Arrays

What are Arrays

- C language provides a capability that enables the user to design a set of similar data types, called array
- ordinary variables (the ones which we have used so far) are capable of holding only one value at a time
- However, there are situations in which we would want to store more than one value at a time in a single variable.
- For example, suppose we wish to arrange the percentage marks obtained by 100 students in ascending order.

What are Arrays

- In such a case we have two options to store these marks in memory:
- Construct 100 variables to store percentage marks obtained by 100 different students, i.e. each variable containing one student’s marks.
- Construct one variable (called array or subscripted variable) capable of storing or holding all the hundred values.
- the second alternative is better as it would be much easier to handle one variable than handling 100 different variables
- Moreover, there are certain logics that cannot be dealt with, without the use of an array

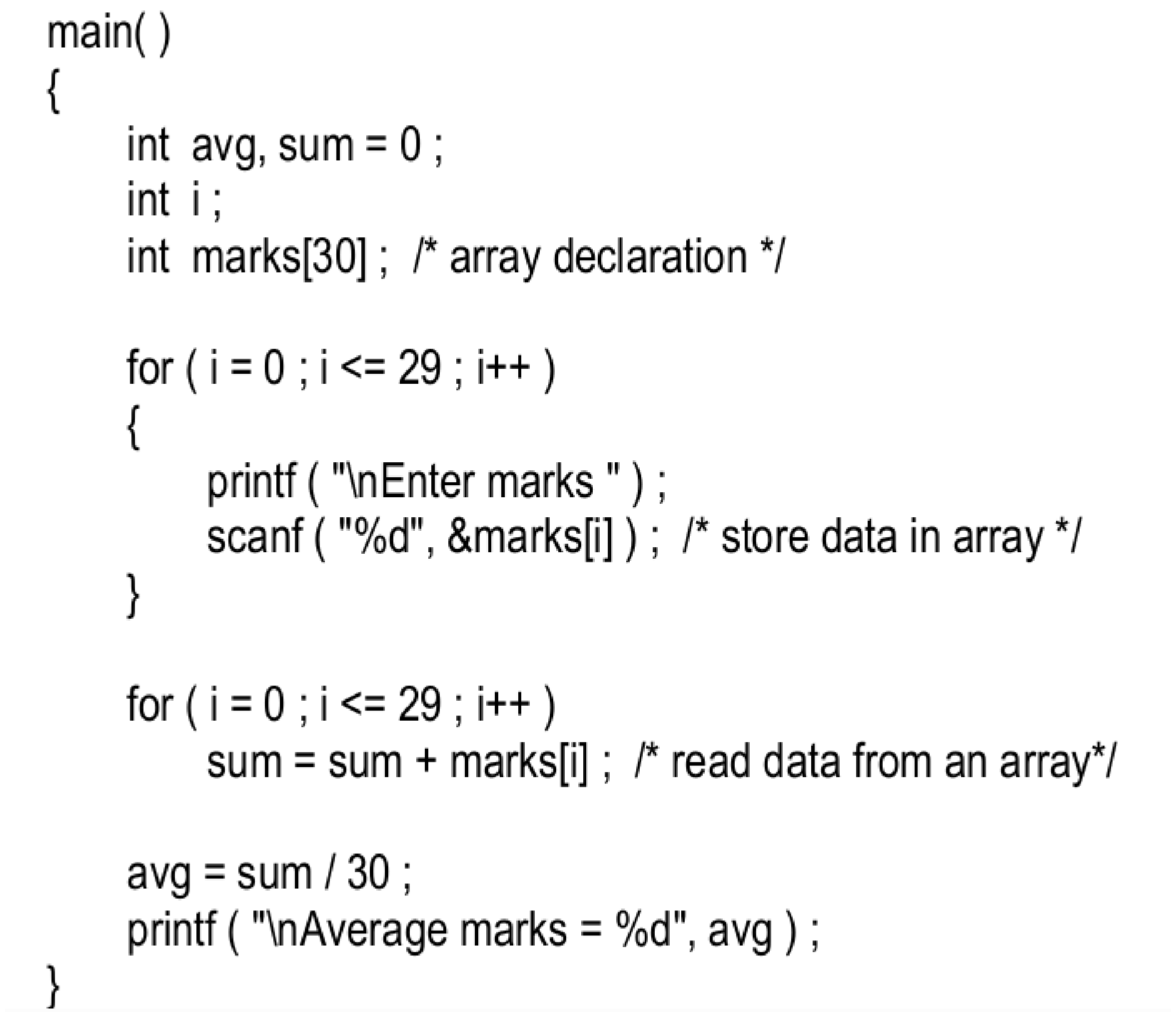
Definition

- An array is a collective name given to a group of ‘similar quantities’.
- These similar quantities could be percentage marks of 100 students, or salaries of 300 employees, or ages of 50 employees.
- What is important is that the quantities must be ‘similar’.
- Each member in the group is referred to by its position in the group.
- For example, assume the following group of numbers, which represent percentage marks obtained by five students.
- per = { 48, 88, 34, 23, 96 }

Referring to elements

- in C the counting of elements begins with 0
- Thus, in this example per[3] refers to 23 and per[4] refers to 96
- In general, the notation would be per[i], where, i can take a value 0, 1, 2, 3, or 4, depending on the position of the element being referred
- Thus, an array is a collection of similar elements. These similar elements could be all ints, or all floats, or all chars, etc.
- Usually, the array of characters is called a ‘string’
- Remember that all elements of any given array must be of the same type. i.e. we cannot have an array of 10 numbers, of which 5 are ints and 5 are floats.

A Simple Program Using Array



```
main( )
{
    int  avg, sum = 0 ;
    int  i ;
    int  marks[30] ;  /* array declaration */

    for ( i = 0 ; i <= 29 ; i++ )
    {
        printf ( "\nEnter marks " ) ;
        scanf ( "%d", &marks[i] ) ;  /* store data in array */
    }

    for ( i = 0 ; i <= 29 ; i++ )
        sum = sum + marks[i] ;  /* read data from an array*/

    avg = sum / 30 ;
    printf ( "\nAverage marks = %d", avg ) ;
}
```

- a program to find average marks obtained by a class of 30 students in a test.

Array Declaration

- int marks[30] ;
- Here, int specifies the type of the variable, just as it does with ordinary variables and the word marks specifies the name of the variable
- The number 30 tells how many elements of the type int will be in our array.
- This number is often called the ‘dimension’ of the array.
- The bracket ([ ]) tells the compiler that we are dealing with an array.

Accessing Elements of an Array

- Individual elements in the array can be referred by the number in the brackets following the array name
- This number specifies the element’s position in the array
- All the array elements are numbered, starting with 0
- Thus, marks[2] is not the second element of the array, but the third.
- In our program we are using the variable i as a subscript to refer to various elements of the array
- This variable can take different values and hence can refer to the different elements in the array in turn.
- This ability to use variables as subscripts is what makes arrays so useful.

