Here, we are passing addresses of individual array elements to the function **display( )**. Hence, the variable in which this address is collected **(n)** must be a pointer variable. And since **n** contains the address of array element, to print out the array element we must use the 'value at address' operator **(*)**.

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

```
/* Program 14 */
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

// Read from here.

To be able to see what pointers have got to do with arrays, let us first learn some pointer arithmetic. Consider the following example:

```
/* Program 15 */
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 ( "\n\nOriginal value in x = %u", x ) ;
    printf ( "\nOriginal value in y = %u", y ) ;
    printf ( "\nOriginal value in z = %u", z ) ;

    x++ ;
    y++ ;
    z++ ;

    printf ( "\n\nNew value in x = %u", x ) ;
    printf ( "\nNew value in y = %u", y ) ;
    printf ( "\nNew value in z = %u", z ) ;
}
```

Suppose **i**, **j** and **k** are stored in memory at addresses 1002, 2004 and 5006, the output would be...

```
Value of i = 3
Value of j = 1.500000
Value of k = c

Original value in x = 1002
Original value in y = 2004
Original value in z = 5006

New value in x = 1004
New value in y = 2008
New value in z = 5007
```

The size of different data types as mentioned here are for Turbo C. It will vary depending on the compiler you are using.

Observe the last three lines of the output. 1004 is equal to original value in **x** plus 2, 2008 is equal to original value in **y** plus 4, and 5007 is equal to original value in **z** plus 1. This so happens because every time a pointer is incremented it points to the immediately next location of its type. That is why, when the integer pointer **x** is incremented, it points to an address two locations after the current location, since an **int** is always 2 bytes long. Similarly, **y** points to an address 4 locations after the current location and **z** points 1 location after the current location. This is a very important result and can be effectively used while passing the entire array to a function.

Rules of Pointer Arithmetic:

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:

(a) 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 ;
```

(b) 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 ;
```

A word of caution! Do not attempt the following operations on pointers... they would never work out.

(a) Addition of two pointers
(b) Multiplying a pointer with a number
(c) Dividing a pointer with a number

Now we will try to correlate the following two facts, which we have already learnt:

(a) Array elements are always stored in contiguous memory locations.

(b) A pointer when incremented always points to an immediately next location of its type.

Suppose we have an array,

```
int num[ ] = { 23, 34, 12, 44, 56, 17 } ;
```

The following figure shows how this array is located in memory.

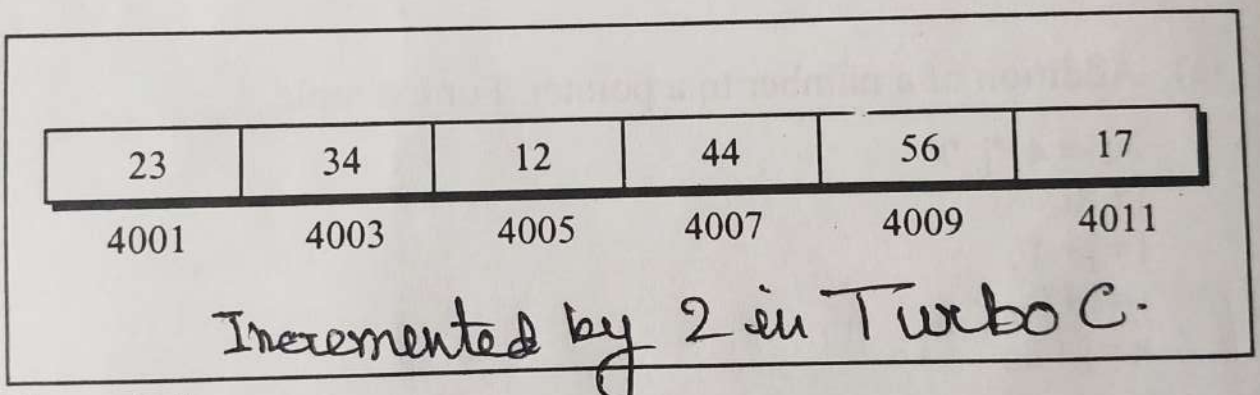

Figure 2.1

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

```
/* Program 16 */
main( )
{
    int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
    int i = 0 ;

    while ( i <= 5 )
    {
        printf ( "\nelement no. %d ", i ) ;
```

```
        printf ( "address = %u" , &num[i] ) ;
        i++ ;
    }
}
```

The output of this program would be:

```
element no. 0   address = 4001
element no. 1   address = 4003
element no. 2   address = 4005
element no. 3   address = 4007
element no. 4   address = 4009
element no. 5   address = 4011
```

Note that the array elements are stored in contiguous memory locations, each element occupying two bytes, since it is an integer array. When you run this program, you may get different addresses, but what is promised is that each subsequent address would be 2 bytes greater than its immediate predecessor.

Our next two programs show two ways in which we can access the elements of this array. The first one uses the subscript notation.

```
/* Program 17 */
main( )
{
    int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
    int i = 0 ;

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

The output of this program would be:

```
address = 4001   element = 24
address = 4003   element = 34
address = 4005   element = 12
address = 4007   element = 44
address = 4009   element = 56
address = 4011   element = 17
```

The next method accesses the array elements using pointers.

```
/* Program 18 */
main( )
{
    int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
    int i = 0, *j ;

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

    while ( i <= 5 )
    {
        printf ( "\naddress = %u    " , &num[i] ) ;
        printf ( "element = %d", *j ) ;

        i++ ;
        j++ ;  /* increment pointer to point to next location */
    }
}
```

The output of the program would look like this:

```
address = 4001   element = 24
address = 4003   element = 34
address = 4005   element = 12
address = 4007   element = 44
address = 4009   element = 56
address = 4011   element = 17
```

In this program, to begin with we have collected the base address of the array (address of 0^{th} element) in the variable **j** using the statement,

```
j = &num[0] ;  /* assigns address 4001 to j */
```

When we are inside the loop for the first time **j** contains the address 4001, and the value at this address is 24. These are printed using the statements,

```
printf ( "\naddress = %u    " , &num[i] ) ;
printf ( "element = %d", *j ) ;
```

On incrementing **j** it points to the next memory location of its type, that is location no. 4003. But location number 4003 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 4003). This continues till the last element of the array has been printed.

Obviously, a question arises as to which of the above two methods should be used when? Accessing array elements using pointers is **always** faster than accessing them by subscripts. However, from the point of view of convenience in programming we should observe the following:

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.

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.