of 5 elements, each of which is a one-dimensional array containing 2 integers. We refer to an element of a one-dimensional array using a single subscript. Similarly, if we can imagine **s** to be a one-dimensional array, then we can refer to its zeroth element as **s[0]**, the next element as **s[1]** and so on. More specifically, **s[0]** gives the address of the zeroth one-dimensional array, **s[1]** gives the address of the first one-dimensional array and so on. This fact can be demonstrated by the following program:

```
/* Demo: 2-D array is an array of arrays */
# include <stdio.h>
int main( )
{
    int  s[ 4 ][ 2 ] = {
                            { 1234, 56 },
                            { 1212, 33 },
                            { 1434, 80 },
                            { 1312, 78 }
                        };
    int  i ;
    for ( i = 0 ; i <= 3 ; i++ )
        printf ( "Address of %d th 1-D array = %u\n", i, s[ i ] ) ;
    return 0 ;
}
```

And here is the output...

```
Address of 0 th 1-D array = 65508
Address of 1 th 1-D array = 65516
Address of 2 th 1-D array = 65524
Address of 3 th 1-D array = 65532
```


Scanned with CamScanner

Let's figure out how the program works. The compiler knows that **s** is an array containing 4 one-dimensional arrays, each containing 2 integers. Each one-dimensional array occupies 4 bytes (two bytes for each integer). These one-dimensional arrays are placed linearly (zeroth 1-D array followed by first 1-D array, etc.). Hence, each one-dimensional array starts 4 bytes further along than the last one, as can be seen in the memory map of the array shown in Figure 14.3.

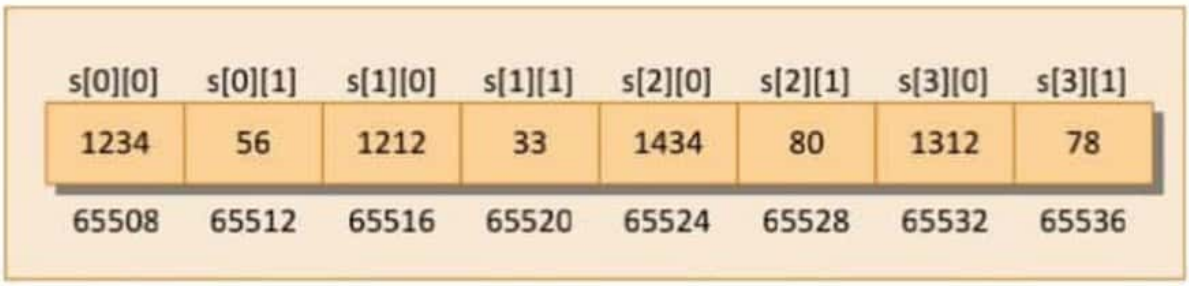


Figure 14.3

We know that the expressions **s[0]** and **s[1]** would yield the addresses of the zeroth and first one-dimensional array respectively. From Figure 14.3 these addresses turn out to be 65508 and 65516.

Now, we have been able to reach each one-dimensional array. What remains is to be able to refer to individual elements of a one-dimensional array. Suppose we want to refer to the element **s[2][1]** using pointers. We know (from the above program) that **s[2]** would give the address 65524, the address of the second one-dimensional array. Obviously (65524 + 1) would give the address 65528. Or **(s[2] + 1)** would give the address 65528. And the value at this address can be obtained by using the value at address operator, saying ***(s[2] + 1)**. But, we have already studied while learning one-dimensional arrays that **num[i]** is same as ***(num + i)**. Similarly, ***(s[2] + 1)** is same as, ***(*(s + 2) + 1)**. Thus, all the following expressions refer to the same element:

```
s[ 2 ][ 1 ]
* ( s[ 2 ] + 1 )
* ( * ( s + 2 ) + 1 )
```

Using these concepts, the following program prints out each element of a two-dimensional array using pointer notation:

```
/* Pointer notation to access 2-D array elements */
# include <stdio.h>
int main( )
{
```


Scanned with CamScanner

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

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

And here is the output...