Entering Data into an Array

- The for loop causes the process of asking for and receiving a student’s marks from the user to be repeated 30 times
- The first time through the loop, i has a value 0, so the scanf( ) function will cause the value typed to be stored in the array element marks[0], the first element of the array. This process will be repeated until i becomes 29
- In scanf( ) function, we have used the “address of” operator (&) on the element marks[i] of the array, just as we have used it earlier on other variables (&rate, for example).
- In so doing, we are passing the address of this particular array element to the scanf( ) function, rather than its value; which is what scanf( ) requires.

Reading Data from an Array

- The balance of the program reads the data back out of the array and uses it to calculate the average
- The for loop is much the same, but now the body of the loop causes each student’s marks to be added to a running total stored in a variable called sum
- When all the marks have been added up, the result is divided by 30, the number of students, to get the average.

Summary

- An array is a collection of similar elements.
- The first element in the array is numbered 0, so the last element is 1 less than the size of the array.
- An array is also known as a subscripted variable.
- Before using an array its type and dimension must be declared.
- However big an array its elements are always stored in contiguous memory locations. This is a very important point which we would discuss in more detail later on.

More on Arrays

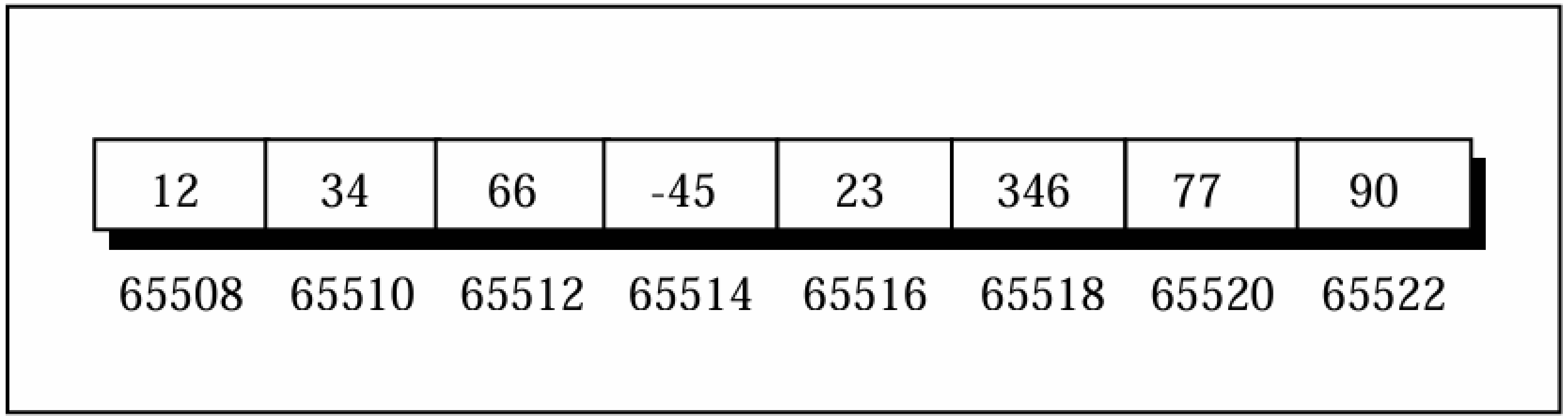
- Array Initialisation
- Following are a few examples that demonstrate this
- int num[6] = { 2, 4, 12, 5, 45, 5 } ;
- int n[ ] = { 2, 4, 12, 5, 45, 5 } ;
- float press[ ] = { 12.3, 34.2 -23.4, -11.3 } ;
- Till the array elements are not given any specific values, they are supposed to contain garbage values.
- If the array is initialised where it is declared, mentioning the dimension of the array is optional as in the 2nd example above

Array Elements in Memory

- Consider the following array declaration:
- int arr[8] ;
- What happens in memory when we make this declaration?
- 16 bytes get immediately reserved in memory, 2 bytes each for the 8 integers (under Windows/Linux the array would occupy 32 bytes as each integer would occupy 4 bytes).
- And since the array is not being initialized, all eight values present in it would be garbage values.
- This so happens because the storage class of this array is assumed to be auto. If the storage class is declared to be static then all the array elements would have a default initial value as zero.

Positioning of elements

- Whatever be the initial values, all the array elements would always be present in contiguous memory locations.



Bounds Checking

- In C there is no check to see if the subscript used for an array exceeds the size of the array.
- Data entered with a subscript exceeding the array size will simply be placed in memory outside the array; probably on top of other data, or on the program itself.
- This will lead to unpredictable results, to say the least, and there will be no error message to warn you that you are going beyond the array size.
- In some cases the computer may just hang.
- Thus, the following program may turn out to be suicidal.
- Thus, to see to it that we do not reach beyond the array size is entirely the programmer’s botheration and not the compiler’s.

```
main( )
{
      int  num[40], i ;

      for ( i = 0 ; i <= 100 ; i++ )
            num[i] = i ;
}
```

Passing Array Elements to a Function

- Array elements can be passed to a function by calling the function by value, or by reference.
- In the call by value we pass values of array elements to the function, whereas in the call by reference we pass addresses of array elements to the function.
- These two calls are illustrated below:

```
/* Demonstration of call by value */
main( )
{
     int  i ;
     int  marks[ ] = { 55, 65, 75, 56, 78, 78, 90 } ;

     for ( i = 0 ; i <= 6 ; i++ )
          display ( marks[i] ) ;
}

display ( int  m )
{
     printf ( "%d ", m ) ;
}
```

And here's the output...

```
55 65 75 56 78 78 90
```

```
/* Demonstration of call by reference */
main( )
{
	int  i ;
	int  marks[ ] = { 55, 65, 75, 56, 78, 78, 90 } ;

	for ( i = 0 ; i <= 6 ; i++ )
		disp ( &marks[i] ) ;
}

disp ( int  *n )
{
	printf ( "%d ", *n ) ;
}
```

And here's the output...

```
55 65 75 56 78 78 90
```

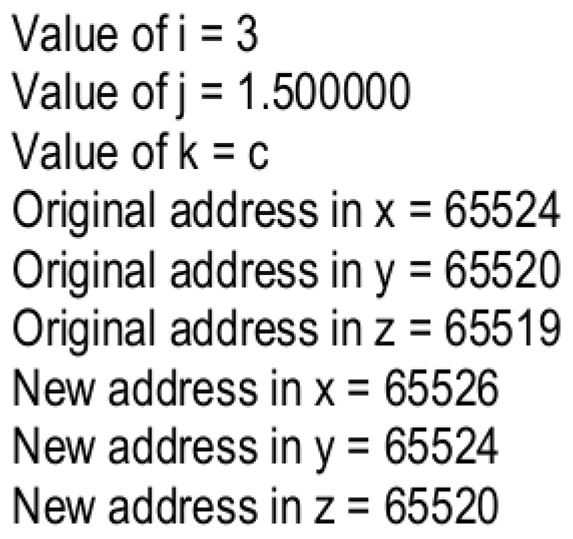
Exercise

- Read the following program carefully.
- The purpose of the function disp( ) is just to display the array elements on the screen.
- The program is only partly complete.
- You are required to write the function show( ) on your own.

