8 Arrays

- What are Arrays
  - A Simple Program Using Array
- More on Arrays
  - Array Initialisation
  - Bounds Checking
  - Passing Array Elements to a Function
- Pointers and Arrays
  - Passing an Entire Array to a Function
  - The Real Thing
- Two Dimensional Arrays
  - Initialising a 2-Dimensional Array
  - Memory Map of a 2-Dimensional Array
  - Pointers and 2-Dimensional Arrays
  - Pointer to an Array
  - Passing 2-D Array to a Function
- Array of Pointers
- Three-Dimensional Array
- Summary
- Exercise

The C language provides a capability that enables the user to design a set of similar data types, called array. This chapter describes how arrays can be created and manipulated in C.

We should note that, in many C books and courses arrays and pointers are taught separately. I feel it is worthwhile to deal with these topics together. This is because pointers and arrays are so closely related that discussing arrays without discussing pointers would make the discussion incomplete and wanting. In fact all arrays make use of pointers internally. Hence it is all too relevant to study them together rather than as isolated topics.

What are Arrays

For understanding the arrays properly, let us consider the following program:

```
main( )
{
    int x ;
    x = 5 ;
    x = 10 ;
    printf ( "\nx = %d", x ) ;
}
```

No doubt, this program will print the value of **x** as 10. Why so? Because when a value 10 is assigned to x, the earlier value of **x**, i.e. 5, is lost. Thus, ordinary variables (the ones which we have used so far) are capable of holding only one value at a time (as in the above example). 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. In such a case we have two options to store these marks in memory:

(a) Construct 100 variables to store percentage marks obtained by 100 different students, i.e. each variable containing one student's marks.

(b) Construct one variable (called array or subscripted variable) capable of storing or holding all the hundred values.

Obviously, the second alternative is better. A simple reason for this is, 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. Now a formal definition of an array—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 }

If we want to refer to the second number of the group, the usual notation used is per_2. Similarly, the fourth number of the group is referred as per_4. However, in C, the fourth number is referred as **per[3]**. This is because in C the counting of elements begins with 0 and not with 1. 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. Here **per** is the subscripted variable (array), whereas **i** is its subscript.

Thus, an array is a collection of similar elements. These similar elements could be all **int**s, or all **float**s, or all **char**s, etc. Usually, the array of characters is called a 'string', whereas an array of **int**s or **float**s is called simply an array. 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 **int**s and 5 are **float**s.

A Simple Program Using Array

Let us try to write a program to find average marks obtained by a class of 30 students in a test.

```
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 ) ;
}
```

There is a lot of new material in this program, so let us take it apart slowly.

Array Declaration

To begin with, like other variables an array needs to be declared so that the compiler will know what kind of an array and how large an array we want. In our program we have done this with the statement:

```
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 **[30]** however is new. 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

Once an array is declared, let us see how individual elements in the array can be referred. This is done with subscript, 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

Here is the section of code that places data into an array:

```
for ( i = 0 ; i <= 29 ; i++ )
{
    printf ( "\nEnter marks " ) ;
    scanf ( "%d", &marks[i] ) ;
}
```

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. This is last time through the loop, which is a good thing, because there is no array element like **marks[30]**.

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.

```
for ( i = 0 ; i <= 29 ; i++ )
    sum = sum + marks[i] ;

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

To fix our ideas, let us revise whatever we have learnt about arrays:

(a) An array is a collection of similar elements.
(b) The first element in the array is numbered 0, so the last element is 1 less than the size of the array.
(c) An array is also known as a subscripted variable.
(d) Before using an array its type and dimension must be declared.
(e) 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 is a very popular data type with C programmers. This is because of the convenience with which arrays lend themselves to programming. The features which make arrays so convenient to program would be discussed below, along with the possible pitfalls in using them.

Array Initialisation

So far we have used arrays that did not have any values in them to begin with. We managed to store values in them during program execution. Let us now see how to initialize an array while declaring it. 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 } ;
```

Note the following points carefully:

(a) Till the array elements are not given any specific values, they are supposed to contain garbage values.

(b) 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. Whatever be the initial values, all the array elements would always be present in contiguous memory locations. This arrangement of array elements in memory is shown in Figure 8.1.

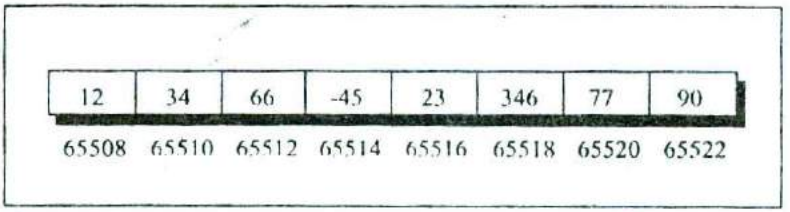


Figure 8.1

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.

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

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

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.

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
```

Here, we are passing an individual array element at a time to the function **display( )** and getting it printed in the function **display( ).** Note that since at a time only one element is being passed, this element is collected in an ordinary integer variable **m**, in the function **display( )**.

And now the call by reference.

```
/* 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
```

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

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. Try your hand at it.

```
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. Consider the following example:

```
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 ) ;
}
```

Here is the output of the program.

```
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
```

Observe the last three lines of the output. 65526 is original value in **x** plus 2, 65524 is original value in **y** plus 4, and 65520 is 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 (under Windows/Linux since **int** is 4 bytes long, new value of **x** would be 65528). 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.

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 ;
```

(c) 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 ) ;
}
```

Here **i** and **j** have been declared as integer pointers holding addresses of first and fifth element of the array respectively.

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**.

(d) 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). The following program illustrates how the comparison is carried out.

```
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.

(a) Addition of two pointers
(b) Multiplication of a pointer with a constant
(c) Division of a pointer with a constant

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

(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 **num[ ]** = { 24, 34, 12, 44, 56, 17 }. The following figure shows how this array is located in memory.

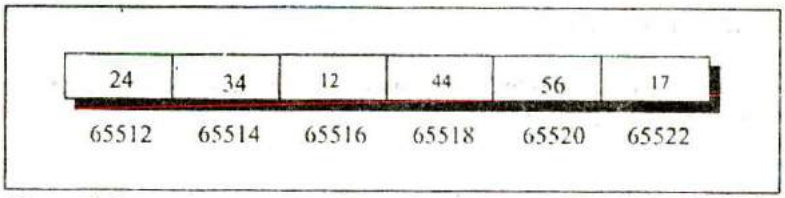


Figure 8.2

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
```

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 certain is that each subsequent address would be 2 bytes (4 bytes under Windows/Linux) greater than its immediate predecessor.

Our next two programs show ways in which we can access the elements of this array.

```
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", num[i] ) ;
    }
}
```

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
```

This method of accessing array elements by using subscripted variables is already known to us. This method has in fact been given here for easy comparison with the next method, which accesses the array elements using pointers.

```
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
```

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

```
j = &num[0] ;  /* assigns address 65512 to j */
```

When we are inside the loop for the first time, **j** contains the address 65512, and the value at this address is 24. These are printed using the statements,

```
printf ( "\naddress = %u ", j ) ;
printf ( "element = %d", *j ) ;
```

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.