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


Scanned with CamScanner

Pointer to an Array

If we can have a pointer to an integer, a pointer to a float, a pointer to a char, then can we not have a pointer to an array? We certainly can. The following program shows how to build and use it:

```
/* Usage of pointer to an array */
# include <stdio.h>
int main( )
{
    int  s[ 4 ][ 2 ] = {
                            { 1234, 56 },
                            { 1212, 33 },
                            { 1434, 80 },
                            { 1312, 78 }
                        };
    int  ( *p )[ 2 ] ;
```


Scanned with CamScanner

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

And here is the output...

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

Here **p** is a pointer to an array of two integers. Note that the parentheses in the declaration of **p** are necessary. Absence of them would make **p** an array of 2 integer pointers. Array of pointers is covered in a later section in this chapter. In the outer **for** loop, each time we store the address of a new one-dimensional array. Thus first time through this loop, **p** would contain the address of the zeroth 1-D array. This address is then assigned to an integer pointer **pint**. Lastly, 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.

But why should we use a pointer to an array to print elements of a 2-D array. Is there any situation where we can appreciate its usage better? The entity pointer to an array is immensely useful when we need to pass a 2-D array to a function. This is discussed in the next section.

Scanned with CamScanner

Array of Pointers

The way there can be an array of **int**s or an array of **float**s, similarly, there can be an array of pointers. Since a pointer variable always contains an address, an array of pointers would be nothing but a collection of addresses. The addresses present in the array of pointers can be addresses of isolated variables or addresses of array elements or any other addresses. All rules that apply to an ordinary array apply to the array of pointers as well. I think a program would clarify the concept.

```
# include <stdio.h>
int main( )
{
```


Scanned with CamScanner

```
    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\n", * ( arr[ m ] ) ) ;
    return 0 ;
}
```

Figure 14.5 shows the contents and the arrangement of the array of pointers in memory. As you can observe, **arr** contains addresses of isolated **int** variables **i**, **j**, **k** and **l**. The **for** loop in the program picks up the addresses present in **arr** and prints the values present at these addresses.

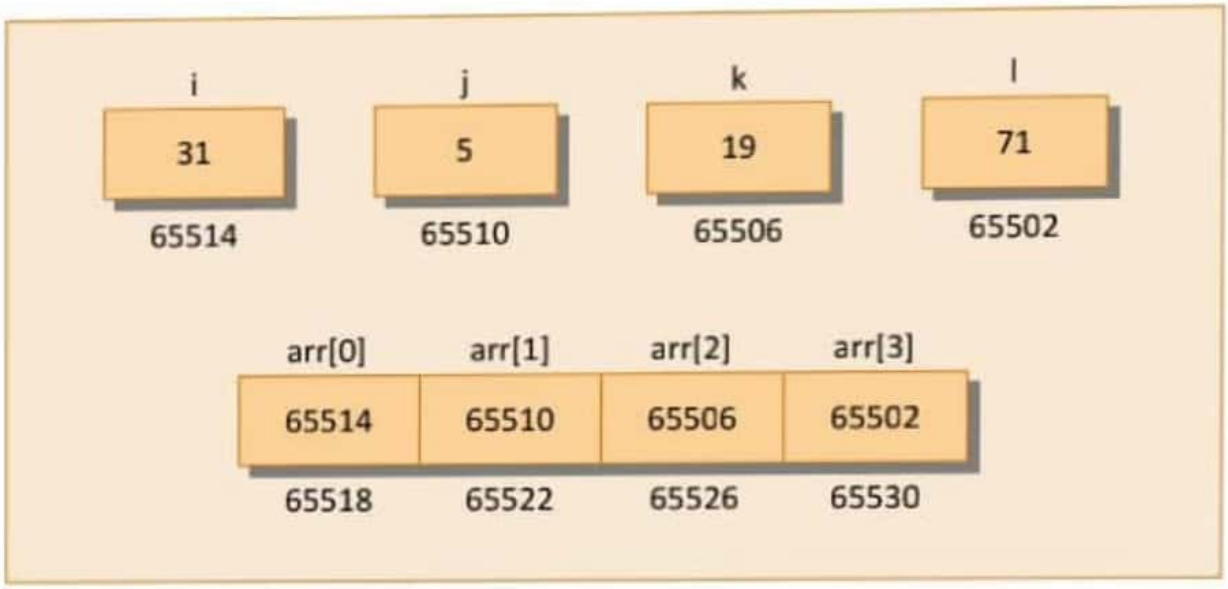


Figure 14.5

An array of pointers can even contain the addresses of other arrays. The following program would justify this:

```
# include <stdio.h>
int main( )
{
    static int a[ ] = { 0, 1, 2, 3, 4 } ;
    int *p[ ] = { a, a + 1, a + 2, a + 3, a + 4 } ;
```