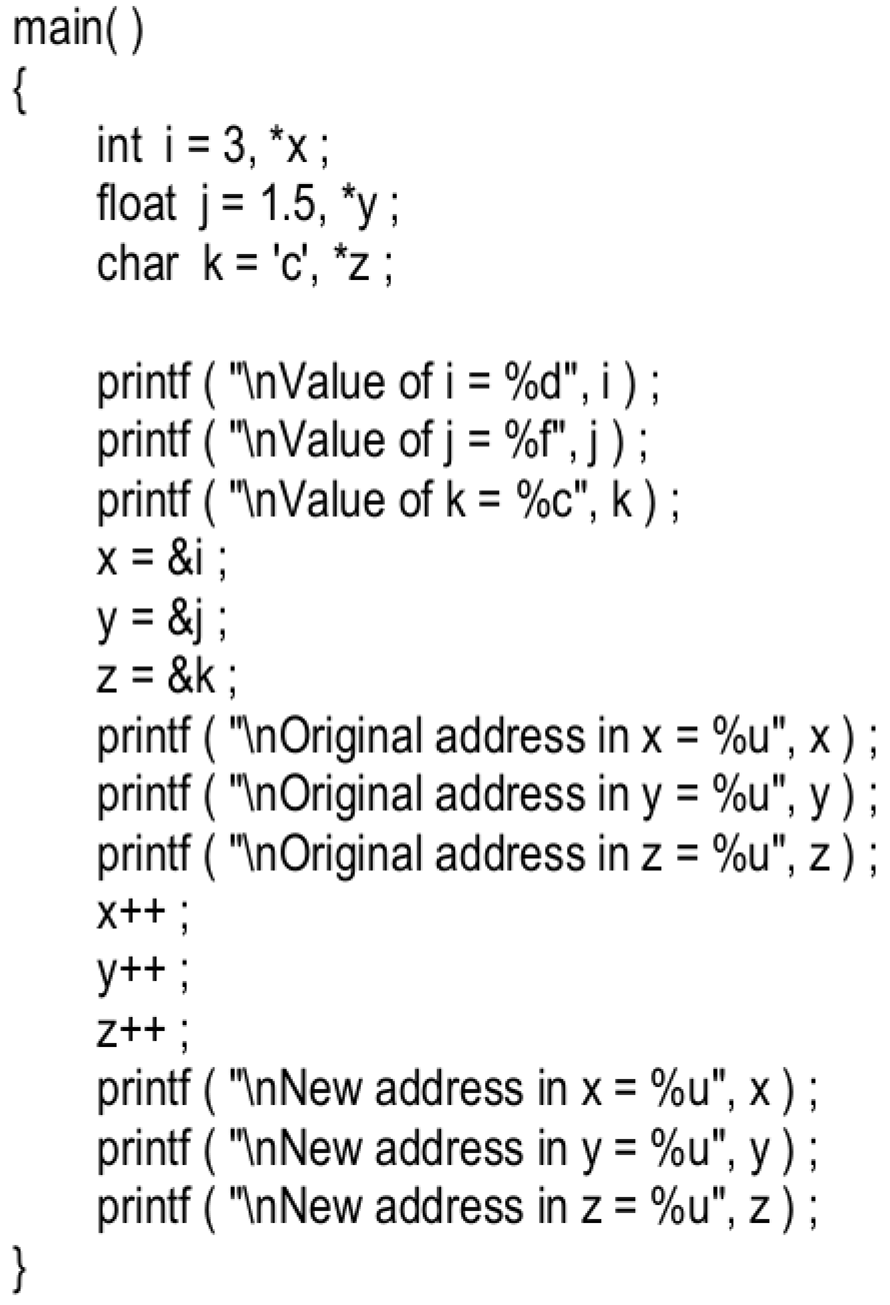
```
main( )
{
      int  i ;
      int  marks[ ] = { 55, 65, 75, 56, 78, 78, 90 } ;

      for ( i = 0 ; i <= 6 ; i++ )
            disp ( &marks[i] ) ;

 }

 disp ( int  *n )
 {
       show ( &n ) ;
 }
```

Pointers and Arrays

- To be able to see what pointers have got to do with arrays, let us first learn some pointer arithmetic.



```
Value of i = 3
Value of j = 1.500000
Value of k = c
Original address in x = 65524
Original address in y = 65520
Original address in z = 65519
New address in x = 65526
New address in y = 65524
New address in z = 65520
```



```
main( )
{
    int i = 3, *x ;
    float j = 1.5, *y ;
    char k = 'c', *z ;

    printf ( "\nValue of i = %d", i ) ;
    printf ( "\nValue of j = %f", j ) ;
    printf ( "\nValue of k = %c", k ) ;
    x = &i ;
    y = &j ;
    z = &k ;
    printf ( "\nOriginal address in x = %u", x ) ;
    printf ( "\nOriginal address in y = %u", y ) ;
    printf ( "\nOriginal address in z = %u", z ) ;
    x++ ;
    y++ ;
    z++ ;
    printf ( "\nNew address in x = %u", x ) ;
    printf ( "\nNew address in y = %u", y ) ;
    printf ( "\nNew address in z = %u", z ) ;
}
```

Addition and Subtraction to pointers

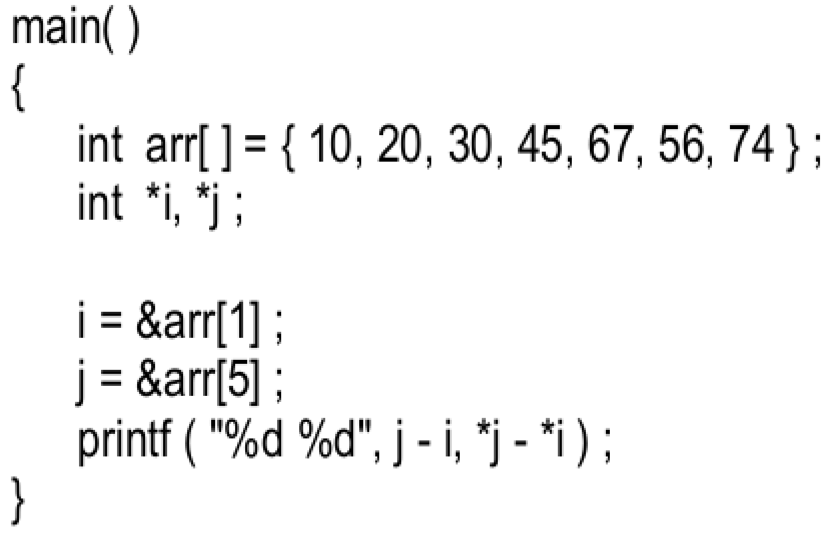
- The way a pointer can be incremented, it can be decremented as well, to point to earlier locations.
- Thus, the following operations can be performed on a pointer:
- Addition of a number to a pointer. For example,
- int i = 4, *j, *k ;
- j = &i ;
- j = j + 1 ;
- j = j + 9 ;
- k = j + 3 ;

Subtraction of a number

- Subtraction of a number from a pointer. For example,
- int i = 4, *j, *k ;
- j = &i ;
- j = j - 2 ;
- j = j - 5 ;
- k = j - 6 ;

Subtraction of one pointer from another

- One pointer variable can be subtracted from another provided both variables point to elements of the same array.
- The resulting value indicates the number of bytes separating the corresponding array elements.
- This is illustrated in the following program.