Obviously, a question arises as to which of the above two methods should be used when? Accessing array elements by 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.

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

In the previous section we saw two programs—one in which we passed individual elements of an array to a function, and another in which we passed addresses of individual elements to a function. Let us now see how to pass an entire array to a function rather than its individual elements. Consider the following example:

```
/* 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 */
    }
}
```

Here, the **display( )** function is used to print out the array elements. Note that the address of the zeroth element is being passed to the **display( )** function. The **for** loop is same as the one used in the earlier program to access the array elements using pointers. Thus, 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

If you have grasped the concept of storage of array elements in memory and the arithmetic of pointers, here is some real food for thought. Once again consider the following array.

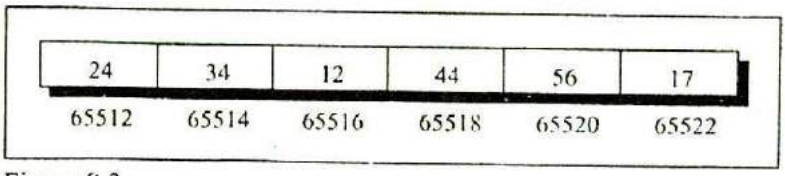


Figure 8.3

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:

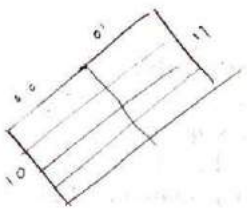
```
num[i]
*( num + i )
*( i + num )
i[num]
```

And here is a program to prove my point.

```
/* 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] ) ;
    }
}
```

The output of this program would be:

```
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

So far we have explored arrays with only one dimension. It is also possible for arrays to have two or more dimensions. The two-dimensional array is also called a matrix.

Here is a sample program that stores roll number and marks obtained by a student side by side in a matrix.

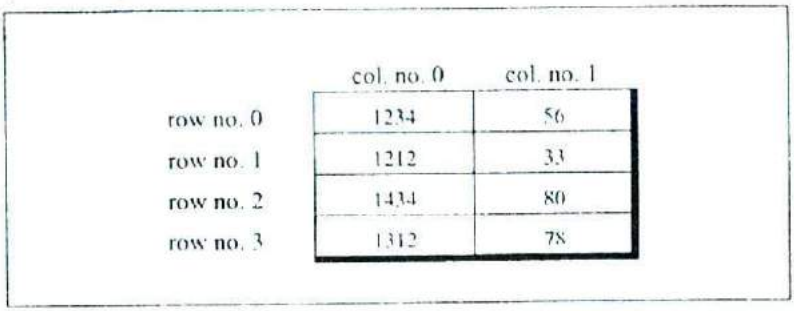
```
main( )
{
    int stud[4][2] ;
    int i, j ;

    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] ) ;
}
```

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 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] ) ;
```

In **stud[i][0]** and **stud[i][1]** the first subscript of the variable **stud**, is row number which changes for every student. The second subscript tells which of the two columns are we talking about—the zeroth column which contains the roll no. or the first column which contains the marks. Remember the counting of rows and columns begin with zero. The complete array arrangement is shown below.



	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

Figure 8.4

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

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.

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

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 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:

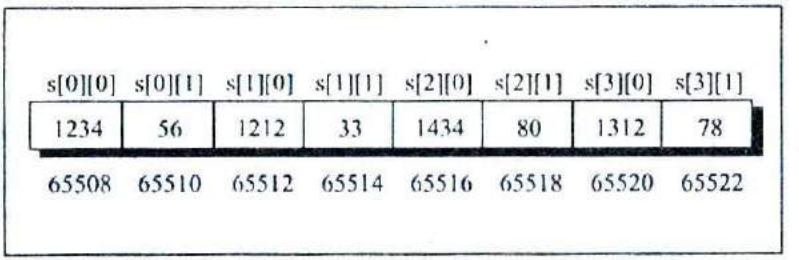


Figure 8.5

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 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...

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. 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
```

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 arrays starts 4 bytes further along than the last one, as can be seen in the memory map of the array shown below.

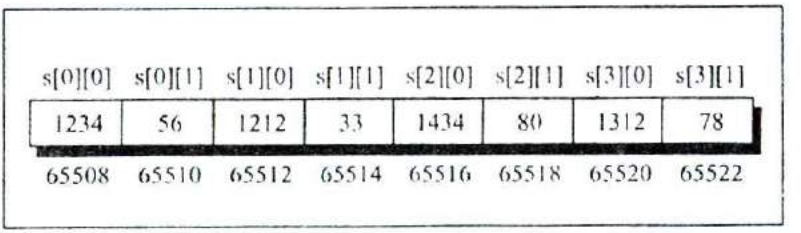


Figure 8.6

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 8.6 these addresses turn out to be 65508 and 65512.

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 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 )
```

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 */
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

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 */
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. 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.

Passing 2-D Array to a Function

There are three ways in which we can pass a 2-D array to a function. These are illustrated in the following program.

```
/* Three ways of accessing a 2-D array */

main( )
{
    int a[3][4] = {
                      1, 2, 3, 4,
                      5, 6, 7, 8,
                      9, 0, 1, 6
                  };

    clrscr( ) ;
    display ( a, 3, 4 ) ;
    show ( a, 3, 4 ) ;
    print ( a, 3, 4 ) ;
}
display ( int *q, int row, int col )
{
    int i, j ;

    for ( i = 0 ; i < row ; i++ )
    {
        for ( j = 0 ; j < col ; j++ )
            printf ( "%d ", * ( q + i * col + j ) ) ;
```

```
        printf ( "\n" ) ;
    }
    printf ("\n" ) ;
}

show ( int ( *q )[4], int row, int col )
{
    int i, j ;
    int *p ;

    for ( i = 0 ; i < row ; i++ )
    {
        p = q + i ;
        for ( j = 0 ; j < col ; j++ )
            printf ( "%d ", * ( p + j ) ) ;

        printf ( "\n" ) ;
    }
    printf ( "\n" ) ;
}

print ( int q[ ][4], int row, int col )
{
    int i, j ;

    for ( i = 0 ; i < row ; i++ )
    {
        for ( j = 0 ; j < col ; j++ )
            printf ( "%d ", q[i][j] ) ;
        printf ( "\n" ) ;
    }
    printf ( "\n" ) ;
}
```

And here is the output...

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

```
9016

1234
5678
9016

1234
5678
9016
```

In the **display( )** function we have collected the base address of the 2-D array being passed to it in an ordinary **int** pointer. Then through the two **for** loops using the expression *** (q + i * col + j)** we have reached the appropriate element in the array. Suppose **i** is equal to 2 and **j** is equal to 3, then we wish to reach the element **a[2][3]**. Let us see whether the expression *** (q + i * col + j)** does give this element or not. Refer Figure 8.7 to understand this.