Scanned with CamScanner

Three-Dimensional Array

We aren't going to show a programming example that uses a three-dimensional array. This is because, in practice, one rarely uses this array. However, an example of initializing a three-dimensional array will consolidate your understanding of subscripts.

```
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 },
                          }
                      };
```

A 3-D array can be thought of as an array of arrays of arrays. The outer array has three elements, each of which is a 2-D array of four 1-D arrays, each of which contains two integers. In other words, a 1-D array of two elements is constructed first. Then four such 1-D arrays are placed one below the other to give a 2-D array containing four rows. Then, three such 2-D arrays are placed one behind the other to yield a 3-D array containing three 2-D arrays. In the array declaration, note how the

Scanned with CamScanner

commas have been given. Figure 14.6 would possibly help you in visualizing the situation better.

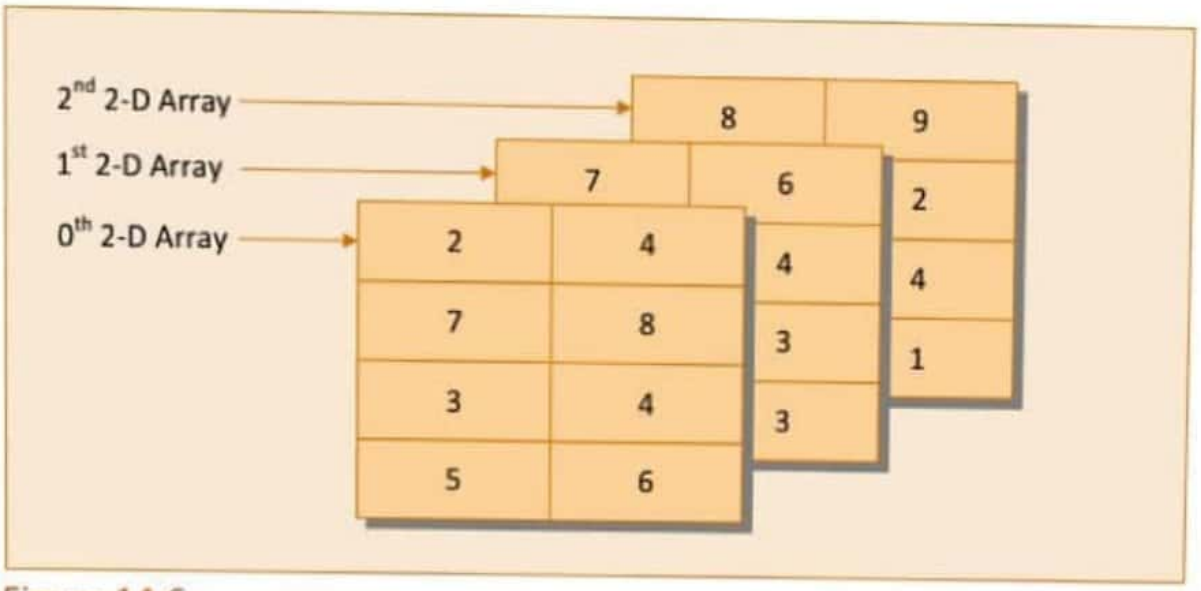


Figure 14.6

Again remember that the arrangement shown above is only conceptually true. In memory, the same array elements are stored linearly as shown in Figure 14.7.