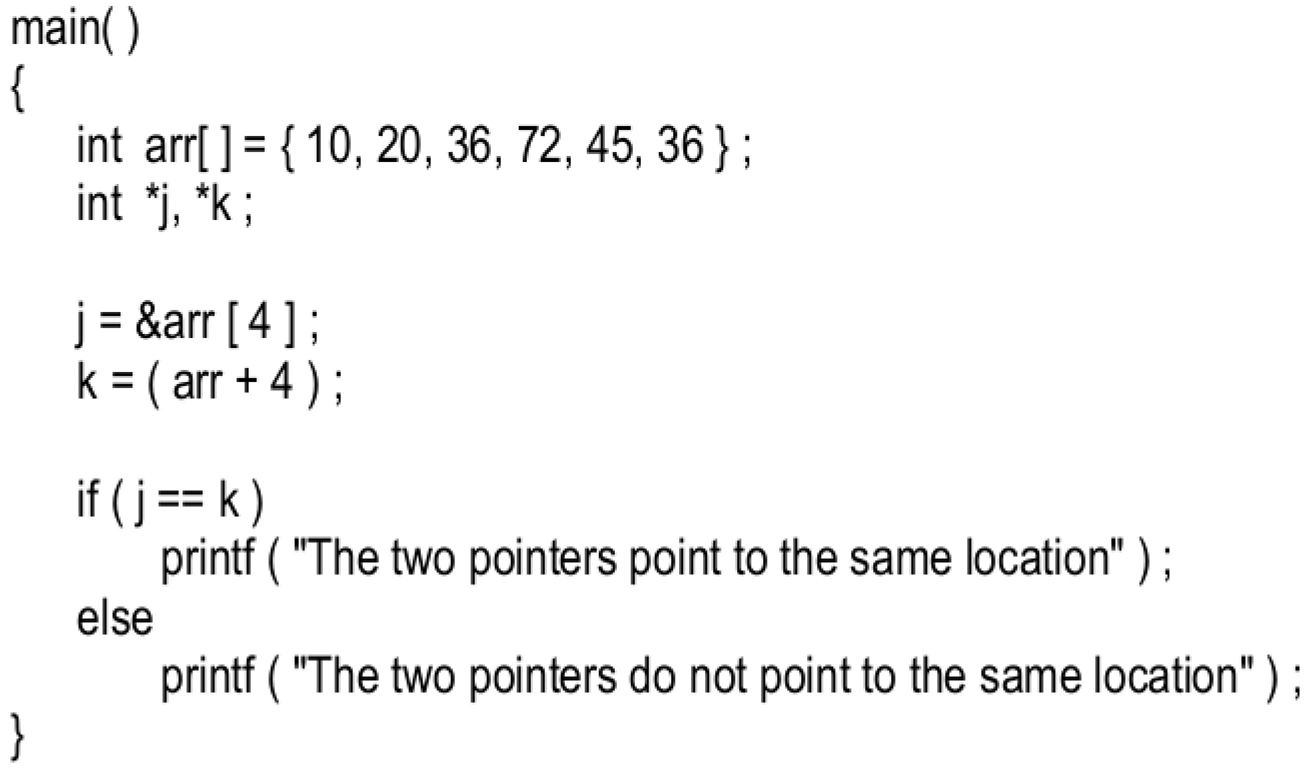
```
main( )
{
    int  arr[ ] = { 10, 20, 30, 45, 67, 56, 74 } ;
    int  *i, *j ;

    i = &arr[1] ;
    j = &arr[5] ;
    printf ( "%d %d", j - i, *j - *i ) ;
}
```

Solution

- Suppose the array begins at location 65502,
- then the elements arr[1] and arr[5] would be present at locations 65504 and 65512 respectively, since each integer in the array occupies two bytes in memory.
- The expression j - i would print a value 4 and not 8.
- This is because j and i are pointing to locations that are 4 integers apart.
- What would be the result of the expression *j - *i?
- 36, since *j and *i return the values present at addresses contained in the pointers j and i.

Comparison of two pointer variables

- Pointer variables can be compared provided both variables point to objects of the same data type.
- Such comparisons can be useful when both pointer variables point to elements of the same array.
- The comparison can test for either equality or inequality.
- Moreover, a pointer variable can be compared with zero (usually expressed as NULL).

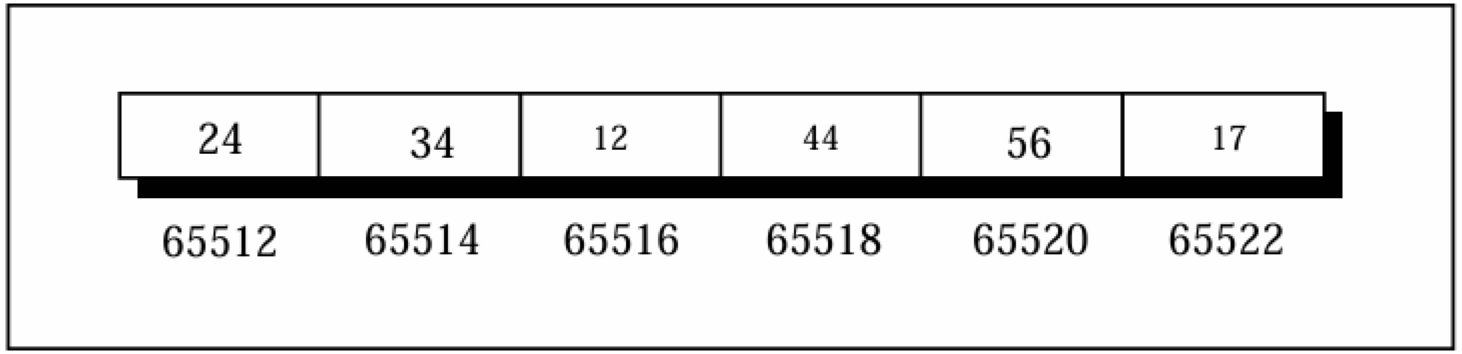


```
main( )
{
    int  arr[ ] = { 10, 20, 36, 72, 45, 36 } ;
    int  *j, *k ;

    j = &arr [ 4 ] ;
    k = ( arr + 4 ) ;

    if ( j == k )
        printf ( "The two pointers point to the same location" ) ;
    else
        printf ( "The two pointers do not point to the same location" ) ;
}
```