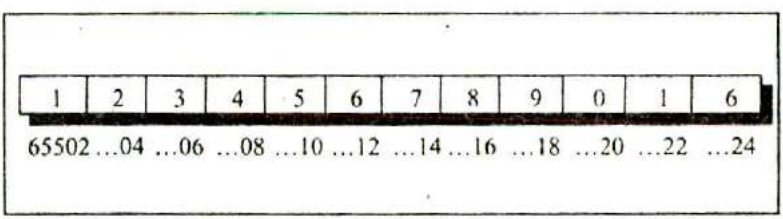


Figure 8.7

The expression *** (q + i * col + j)** becomes *** (65502 + 2 * 4 + 3)**. This turns out to be *** (65502 + 11).** Since **65502** is address of an integer, *** (65502 + 11)** turns out to be *** (65524)**. Value at this address is 6. This is indeed same as **a[2][3]**. A more general formula for accessing each array element would be:

* (base address + row no. * no. of columns + column no.)

In the **show( )** function we have defined **q** to be a pointer to an array of 4 integers through the declaration:

```
int ( *q )[4] ;
```

To begin with, **q** holds the base address of the zeroth 1-D array, i.e. 4001 (refer Figure 8.7). This address is then assigned to **p**, an **int** pointer, and then using this pointer all elements of the zeroth 1-D array are accessed. Next time through the loop when **i** takes a value 1, the expression **q + i** fetches the address of the first 1-D array. This is because, **q** is a pointer to zeroth 1-D array and adding 1 to it would give us the address of the next 1-D array. This address is once again assigned to **p**, and using it all elements of the next 1-D array are accessed.

In the third function **print( )**, the declaration of **q** looks like this:

```
int q[ ][4] ;
```

This is same as **int (*q)[4]**, where **q** is pointer to an array of 4 integers. The only advantage is that we can now use the more familiar expression **q[i][j]** to access array elements. We could have used the same expression in **show( )** as well.

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.

```
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] ) ) ;
}
```

Figure 8.8 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.

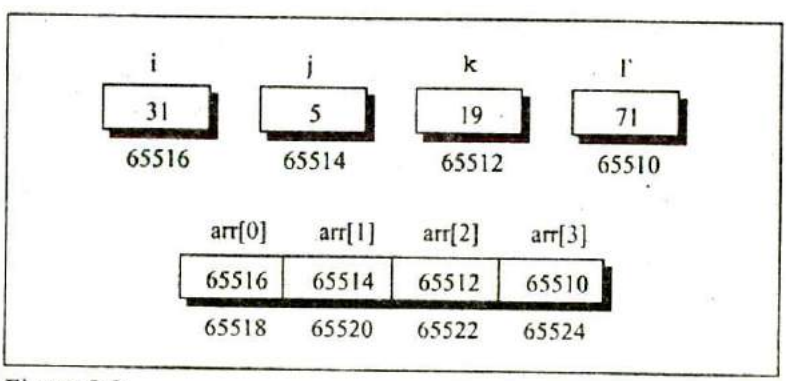


Figure 8.8

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

```
main( )
{
    static int a[ ] = { 0, 1, 2, 3, 4 } ;
```

```
    int *p[ ] = { a, a + 1, a + 2, a + 3, a + 4 };

    printf ( "\n%u %u %d", p, *p, * ( *p ) );
}
```

I would leave it for you to figure out the output of this program.

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 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. In the array declaration note how the commas have been given. Figure 8.9 would possibly help you in visualising the situation better.

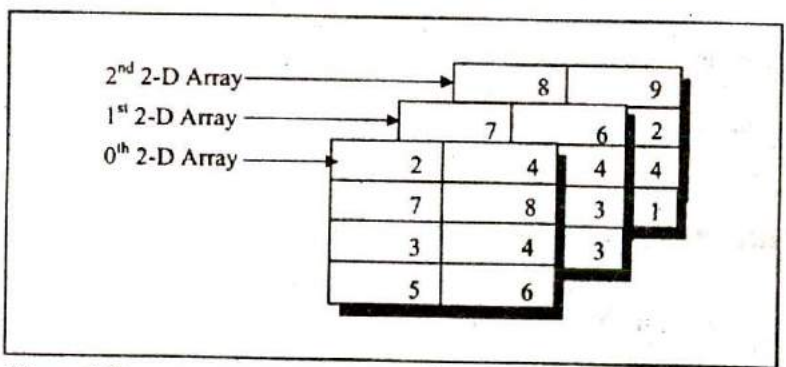


Figure 8.9

Again remember that the arrangement shown above is only conceptually true. In memory the same array elements are stored linearly as shown in Figure 8.10.

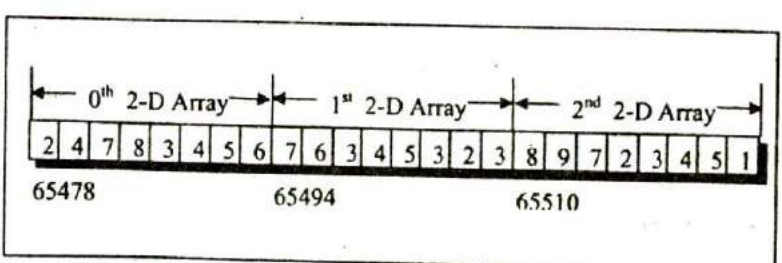


Figure 8.10

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 )
```

Summary

(a) An array is similar to an ordinary variable except that it can store multiple elements of similar type.
(b) Compiler doesn't perform bounds checking on an array.
(c) The array variable acts as a pointer to the zeroth element of the array. In a 1-D array, zeroth element is a single value, whereas, in a 2-D array this element is a 1-D array.
(d) On incrementing a pointer it points to the next location of its type.
(e) Array elements are stored in contiguous memory locations and so they can be accessed using pointers.
(f) Only limited arithmetic can be done on pointers.

Exercise

Simple arrays

[A] What would be the output of the following programs:

(a) main( )

```
{
    int num[26], temp ;
    num[0] = 100 ;
    num[25] = 200 ;
    temp = num[25] ;
    num[25] = num[0] ;
    num[0] = temp ;
    printf ( "\n%d %d", num[0], num[25] ) ;
}
```

(b)
```
main( )
{
    int array[26], i ;
    for ( i = 0 ; i <= 25 ; i++ )
    {
        array[i] = 'A' + i ;
        printf ( "\n%d %c", array[i], array[i] ) ;
    }
}
```

(c)
```
main( )
{
    int sub[50], i ;
    for ( i = 0 ; i <= 48 ; i++ ) ;
    {
        sub[i] = i ;
        printf ( "\n%d", sub[i] ) ;
    }
}
```

[B] Point out the errors, if any, in the following program segments:

(a)
```
/* mixed has some char and some int values */
int char mixed[100] ;

main( )
{
    int a[10], i ;
```

```
    for ( i = 1 ; i <= 10 ; i++ )
    {
        scanf ( "%d", a[i] ) ;
        printf ( "%d", a[i] ) ;
    }
}
```

(b)
```
main( )
{
    int size ;
    scanf ( "%d", &size ) ;
    int arr[size] ;
    for ( i = 1 ; i <= size ; i++ )
    {
        scanf ( "%d", arr[i] ) ;
        printf ( "%d", arr[i] ) ;
    }
}
```

(c)
```
main( )
{
    int i, a = 2, b = 3 ;
    int arr[ 2 + 3 ] ;
    for ( i = 0 ; i < a+b ; i++ )
    {
        scanf ( "%d", &arr[i] ) ;
        printf ( "\n%d", arr[i] ) ;
    }
}
```

[C] Answer the following:

(a) An array is a collection of

1. different data types scattered throughout memory
2. the same data type scattered throughout memory
3. the same data type placed next to each other in memory
4. different data types placed next to each other in memory

(b) Are the following array declarations correct?