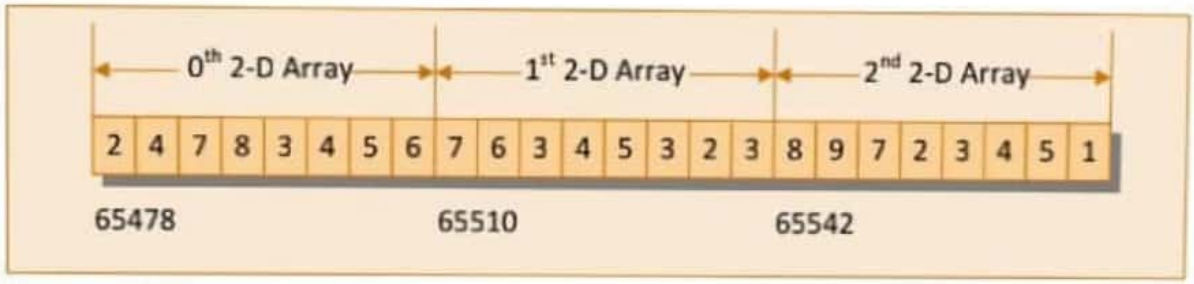


Figure 14.7

How would you refer to the array element 1 in the above array? The first subscript should be [2], since the element is in third two-dimensional array; the second subscript should be [3] since the element is in fourth row of the two-dimensional array; and the third subscript should be [1] since the element is in second position in the one-dimensional array. We can, therefore, say that the element 1 can be referred as **arr[2][3][1]**. It may be noted here that the counting of array elements even for a 3-D array begins with zero. Can we not refer to this element using pointer notation? Of course, yes. For example, the following two expressions refer to the same element in the 3-D array:

```
arr[ 2 ][ 3 ][ 1 ]
*( *( *( arr + 2 ) + 3 ) + 1 )
```


Scanned with CamScanner

Summary

(a) It is possible to construct multidimensional arrays.

(b) A 2-D array is a collection of several 1-D arrays.

(c) A 3-D array is a collection of several 2-D arrays.

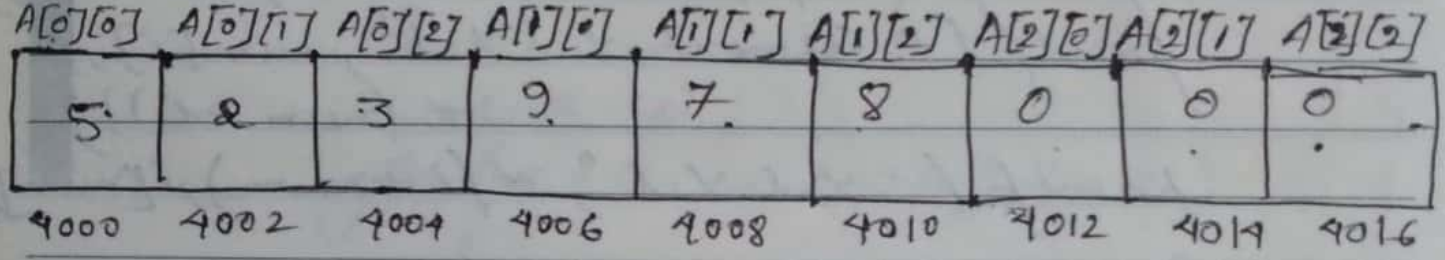
(d) All elements of a 2-D or a 3-D array are internally accessed using pointers.