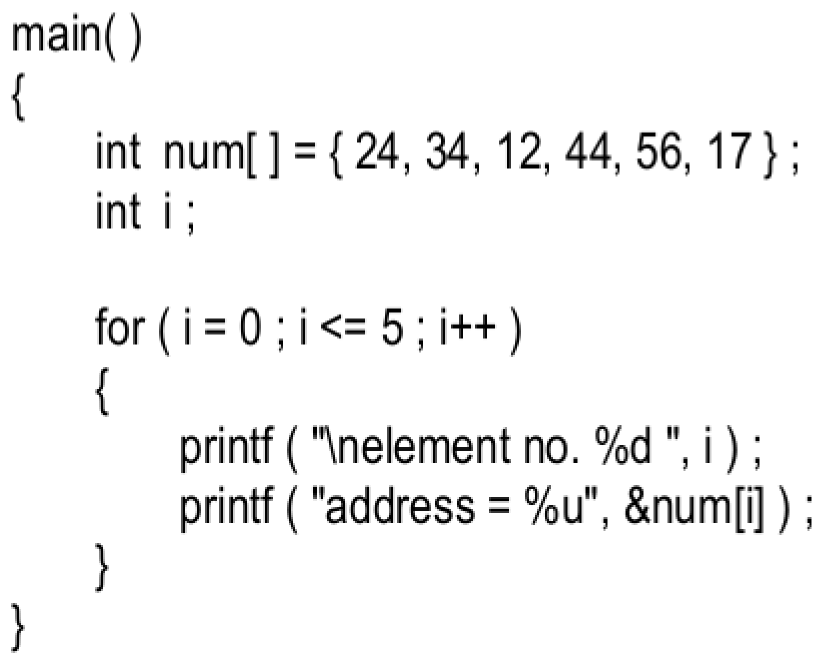
A word of caution!

- Do not attempt the following operations on pointers... they would never work out.
- Addition of two pointers
- Multiplication of a pointer with a constant
- Division of a pointer with a constant

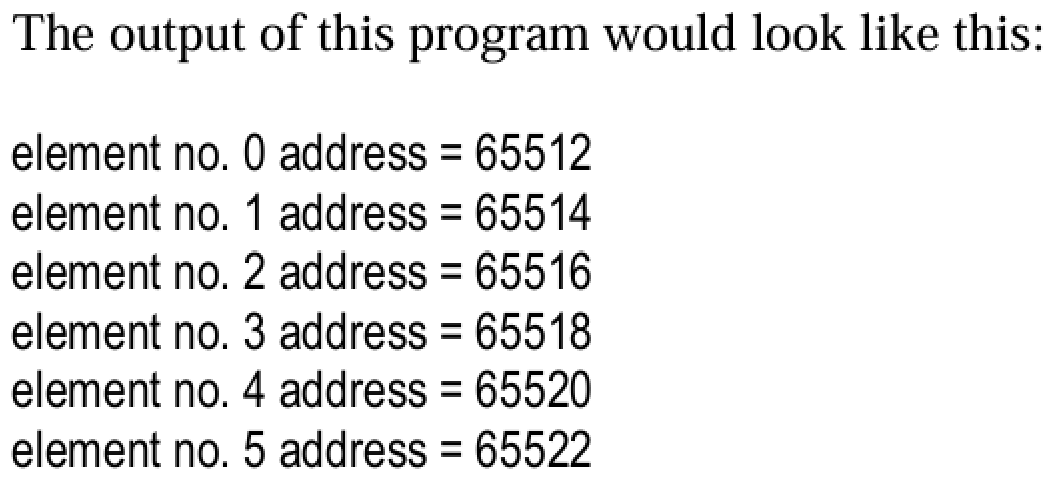
- Now we will try to correlate the following two facts, which we have learnt above:
- Array elements are always stored in contiguous memory locations.
- A pointer when incremented always points to an immediately next location of its type.
- Suppose we have an array num[ ] = { 24, 34, 12, 44, 56, 17 }.
- The following figure shows how this array is located in memory.



- Here is a program that prints out the memory locations in which the elements of this array are stored.



```
main( )
{
    int  num[ ] = { 24, 34, 12, 44, 56, 17 } ;
    int  i ;

    for ( i = 0 ; i <= 5 ; i++ )
    {
        printf ( "\nelement no. %d ", i ) ;
        printf ( "address = %u", &num[i] ) ;
    }
}
```



The output of this program would look like this:

```
element no. 0 address = 65512
element no. 1 address = 65514
element no. 2 address = 65516
element no. 3 address = 65518
element no. 4 address = 65520
element no. 5 address = 65522
```

```
main( )
{
    int  num[ ] = { 24, 34, 12, 44, 56, 17 } ;

    int  i, *j ;

    j = &num[0] ;  /* assign address of zeroth element */

    for ( i = 0 ; i <= 5 ; i++ )
    {
        printf ( "\naddress = %u ", j ) ;
        printf ( "element = %d", *j ) ;
        j++ ;  /* increment pointer to point to next location */
    }
}
```

The output of this program would be:

```
address = 65512 element = 24
address = 65514 element = 34
address = 65516 element = 12
address = 65518 element = 44
address = 65520 element = 56
address = 65522 element = 17
```

How it works

- In this program, to begin with we have collected the base address of the array (address of the 0th element) in the variable j
- When we are inside the loop for the first time, j contains the address 65512, and the value at this address is 24
- On incrementing j it points to the next memory location of its type (that is location no. 65514). But location no. 65514 contains the second element of the array, therefore when the printf( ) statements are executed for the second time they print out the second element of the array and its address (i.e. 34 and 65514)... and so on till the last element of the array has been printed.

Which method to use when

- Accessing array elements by pointers is always faster than accessing them by subscripts
- Array elements should be accessed using pointers if the elements are to be accessed in a fixed order, say from beginning to end, or from end to beginning, or every alternate element or any such definite logic.
- Instead, it would be easier to access the elements using a subscript if there is no fixed logic in accessing the elements.
- However, in this case also, accessing the elements by pointers would work faster than subscripts.

Passing an Entire Array to a Function

```
/* Demonstration of passing an entire array to a function */
main( )
{
      int  num[ ] = { 24, 34, 12, 44, 56, 17 } ;
      dislpay ( &num[0], 6 ) ;
}

display ( int  *j, int  n )
{

     int  i ;

     for ( i = 0 ; i <= n - 1 ; i++ )
     {
           printf ( "\nelement = %d", *j ) ;
           j++ ;  /* increment pointer to point to next element */
     }
}
```

Explanation

- Just passing the address of the zeroth element of the array to a function is as good as passing the entire array to the function.It is also necessary to pass the total number of elements in the array, otherwise the display( ) function would not know when to terminate the for loop.
- Note that the address of the zeroth element (many a times called the base address) can also be passed by just passing the name of the array.
- Thus, the following two function calls are same:
- display (&num[0], 6) ;
- display (num, 6) ;

The Real Thing

- This is how we would declare the above array in C,
- int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
- We also know that on mentioning the name of the array we get its base address.
- Thus, by saying *num we would be able to refer to the zeroth element of the array, that is, 24.
- One can easily see that *num and *(num + 0) both refer to 24.
- Similarly, by saying *(num + 1) we can refer the first element of the array, that is, 34.