```
int a (25) ;
int size = 10, b[size] ;
int c = {0,1,2} ;
```

(c) Which element of the array does this expression reference?

```
num[4]
```

(d) What is the difference between the 5's in these two expressions? (Select the correct answer)

```
int num[5] ;
num[5] = 11 ;
```

1. first is particular element, second is type
2. first is array size, second is particular element
3. first is particular element, second is array size
4. both specify array size

(e) State whether the following statements are True or False:

1. The array **int num[26]** has twenty-six elements.
2. The expression **num[1]** designates the first element in the array
3. It is necessary to initialize the array at the time of declaration.
4. The expression **num[27]** designates the twenty-eighth element in the array.

[D] Attempt the following:

(a) Twenty-five numbers are entered from the keyboard into an array. The number to be searched is entered through the keyboard by the user. Write a program to find if the number to be searched is present in the array and if it is present, display the number of times it appears in the array.

(b) Twenty-five numbers are entered from the keyboard into an array. Write a program to find out how many of them are positive, how many are negative, how many are even and how many odd.

(c) Implement the Selection Sort, Bubble Sort and Insertion sort algorithms on a set of 25 numbers. (Refer Figure 8.11 for the logic of the algorithms)

- Selection sort
- Bubble Sort
- Insertion Sort

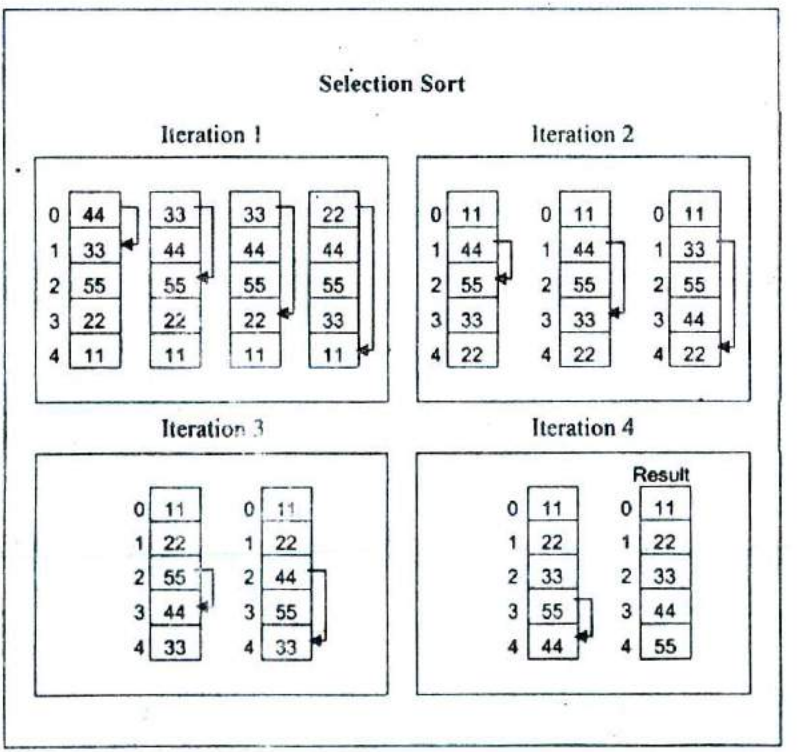


Figure 8.11 (a)

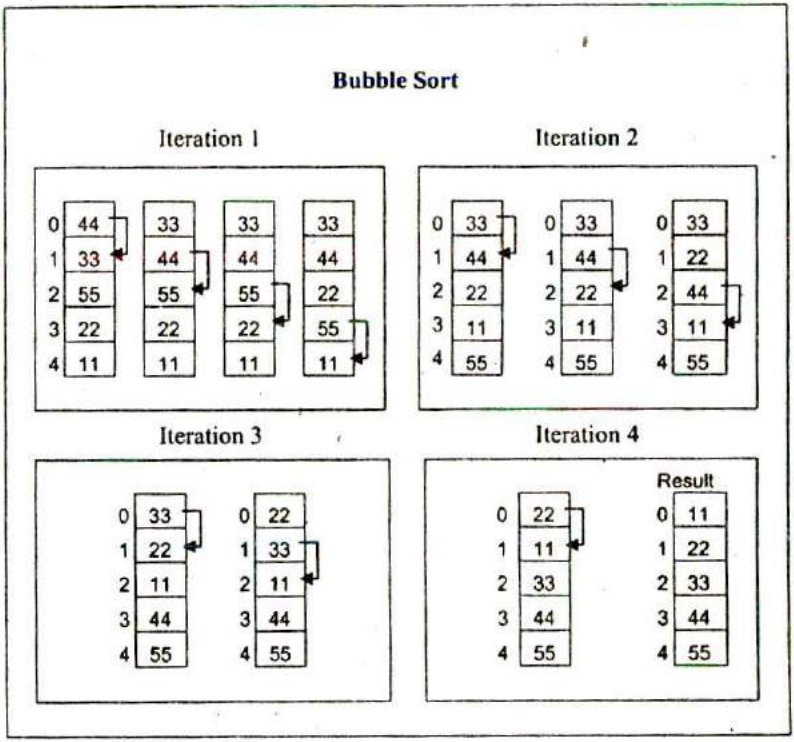


Figure 8.11 (b)

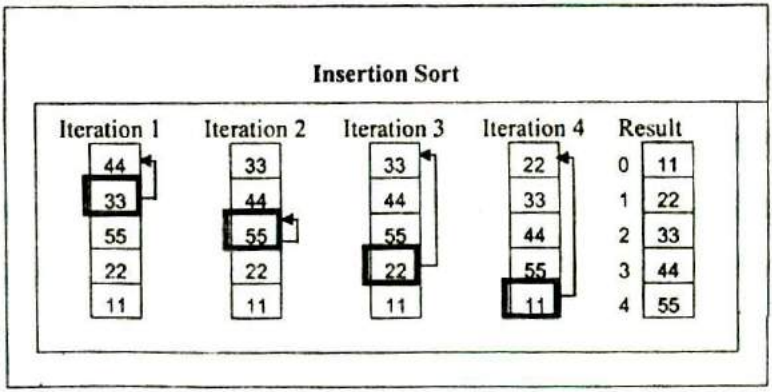


Figure 8.11 (c)

(d) Implement the following procedure to generate prime numbers from 1 to 100 into a program. This procedure is called sieve of Eratosthenes.

step 1 Fill an array **num[100]** with numbers from 1 to 100

step 2 Starting with the second entry in the array, set all its multiples to zero.

step 3 Proceed to the next non-zero element and set all its multiples to zero.

step 4 Repeat step 3 till you have set up the multiples of all the non-zero elements to zero

step 5 At the conclusion of step 4, all the non-zero entries left in the array would be prime numbers, so print out these numbers.

More on arrays, Arrays and pointers

[E] What would be the output of the following programs:

(a)
```
main( )
{
    int b[ ] = { 10, 20, 30, 40, 50 } ;
    int i ;
    for ( i = 0 ; i <= 4 ; i++ )
        printf ( "\n%d" *( b + i ) ) ;
}
```

(b)
```
main( )
{
    int b[ ] = { 0, 20, 0, 40, 5 } ;
    int i, *k ;
    k = b ;
    for ( i = 0 ; i <= 4 ; i++ )
    {
        printf ( "\n%d" *k ) ;
```