Scanned with CamScanner

Multidimensional Arrays:-

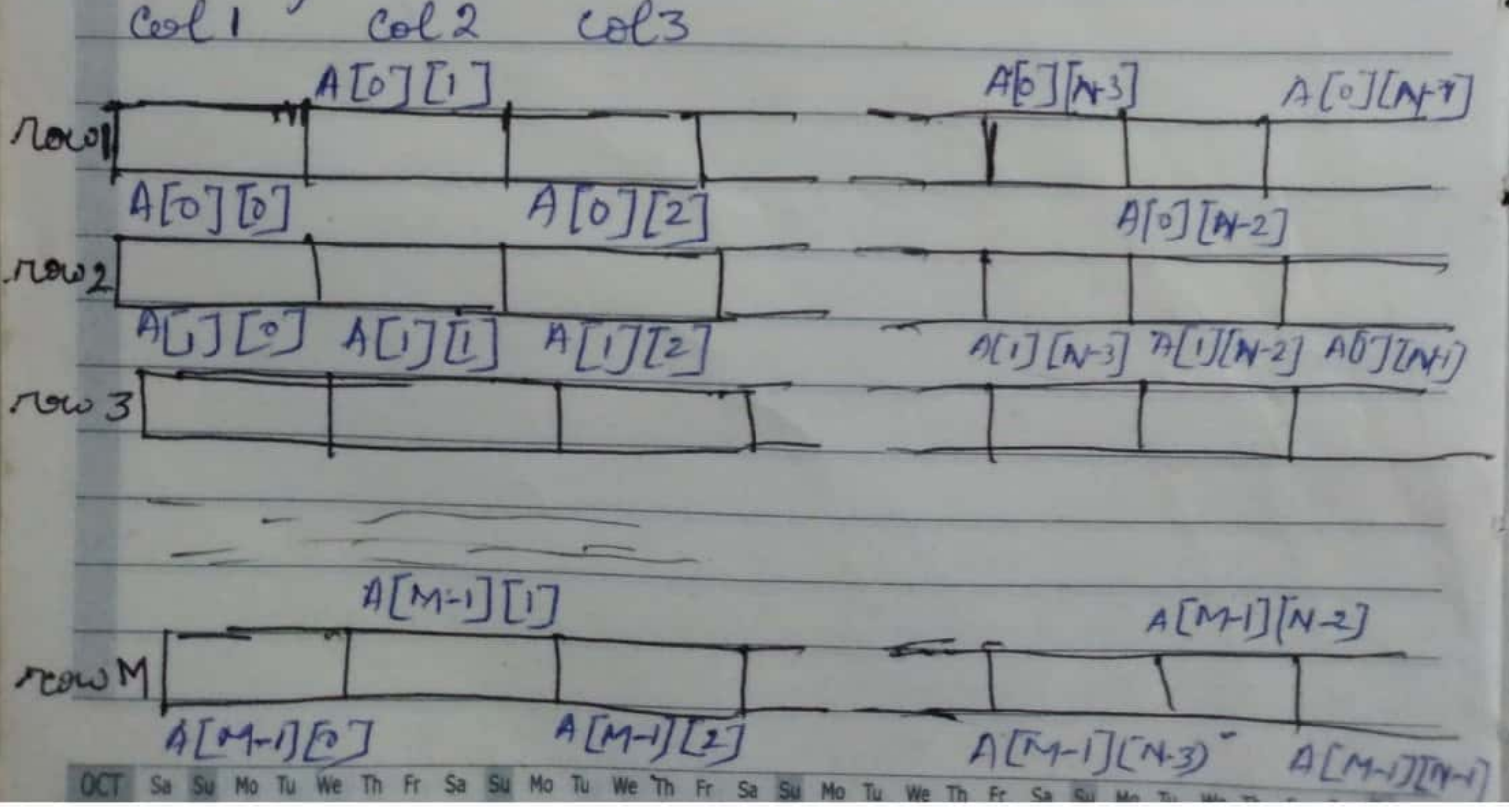
When the array elements are stored on the memory they are stored in contiguous memory cells. If we declare an array

int A[3][3] = {5, 2, 3, 9, 7, 8},
the memory allocation look like



Thursday • October 20

So, A[M][N] → M represents row & N represents column.



Scanned with CamScanner

```
int value[3][4] = {
                    {10, 20, 30},
                    {40, 50, 60},
                    {70, 80, 90},
                  };
```

the array elements will have the following initial value.

value[0][0] = 10 value[0][1] = 20 value[0][2] = 30,
value[1][0] = 40 value[1][1] = 50. value[1][2] = 60,
value[2][0] = 70 value[2][1] = 80. value[2][2] = 90

value[0][3] = 0
value[1][3] = 0
value[2][3] = 0

Scanned with CamScanner

Sum of 2D Array [using Multi-D array]

```
#define MR 10
#define MC 10

void readinput (int a[MR][MC], int R,
                int C);
int ComputeSum (int a[MR][MC], int R,
                int C);
void display (int a[MR][MC], int R,
              int C);

void main ()
{
```

Tuesday • October 25

```
    int r, c, sum;
    int A[MR][MC];

    printf ("\n How many rows & columns:
    scanf ("%d %d", &r, &c);
    printf ("\n The table is : ");
    readinput (A, r, c);
    Sum = ComputeSum (A, r, c)
    display (A, r, c);
    printf ("\n The sum of the elements
                %d", sum);
}
```