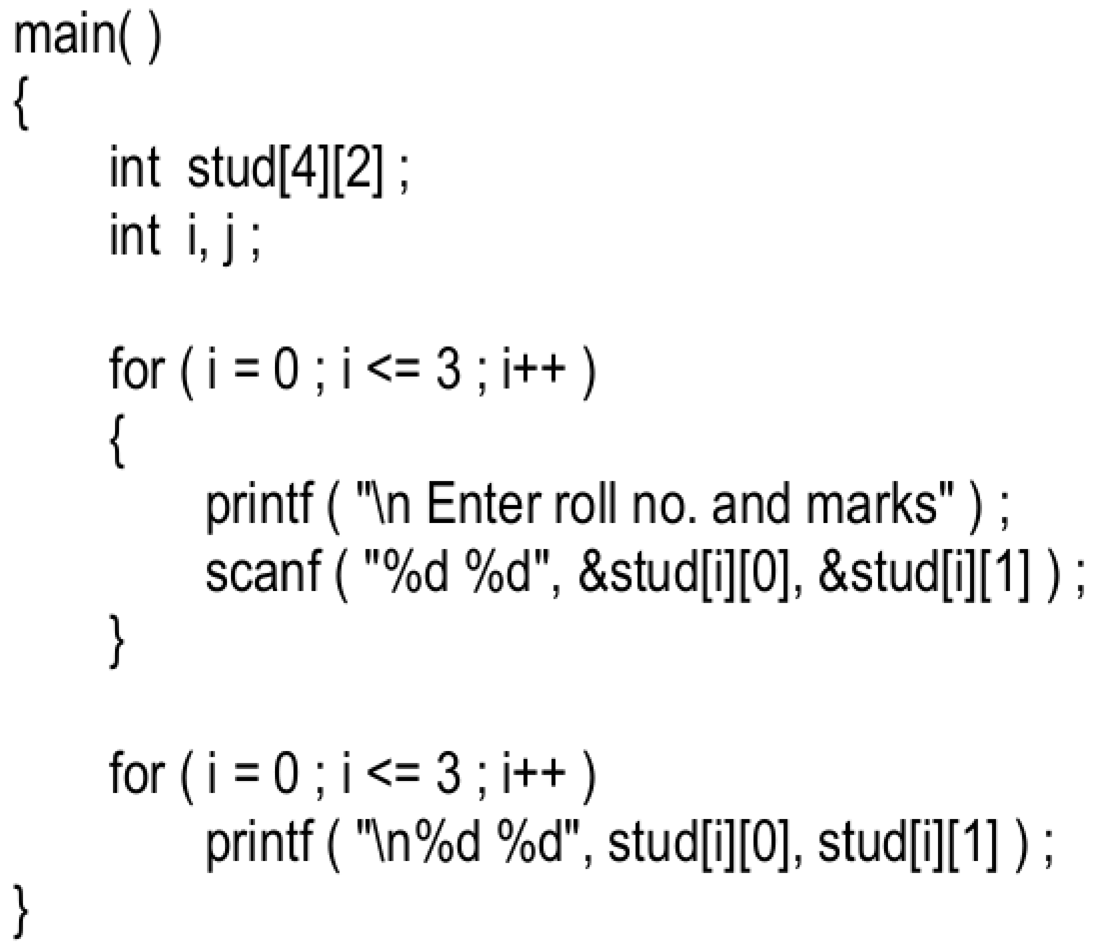
- In fact, this is what the C compiler does internally.
- When we say, num[i], the C compiler internally converts it to *(num + i).
- This means that all the following notations are same:
- num[i]
- *(num + i)
- *(i + num)
- i[num]

```
/* Accessing array elements in different ways */
main( )
{
    int  num[ ] = { 24, 34, 12, 44, 56, 17 } ;
    int  i ;

    for ( i = 0 ; i <= 5 ; i++ )
    {
        printf ( "\naddress = %u ", &num[i] ) ;
        printf ( "element = %d %d ", num[i], *( num + i ) ) ;
        printf ( "%d %d", *( i + num ), i[num] ) ;
    }
}
```

```
address = 65512 element = 24 24 24 24
address = 65514 element = 34 34 34 34
address = 65516 element = 12 12 12 12
address = 65518 element = 44 44 44 44
address = 65520 element = 56 56 56 56
address = 65522 element = 17 17 17 17
```