```
            k++ ;
        }
    }
```

(c)
```
main( )
{
    int a[ ] = { 2, 4, 6, 8, 10 } ;
    int i ;
    change ( a, 5 ) ;
    for ( i = 0 ; i <= 4 ; i++ )
        printf( "\n%d", a[i] ) ;
}
change ( int *b, int n )
{
    int i ;
    for ( i = 0 ; i < n ; i++ )
        *( b + i ) = *( b + i ) + 5 ;
}
```

(d)
```
main( )
{
    int a[5], i, b = 16 ;
    for ( i = 0 ; i < 5 ; i++ )
        a[i] = 2 * i ;
    f ( a, b ) ;
    for ( i = 0 ; i < 5 ; i++ )
        printf ( "\n%d", a[i] ) ;
    printf( "\n%d", b ) ;
}
f ( int *x, int y )
{
    int i ;
    for ( i = 0 ; i < 5 ; i++ )
        *( x + i ) += 2 ;
    y += 2 ;
}
```

(e)
```
main( )
{
    static int a[5] ;
    int i ;
    for ( i = 0 ; i <= 4 ; i++ )
        printf ( "\n%d", a[i] ) ;
}
```

(f)
```
main( )
{
    int a[5] = { 5, 1, 15, 20, 25 } ;
    int i, j, k = 1, m ;
    i = ++a[1] ;
    j = a[1]++ ;
    m = a[i++] ;
    printf ( "\n%d %d %d", i, j, m ) ;
}
```

[F] Point out the errors, if any, in the following programs:

(a)
```
main( )
{
    int array[6] = { 1, 2, 3, 4, 5, 6 } ;
    int i ;
    for ( i = 0 ; i <= 25 ; i++ )
        printf ( "\n%d", array[i] ) ;
}
```

(b)
```
main( )
{
    int sub[50], i ;
    for ( i = 1 ; i <= 50 ; i++ )
    {
        sub[i] = i ;
        printf ( "\n%d" , sub[i] ) ;
    }
}
```

(c)
```
main( )
{
    int a[ ] = { 10, 20, 30, 40, 50 } ;
    int j ;
    j = a ; /* store the address of zeroth element */
    j = j + 3 ;
    printf ( "\n%d" *j ) ;
}
```

(d)
```
main( )
{
    float a[ ] = { 13.24, 1.5, 1.5, 5.4, 3.5 } ;
    float *j ;
    j = a ;
    j = j + 4 ;
    printf ( "\n%d %d %d", j, *j, a[4] ) ;
}
```

(e)
```
main( )
{
    float a[ ] = { 13.24, 1.5, 1.5, 5.4, 3.5 } ;
    float *j, *k ;
    j = a ;
    k = a + 4 ;
    j = j * 2 ;
    k = k / 2 ;
    printf ( "\n%d %d", *j, *k ) ;
}
```

(f)
```
main( )
{
    int max = 5 ;
    float arr[max] ;
    for ( i = 0 ; i < max ; i++ )
        scanf ( "%f", &arr[i] ) ;
}
```

[G] Answer the following:

(a) What would happen if you try to put so many values into an array when you initialize it that the size of the array is exceeded?

1. nothing
2. possible system malfunction
3. error message from the compiler
4. other data may be overwritten

(b) In an array **int arr[12]** the word **arr** represents the a_____ of the array

(c) What would happen if you put too few elements in an array when you initialize it?

1. nothing
2. possible system malfunction
3. error message from the compiler
4. unused elements will be filled with 0's or garbage

(d) What would happen if you assign a value to an element of an array whose subscript exceeds the size of the array?

1. the element will be set to 0
2. nothing, it's done all the time
3. other data may be overwritten
4. error message from the compiler

(e) When you pass an array as an argument to a function, what actually gets passed?

1. address of the array
2. values of the elements of the array
3. address of the first element of the array
4. number of elements of the array

(f) Which of these are reasons for using pointers?

1. To manipulate parts of an array
2. To refer to keywords such as **for** and **if**
3. To return more than one value from a function
4. To refer to particular programs more conveniently

(g) If you don't initialize a static array, what would be the elements set to?

1. 0
2. an undetermined value
3. a floating point number
4. the character constant '\0'

[H] State True or False:

(a) Address of a floating-point variable is always a whole number.

(b) Which of the following is the correct way of declaring a float pointer:

5. float ptr ;
6. float *ptr ;
7. *float ptr ;
8. None of the above

(c) Add the missing statement for the following program to print 35.

```
main( )
{
    int j, *ptr ;
    *ptr = 35 ;
    printf ( "\n%d", j ) ;
}
```

(d) if **int s[5]** is a one-dimensional array of integers, which of the following refers to the third element in the array?

9. *(s + 2)
10. *(s + 3)
11. s + 3
12. s + 2

[I] Attempt the following:

(a) Write a program to copy the contents of one array into another in the reverse order.

(b) If an array **arr** contains **n** elements, then write a program to check if **arr[0] = arr[n-1], arr[1] = arr[n-2]** and so on.

(c) Find the smallest number in an array using pointers.

(d) Write a program which performs the following tasks:

- initialize an integer array of 10 elements in **main( )**
- pass the entire array to a function **modify( )**
- in **modify( )** multiply each element of array by 3
- return the control to **main( )** and print the new array elements in **main( )**

(e) The screen is divided into 25 rows and 80 columns. The characters that are displayed on the screen are stored in a special memory called VDU memory (not to be confused with ordinary memory). Each character displayed on the screen occupies two bytes in VDU memory. The first of these bytes contains the ASCII value of the character being displayed, whereas, the second byte contains the colour in which the character is displayed.

For example, the ASCII value of the character present on zeroth row and zeroth column on the screen is stored at

location number 0xB8000000. Therefore the colour of this character would be present at location number 0xB8000000 + 1. Similarly ASCII value of character in row 0, col 1 will be at location 0xB8000000 + 2, and its colour at 0xB8000000 + 3.

With this knowledge write a program which when executed would keep converting every capital letter on the screen to small case letter and every small case letter to capital letter. The procedure should stop the moment the user hits a key from the keyboard.

This is an activity of a rampant Virus called Dancing Dolls. (For monochrome adapter, use 0xB0000000 instead of 0xB8000000).

More than one dimension

[J] What would be the output of the following programs:

(a)
```
main( )
{
    int n[3][3] = {
                        2, 4, 3,
                        6, 8, 5,
                        3, 5, 1
                  };
    printf ( "\n%d %d %d", *n, n[3][3], n[2][2] ) ;
}
```

(b)
```
main( )
{
    int n[3][3] = {
                        2, 4, 3,
                        6, 8, 5,
                        3, 5, 1
                  };
    int i, *ptr ;
```