Scanned with CamScanner

```
void readinput (int a[MR][MC], int R,
                int C)
{
    int i, j;
    for (i=0; i<R; i++)
    {
        printf ("\n Enter data for row
                %d", i++);
    for (j=0; j<C; j++)
        scanf ("%d", &a[i][j]);
}
}
```

27

October • Thursday

```
void display (int a[MR][MC], int R,
              int C)
{ int i,j;
  printf ("\n The table is:");
  for (i=0; i<R; i++)
  {
    for (j=0; j<C; j++)
      printf ("%3d", a[i][j]);
    printf ("\n\n");
  }
return;
}
```


Scanned with CamScanner

```
int computesum(int a[MR][MC],
               int R, int C)
{
  int i,j, S=0;
  for(i=0; i<R; i++)
    for(j=0; j<C; j++)
      S = S + a[i][j];
  return (S);
}
```

O/P:-

29

Saturday • October

How many row & Column : 3 4

Enter data for row no : 1

2 3 4 5

Enter data for row no : 2

5 6 7 8

Sunday 30

Enter data for row no : 3

7 8 9 1

The table is :-

2 3 4 5

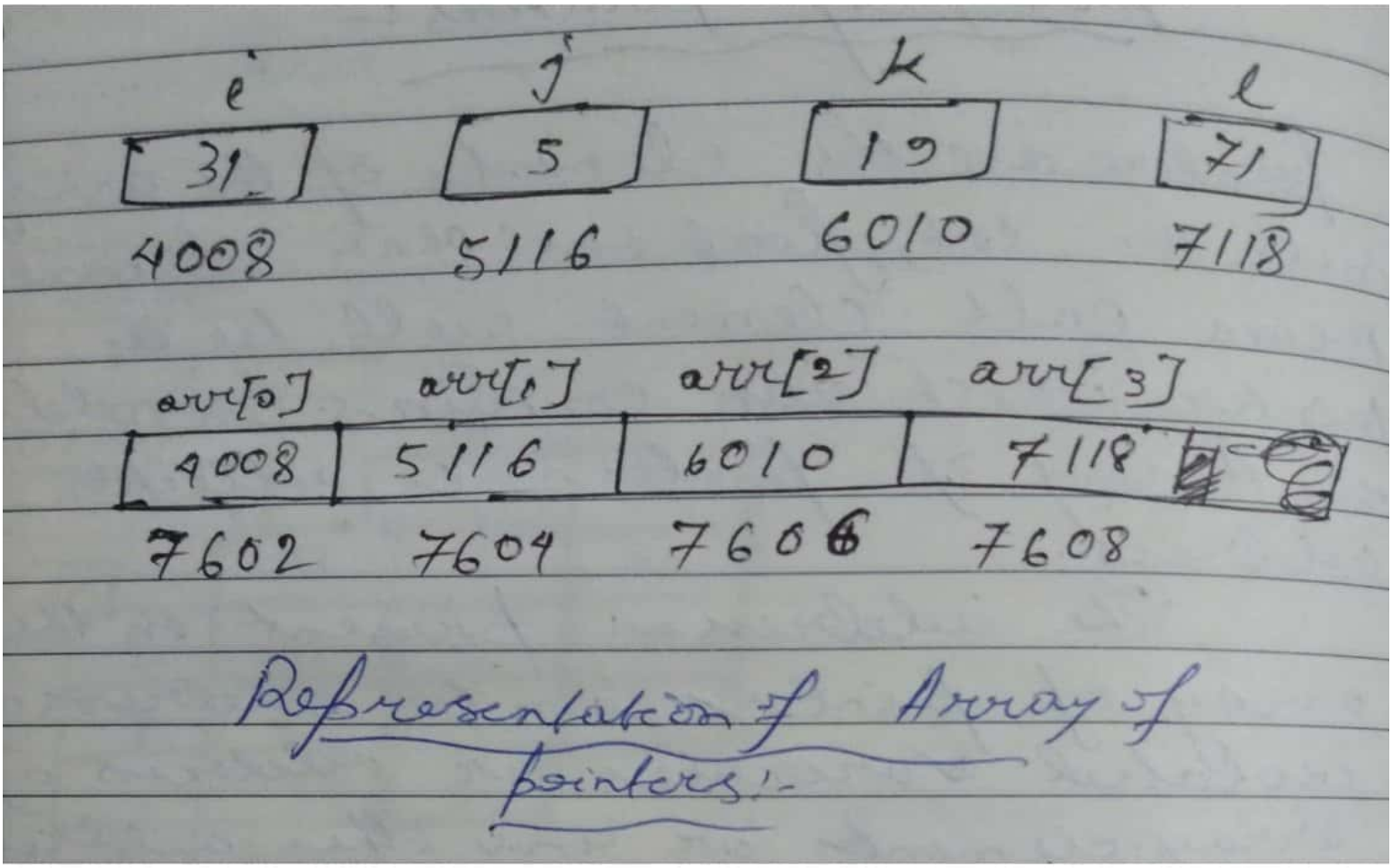
5 6 7 8

7 8 9 1

OCT 2011	Sa 01	Su 02	Mo 03	Tu 04	We 05	Th 06	Fr 07	Sa 08	Su 09	Mo 10	Tu 11	We 12	Th 13	Fr 14	Sa 15	Su 16	Mo 17	Tu 18	We 19	Th 20	Fr 21	Sa 22	Su 23	Mo 24	Tu 25	We 26	Th 27	Fr 28	Sa 29	Su 30	Mo 31

The sum of the elements is 65

Scanned with CamScanner



Representation of Array of pointers:-

Scanned with CamScanner

Pointer to Array and Array of pointers

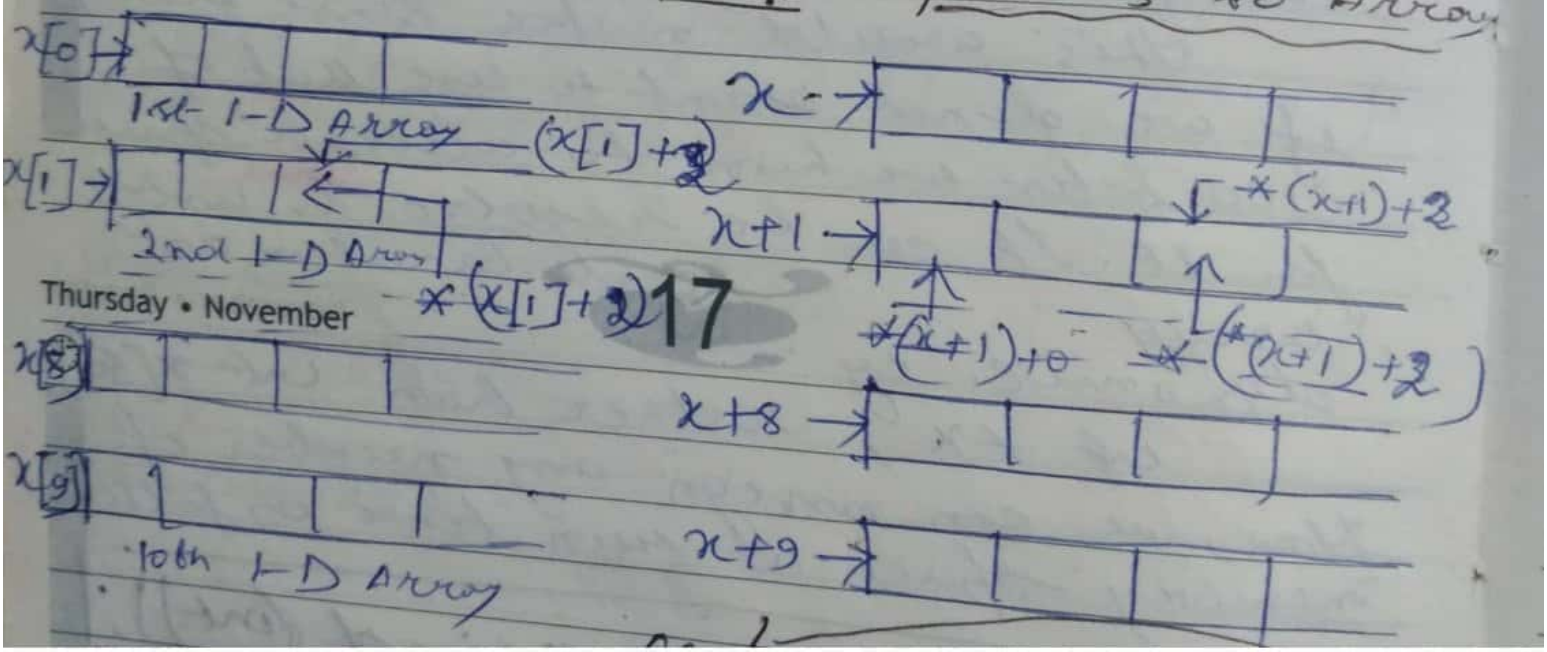
A two dimensional array can be written as :-