Two Dimensional Arrays

- It is also possible for arrays to have two or more dimensions.
- The two dimensional array is also called a matrix.
-



```
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

Storage of Elements in the 2D array

- 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 rowwise and accessed rowwise.
- However, you can access the array elements columnwise as well. Traditionally, the array elements are being stored and accessed rowwise; therefore we would also stick to the same strategy.

Initialising a 2-Dimensional Array

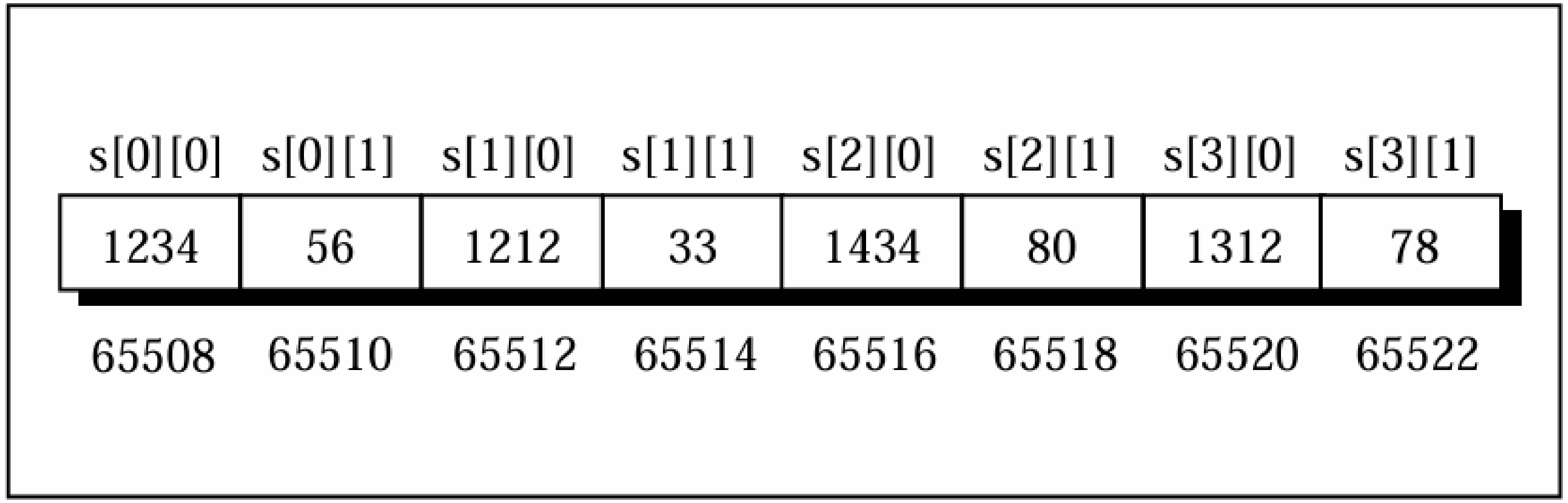
- int stud[4][2] = {
```
          {1234, 56 },
          {1212, 33 },
          {1434, 80 },
          {1312, 78 }
 };
```
- int stud[4][2] = { 1234, 56, 1212, 33, 1434, 80, 1312, 78 } ;

Declaration

- 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.
- 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 2-Dimensional Array

- The array arrangement shown in Figure 8.4 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 below:


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

- 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]) ;
- Pointers can also be used for this as we will see next

Pointers and 2-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.

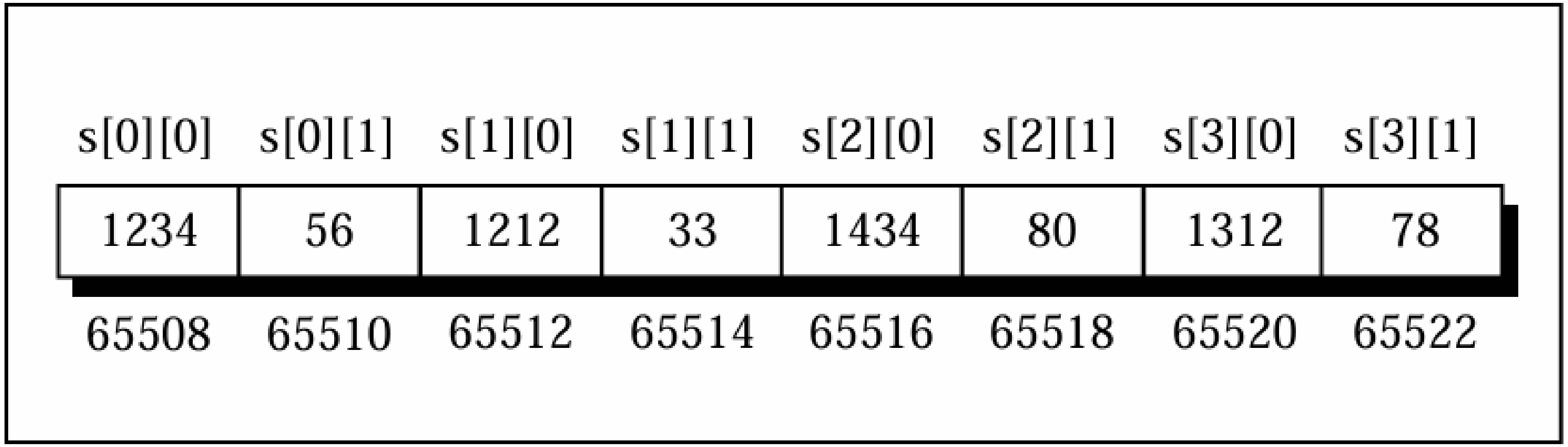
Pointers and 2-Dimensional Arrays

- 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 */
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] ) ;
}
```

And here is the 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
```


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

- 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 these addresses turn out to be 65508 and 65512.
- Now, we have been able to reach each 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 65516, the address of the second one-dimensional array.
- Obviously (65516 + 1) would give the address 65518.
- Or (s[2] + 1) would give the address 65518.

- 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)

```
/* Pointer notation to access 2-D array elements */
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 ) ) ;
    }
}
```

And here is the output...

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

Pointer to an Array

```
/* Usage of pointer to an array */
main( )
{
    int  s[5][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 ) ) ;
    }
}
```

And here is the output...

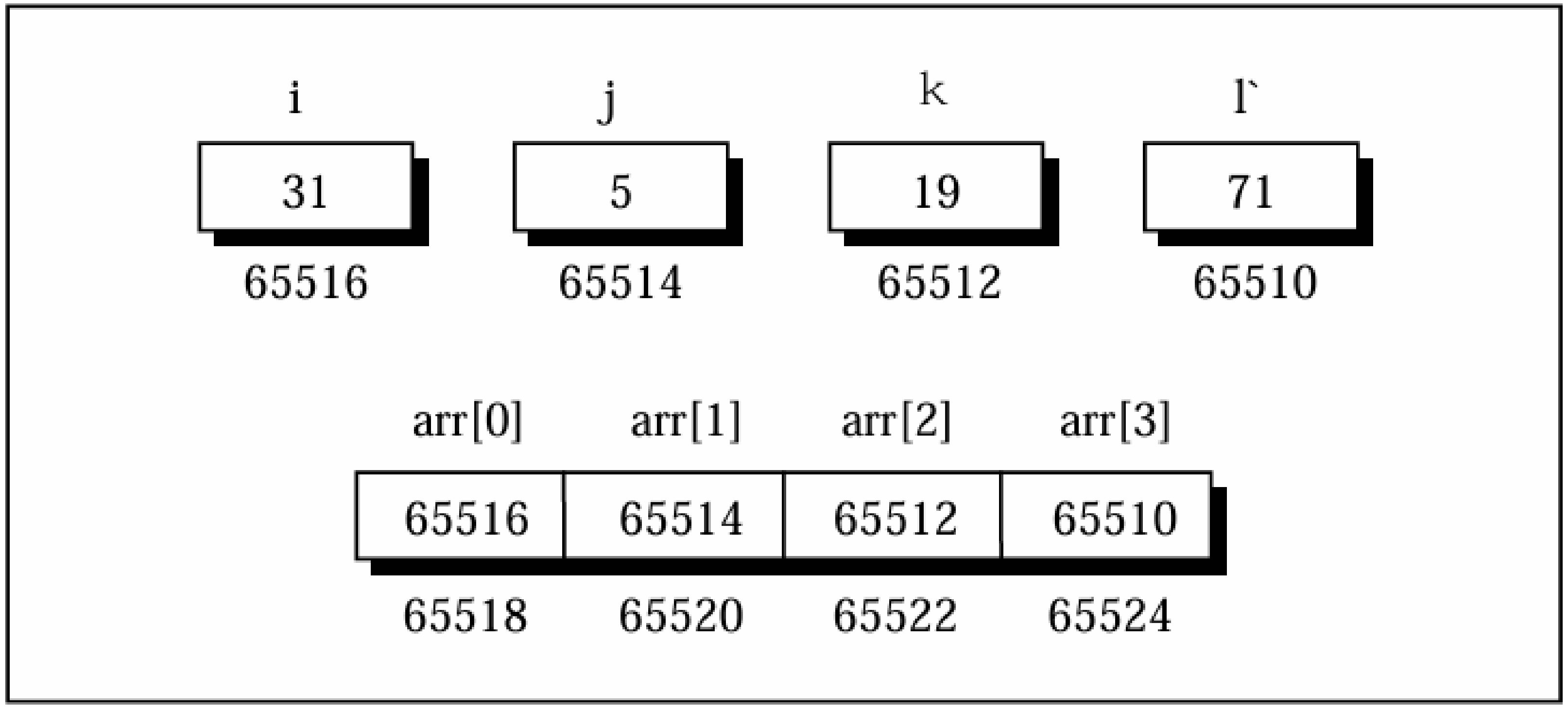
```
1234  56
1212  33
1434  80
1312  78
```

- 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.
- 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.
- The entity pointer to an array is immensely useful when we need to pass a 2-D array to a function.

Array of Pointers

- The way there can be an array of ints or an array of floats, 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.