```
    ptr = n ;
    for ( i = 0 ; i <= 8 ; i++ )
        printf ( "\n%d", *( ptr + i ) ) ;
}
```

(c)
```
main( )
{
    int n[3][3] = {
                    2, 4, 3,
                    6, 8, 5,
                    3, 5, 1
                  };
    int i, j ;
    for ( i = 0 ; i <= 2 ; i++ )
        for ( j = 0 ; j <= 2 ; j++ )
            printf ( "\n%d %d", n[i][j], *( *( n + i ) + j ) ) ;
}
```

[K] Point out the errors, if any, in the following programs:

(a)
```
main( )
{
    int twod[ ][ ] = {
                        2, 4,
                        6, 8
                      };
    printf ( "\n%d", twod ) ;
}
```

(b)
```
main( )
{
    int three[3][ ] = {
                        2, 4, 3,
                        6, 8, 2,
                        2, 3 ,1
                      };
    printf ( "\n%d", three[1][1] ) ;
```

}

[L] Attempt the following:

(a) How will you initialize a three-dimensional array **threed[3][2][3]**? How will you refer the first and last element in this array?

(b) Write a program to pick up the largest number from any 5 row by 5 column matrix.

(c) Write a program to obtain transpose of a 4 x 4 matrix. The transpose of a matrix is obtained by exchanging the elements of each row with the elements of the corresponding column.

(d) Very often in fairs we come across a puzzle that contains 15 numbered square pieces mounted on a frame. These pieces can be moved horizontally or vertically. A possible arrangement of these pieces is shown below:

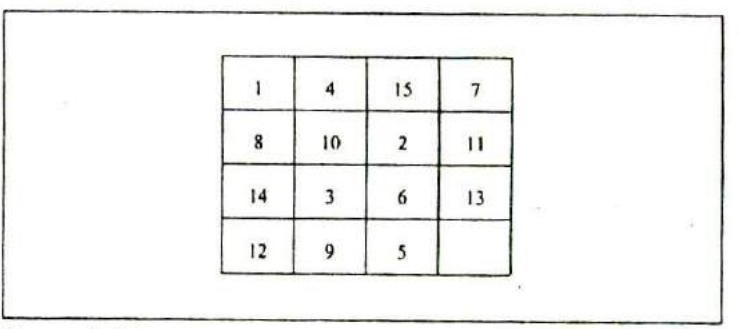


Figure 8.12

As you can see there is a blank at bottom right corner. Implement the following procedure through a program:

Draw the boxes as shown above. Display the numbers in the above order. Allow the user to hit any of the arrow keys (up, down, left, or right).

If user hits say, right arrow key then the piece with a number 5 should move to the right and blank should replace the original position of 5. Similarly, if down arrow key is hit, then 13 should move down and blank should replace the original position of 13. If left arrow key or up arrow key is hit then no action should be taken.

The user would continue hitting the arrow keys till the numbers aren't arranged in ascending order.

Keep track of the number of moves in which the user manages to arrange the numbers in ascending order. The user who manages it in minimum number of moves is the one who wins.

How do we tackle the arrow keys? We cannot receive them using **scanf( )** function. Arrow keys are special keys which are identified by their 'scan codes'. Use the following function in your program. It would return the scan code of the arrow key being hit. Don't worry about how this function is written. We are going to deal with it later. The scan codes for the arrow keys are:

up arrow key – 72 down arrow key – 80

left arrow key – 75 right arrow key – 77

```
/* Returns scan code of the key that has been hit */
#include "dos.h"
getkey( )
{
    union REGS i, o ;
```

```
    while ( !kbhit( ) )
        ;
    i.h.ah = 0 ;
    int86 ( 22, &i, &o ) ;
    return ( o.h.ah ) ;
}
```

(e) Those readers who are from an Engineering/Science background may try writing programs for following problems.

(1) Write a program to add two 6 x 6 matrices.
(2) Write a program to multiply any two 3 x 3 matrices.
(3) Write a program to sort all the elements of a 4 x 4 matrix.
(4) Write a program to obtain the determinant value of a 5 x 5 matrix.

(f) Match the following with reference to the following program segment:

```
int i, j, = 25;
int  *pi, *pj = & j;
.....
..... /* more lines of program */
.....
*pj = j + 5;
j = *pj + 5 ;
pj = pj ;
*pi = i + j
```

Each integer quantity occupies 2 bytes of memory. The value assigned to **i** begin at (hexadecimal) address F9C and the value assigned to j begins at address F9E. Match the value represented by left hand side quantities with the right.

1.	&i	a.	30
2.	&j	b.	F9E
3.	pj	c.	35
4.	*pj	d.	FA2

5.	i	e.	F9C
6.	pi	f.	67
7.	*pi	g.	unspecified
8.	(pi + 2)	h.	65
9.	(*pi + 2)	i.	F9E
10.	* (pi + 2)	j.	F9E
		k.	FAO
		l.	F9D

(g) Match the following with reference to the following segment:

```
int x[3][5] = {
                { 1, 2, 3, 4, 5 },
                { 6, 7, 8, 9, 10 },
                { 11, 12, 13, 14, 15 }
              }, *n = &x ;
```

1.	*(*(x + 2) + 1)	a.	9
2.	*(*x + 2) + 5	b.	13
3.	*(*(x + 1))	c.	4
4.	*(*(x) + 2) + 1	d.	3
5.	* (*(x + 1) + 3)	e.	2
6.	*n	f.	12
7.	*(n +2)	g.	14
8.	(*(n + 3) + 1	h.	7
9.	*(n + 5)+1	i.	1
10.	++*n	j.	8
		k.	5
		l.	10
		m.	6

(h) Match the following with reference to the following program segment:

```
struct
{
   int x, y;
} s[ ] = { 10, 20, 15, 25, 8, 75, 6, 2 };
int *i ;
i = s ;
```

1.	*(i + 3)	a.	85
2.	s[i[7]].x	b.	2
3.	s[(s + 2)->y / 3[1]].y	c.	6
4.	i[i[1]-i[2]]	d.	7
5.	i[s[3].y]	e.	16
6.	(s + 1)->x + 5	f.	15
7.	*(1 +i)**(i + 4) / *i	g.	25
8.	s[i[0] – i[4]].y + 10	h.	8
9.	(*(s + *(i + 1) / *i)).x + 2	i.	1
10.	++i[i[6]]	j.	100
		k.	10
		l.	20

(i) Match the following with reference to the following program segment:

```
unsigned int arr[3][3] = {
                            2, 4, 6,
                            9, 1, 10,
                            16, 64, 5
                          };
```

1.	**arr	a.	64
2.	**arr < *(*arr + 2)	b.	18
3.	*(arr + 2) / (*(*arr + 1) > **arr)	c.	6
4.	*(arr[1] + 1) | arr[1][2]	d.	3
5.	*(arr[0]) | *(arr[2])	e.	0
6.	arr[1][1] < arr[0][1]	f.	16
7.	arr[2][[1] & arr[2][0]	g.	1
8.	arr[2][2] | arr[0][1]	h.	11
9.	arr[0][1] ^ arr[0][2]	i.	20
10.	++**arr + --arr[1][1]	j.	2
		k.	5
		l.	4

(j) Write a program that interchanges the odd and even components of an array.

(k) Write a program to find if a square matrix is symmetric.

(l) Write a function to find the norm of a matrix. The norm is defined as the square root of the sum of squares of all elements in the matrix.

(m) Given an array **p[5]**, write a function to shift it circularly left by two positions. Thus, if p[0] = 15, p[1]= 30, p[2] = 28, p[3]= 19 and p[4] = 61 then after the shift p[0] = 28, p[1] = 19, p[2] = 61, p[3] = 15 and p[4] = 30. Call this function for a (4 x 5) matrix and get its rows left shifted.

(n) A 6 x 6 matrix is entered through the keyboard and stored in a 2-dimensional array **mat[7][7]**. Write a program to obtain the Determinant values of this matrix.

(o) For the following set of sample data, compute the standard deviation and the mean.

-6, -12, 8, 13, 11, 6, 7, 2, -6, -9, -10, 11, 10, 9, 2

The formula for standard deviation is

$$\frac{\sqrt{(x_i - \bar{x})^2}}{n}$$

where x_i is the data item and $\bar{x}$ is the mean.

(p) The area of a triangle can be computed by the sine law when 2 sides of the triangle and the angle between them are known.

Area = (1 / 2) ab sin (angle)

Given the following 6 triangular pieces of land, write a program to find their area and determine which is largest,

Plot No.	a	b	angle
1	137.4	80.9	0.78
2	155.2	92.62	0.89
3	149.3	97.93	1.35

4	160.0	100.25	9.00
5	155.6	68.95	1.25
6	149.7	120.0	1.75

(q) For the following set of n data points (x, y), compute the correlation coefficient r, given by

$$r = \frac{\sum xy - \sum x \sum y}{\sqrt{[n\sum x^2 - (\sum x)^2][n\sum y^2 - (\sum y)^2]}}$$

x	y
34.22	102.43
39.87	100.93
41.85	97.43
43.23	97.81
40.06	98.32
53.29	98.32
53.29	100.07
54.14	97.08
49.12	91.59
40.71	94.85
55.15	94.65

(r) For the following set of point given by (x, y) fit a straight line given by

$y = a + bx$

where,

$a = \bar{y} - b\bar{x}$ and

$$b = \frac{n\sum yx - \sum x \sum y}{[n\sum x^2 - (\sum x)^2]}$$

x	y
3.0	1.5

4.5	2.0
5.5	3.5
6.5	5.0
7.5	6.0
8.5	7.5
8.0	9.0
9.0	10.5
9.5	12.0
10.0	14.0

(s) The **X** and **Y** coordinates of 10 different points are entered through the keyboard. Write a program to find the distance of last point from the first point (sum of distance between consecutive points).

9 Puppetting On Strings

- What are Strings
- More about Strings
- Pointers and Strings
- Standard Library String Functions
  - *strlen( )*
  - *strcpy( )*
  - *strcat( )*
  - *strcmp( )*
- Two-Dimensional Array of Characters
- Array of Pointers to Strings
- Limitation of Array of Pointers to Strings Solution
- Summary
- Exercise

In the last chapter you learnt how to define arrays of differing sizes and dimensions, how to initialize arrays, how to pass arrays to a function, etc. With this knowledge under your belt, you should be ready to handle strings, which are, simply put, a special kind of array. And strings, the ways to manipulate them, and how pointers are related to strings are going to be the topics of discussion in this chapter.

What are Strings

The way a group of integers can be stored in an integer array, similarly a group of characters can be stored in a character array. Character arrays are many a time also called strings. Many languages internally treat strings as character arrays, but somehow conceal this fact from the programmer. Character arrays or strings are used by programming languages to manipulate text such as words and sentences.

A string constant is a one-dimensional array of characters terminated by a null ('\0'). For example,

```
char name[ ] = { 'H', 'A', 'E', 'S', 'L', 'E', 'R', '\0' } ;
```

Each character in the array occupies one byte of memory and the last character is always '\0'. What character is this? It looks like two characters, but it is actually only one character, with the \ indicating that what follows it is something special. '\0' is called null character. Note that '\0' and '0' are not same. ASCII value of '\0' is 0, whereas ASCII value of '0' is 48. Figure 9.1 shows the way a character array is stored in memory. Note that the elements of the character array are stored in contiguous memory locations.

The terminating null ('\0') is important, because it is the only way the functions that work with a string can know where the string

ends. In fact, a string not terminated by a '\0' is not really a string, but merely a collection of characters.

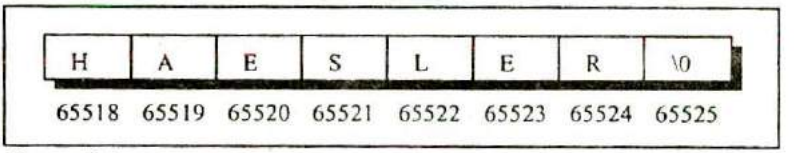


Figure 9.1

C concedes the fact that you would use strings very often and hence provides a shortcut for initializing strings. For example, the string used above can also be initialized as,

```
char name[ ] = "HAESLER" ;
```

Note that, in this declaration '\0' is not necessary. C inserts the null character automatically.

More about Strings

In what way are character arrays different than numeric arrays? Can elements in a character array be accessed in the same way as the elements of a numeric array? Do I need to take any special care of '\0'? Why numeric arrays don't end with a '\0'? Declaring strings is okay, but how do I manipulate them? Questions galore!! Well, let us settle some of these issues right away with the help of sample programs.

```
/* Program to demonstrate printing of a string */
main( )
{
    char name[ ] = "Klinsman" ;
    int i = 0 ;

    while ( i <= 7 )
    {
```

```
        printf ( "%c", name[i] ) ;
        i++ ;
    }
}
```

And here is the output...

```
Klinsman
```

No big deal. We have initialized a character array, and then printed out the elements of this array within a **while** loop. Can we write the **while** loop without using the final value 7? We can; because we know that each character array always ends with a '\0'. Following program illustrates this.

```
main( )
{
    char name[ ] = "Klinsman" ;
    int i = 0 ;

    while ( name[i] != '\0' )
    {
        printf ( "%c", name[i] ) ;
        i++ ;
    }
}
```

And here is the output...

```
Klinsman
```

This program doesn't rely on the length of the string (number of characters in it) to print out its contents and hence is definitely more general than the earlier one. Here is another version of the same program; this one uses a pointer to access the array elements.