int (*x)[20] → pointer to array

int *x[20] → Array of pointers.

rather than int x[10][20]

Array of pointers | Memory Map | pointers to Array



Thursday • November 17

Scanned with CamScanner

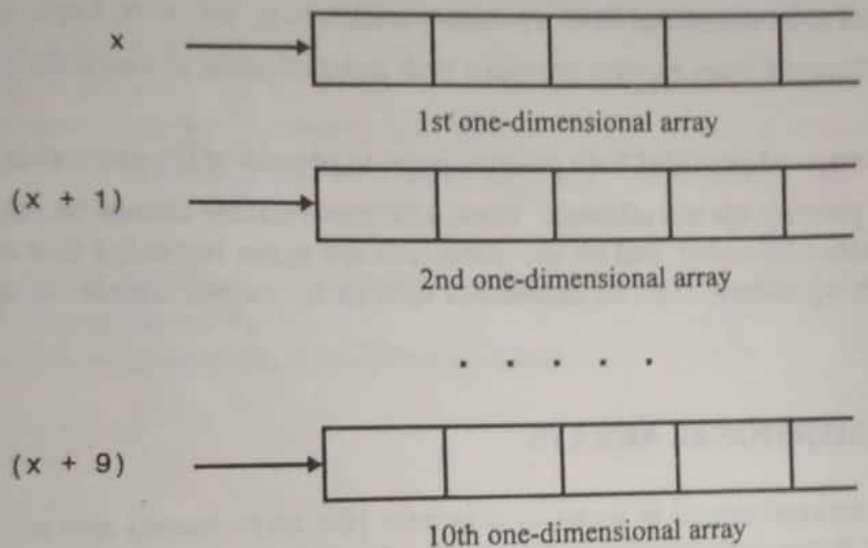
EXAMPLE 10.20 Suppose x is a two-dimensional integer array having 10 rows and 20 columns. We can declare x as

```
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, as illustrated in Fig. 10.4.



Scanned with CamScanner

DIMENSIONAL(2D) ARRAYS

We can perform the following operations on a two-dimensional array:

Transpose: Transpose of a `m × n` matrix `A` is given as a `n × m matrix B` where,

$$B_{i,j} = A_{j,i}$$

Sum: Two matrices that are compatible with each other can be added together thereby storing the result in the third matrix. Two matrices are said to be compatible when they have the same number of rows and columns. Elements of the matrices can be added by writing:

$$C_{i,j} = A_{i,j} + B_{i,j}$$

Scanned with CamScanner

Write a program to read and display a 3×3 matrix.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int i, j, mat[3][3];
   clrscr();

   printf("\n Enter the elements of the
   matrix ");
   printf("\n **************************");

    for(i=0;i<3;i++)
    {
       for(j=0;j<3;j++)
          scanf("%d",&mat[i][j]);
    }
    printf("\n The elements of the matrix are ");
    printf("\n **************************");

    for(i=0;i<3;i++)
    {
       printf("\n");
       for(j=0;j<3;j++)
          printf("\t %d%d%d",i, j, mat[i][j]);
    }
    return 0;
}
```

Output

```
Enter the elements of the matrix
**************************
1 2 3 4 5 6 7 8 9
The elements of the matrix are
**************************
1 2 3
4 5 6
7 8 9
```


Scanned with CamScanner