```
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] ) ) ;
}
```

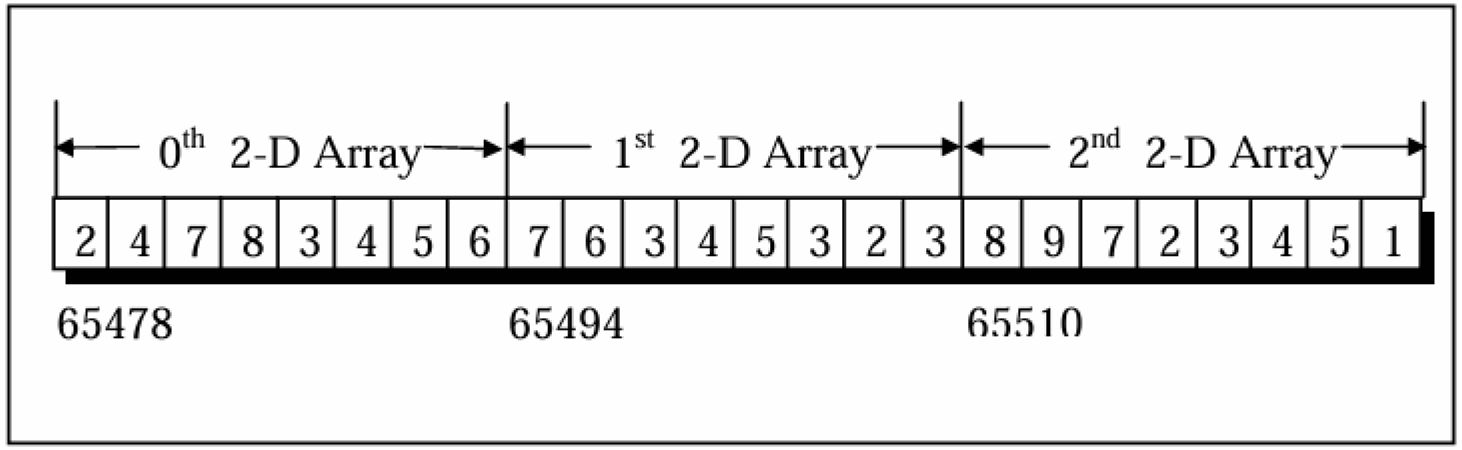

i
j
k
l`
31
5
19
71
65516
65514
65512
65510
arr[0]
arr[1]
arr[2]
arr[3]
65516
65514
65512
65510
65518
65520
65522
65524

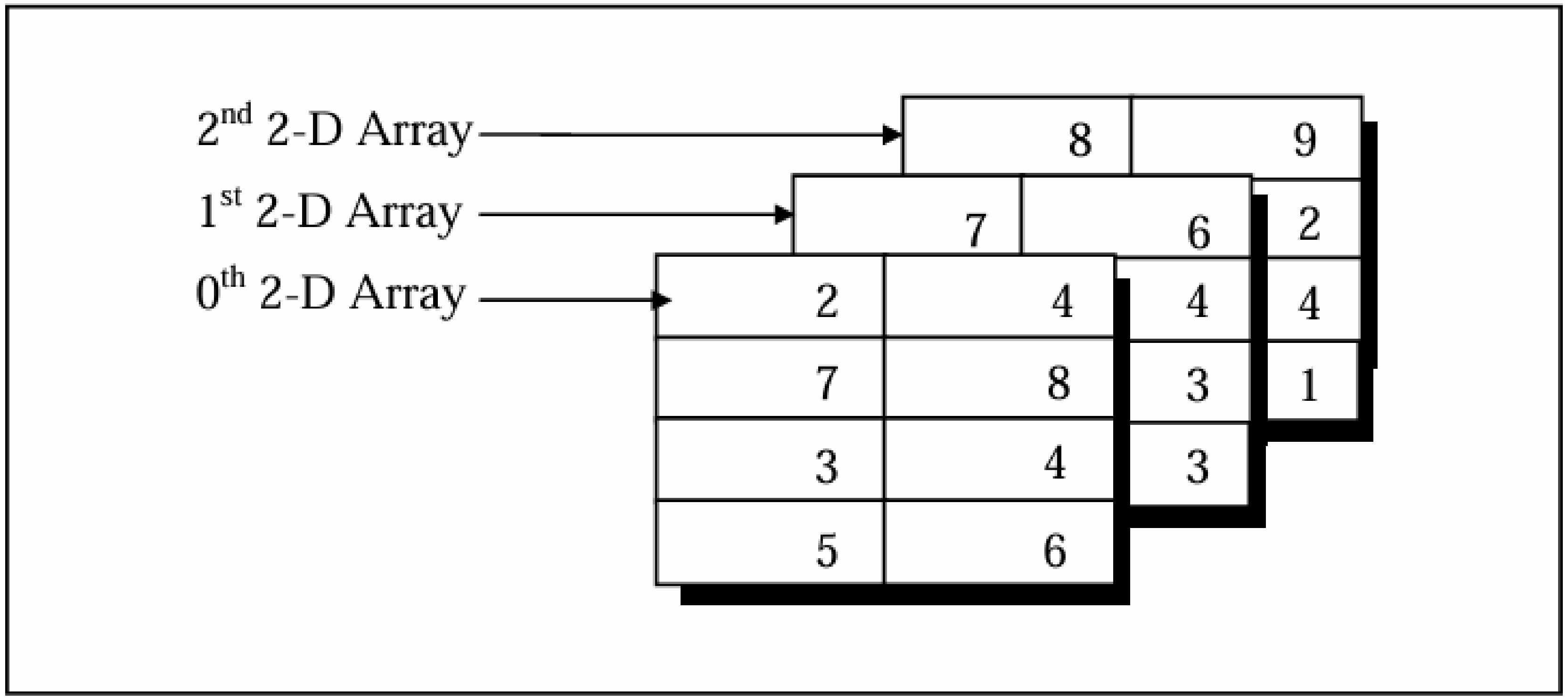
```
int  arr[3][4][2] = {
                        {
                              { 2, 4 },
                              { 7, 8 },
                              { 3, 4 },
                              { 5, 6 }
                        },
                        {
                              { 7, 6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

Three-Dimensional Array

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

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




2nd 2-D Array
1st 2-D Array
0th 2-D Array
8
9
2
4
1
7
6
4
3
3
2
4
7
8
3
4
5
6