```
main( )
{
    char name[ ] = "Klinsman" ;
    char *ptr ;

    ptr = name ; /* store base address of string */

    while ( *ptr != '\0' )
    {
        printf ( "%c", *ptr ) ;
        ptr++ ;
    }
}
```

As with the integer array, by mentioning the name of the array we get the base address (address of the zeroth element) of the array. This base address is stored in the variable **ptr** using,

```
ptr = name ;
```

Once the base address is obtained in **ptr**, ***ptr** would yield the value at this address, which gets printed promptly through,

```
printf ( "%c", *ptr ) ;
```

Then, **ptr** is incremented to point to the next character in the string. This derives from two facts: array elements are stored in contiguous memory locations and on incrementing a pointer it points to the immediately next location of its type. This process is carried out till **ptr** doesn't point to the last character in the string, that is, '\0'.

In fact, the character array elements can be accessed exactly in the same way as the elements of an integer array. Thus, all the following notations refer to the same element:

```
name[i]
*( name + i )
*( i + name )
i[name]
```

Even though there are so many ways (as shown above) to refer to the elements of a character array, rarely is any one of them used. This is because **printf( )** function has got a sweet and simple way of doing it, as shown below. Note that **printf( )** doesn't print the '\0'.

```
main( )
{
    char name[ ] = "Klinsman" ;
    printf ( "%s", name ) ;
}
```

The **%s** used in **printf( )** is a format specification for printing out a string. The same specification can be used to receive a string from the keyboard, as shown below.

```
main( )
{
    char name[25] ;

    printf ( "Enter your name " ) ;
    scanf ( "%s", name ) ;
    printf ( "Hello %s!", name ) ;
}
```

And here is a sample run of the program...

```
Enter your name Debashish
Hello Debashish!
```

Note that the declaration **char name[25]** sets aside 25 bytes under the array **name[ ]**, whereas the **scanf( )** function fills in the characters typed at keyboard into this array until the enter key is hit. Once enter is hit, **scanf( )** places a '\0' in the array. Naturally, we should pass the base address of the array to the **scanf( )** function.

While entering the string using **scanf( )** we must be cautious about two things:

(a) The length of the string should not exceed the dimension of the character array. This is because the C compiler doesn't perform bounds checking on character arrays. Hence, if you carelessly exceed the bounds there is always a danger of overwriting something important, and in that event, you would have nobody to blame but yourselves.

(b) **scanf( )** is not capable of receiving multi-word strings. Therefore names such as 'Debashish Roy' would be unacceptable. The way to get around this limitation is by using the function **gets( )**. The usage of functions **gets( )** and its counterpart **puts( )** is shown below.

```
main( )
{
    char name[25] ;

    printf ( "Enter your full name " ) ;
    gets ( name ) ;
    puts ( "Hello!" ) ;
    puts ( name ) ;
}
```

And here is the output...

```
Enter your name Debashish Roy
Hello!
```

```
Debashish Roy
```

The program and the output are self-explanatory except for the fact that, **puts( )** can display only one string at a time (hence the use of two **puts( )** in the program above). Also, on displaying a string, unlike **printf( )**, **puts( )** places the cursor on the next line. Though **gets( )** is capable of receiving only one string at a time, the plus point with **gets( )** is that it can receive a multi-word string.

If we are prepared to take the trouble we can make **scanf( )** accept multi-word strings by writing it in this manner:

```
char name[25] ;
printf ( "Enter your full name " ) ;
scanf ( "%[^\n]s", name ) ;
```

Though workable this is the best of the ways to call a function, you would agree.

Pointers and Strings

Suppose we wish to store "Hello". We may either store it in a string or we may ask the C compiler to store it at some location in memory and assign the address of the string in a **char** pointer. This is shown below:

```
char str[ ] = "Hello" ;
char *p = "Hello" ;
```

There is a subtle difference in usage of these two forms. For example, we cannot assign a string to another, whereas, we can assign a **char** pointer to another **char** pointer. This is shown in the following program.

```
main( )
{
    char str1[ ] = "Hello" ;
    char str2[10] ;

    char *s = "Good Morning" ;
    char *q ;

    str2 = str1 ; /* error */
    q = s ; /* works */
}
```

Also, once a string has been defined it cannot be initialized to another set of characters. Unlike strings, such an operation is perfectly valid with **char** pointers.

```
main( )
{
    char str1[ ] = "Hello" ;
    char *p = "Hello" ;
    str1 = "Bye" ; /* error */
    p = "Bye" ; /* works */
}
```

Standard Library String Functions

With every C compiler a large set of useful string handling library functions are provided. Figure 9.2 lists the more commonly used functions along with their purpose.

Function	Use
strlen	Finds length of a string
strlwr	Converts a string to lowercase
strupr	Converts a string to uppercase
strcat	Appends one string at the end of another
strncat	Appends first n characters of a string at the end of another
strcpy	Copies a string into another
strncpy	Copies first n characters of one string into another
strcmp	Compares two strings
strncmp	Compares first n characters of two strings
strcmpi	Compares two strings without regard to case ("i" denotes that this function ignores case)
stricmp	Compares two strings without regard to case (identical to strcmpi)
strnicmp	Compares first n characters of two strings without regard to case
strdup	Duplicates a string
strchr	Finds first occurrence of a given character in a string
strrchr	Finds last occurrence of a given character in a string
strstr	Finds first occurrence of a given string in another string
strset	Sets all characters of string to a given character
strnset	Sets first n characters of a string to a given character
strrev	Reverses string

Figure 9.2

Out of the above list we shall discuss the functions **strlen( )**, **strcpy( )**, **strcat( )** and **strcmp( )**, since these are the most commonly used functions. This will also illustrate how the library functions in general handle strings. Let us study these functions one by one.

strlen( )

This function counts the number of characters present in a string. Its usage is illustrated in the following program.

```
main( )
{
    char arr[ ] = "Bamboozled" ;
    int len1, len2 ;

    len1 = strlen ( arr ) ;
    len2 = strlen ( "Humpty Dumpty" ) ;

    printf ( "\nstring = %s length = %d", arr, len1 ) ;
    printf ( "\nstring = %s length = %d", "Humpty Dumpty", len2 ) ;
}
```

The output would be...

```
string = Bamboozled length = 10
string = Humpty Dumpty length = 13
```

Note that in the first call to the function **strlen( )**, we are passing the base address of the string, and the function in turn returns the length of the string. While calculating the length it doesn't count '\0'. Even in the second call,

```
len2 = strlen ( "Humpty Dumpty" ) ;
```

what gets passed to **strlen( )** is the address of the string and not the string itself. Can we not write a function **xstrlen( )** which imitates the standard library function **strlen( )**? Let us give it a try...

```
/* A look-alike of the function strlen( ) */
main( )
{
```

```
    char arr[ ] = "Bamboozled" ;
    int len1, len2 ;

    len1 = xstrlen ( arr ) ;
    len2 = xstrlen ( "Humpty Dumpty" ) ;

    printf ( "\nstring = %s length = %d", arr, len1 ) ;
    printf ( "\nstring = %s length = %d", "Humpty Dumpty", len2 ) ;
}

xstrlen ( char *s )
{
    int length = 0 ;

    while ( *s != '\0' )
    {
        length++ ;
        s++ ;
    }

    return ( length ) ;
}
```

The output would be...

```
string = Bamboozled length = 10
string = Humpty Dumpty length = 13
```

The function **xstrlen( )** is fairly simple. All that it does is keep counting the characters till the end of string is not met. Or in other words keep counting characters till the pointer s doesn't point to '\0'.

strcpy( )

This function copies the contents of one string into another. The base addresses of the source and target strings should be supplied to this function. Here is an example of **strcpy( )** in action...

```
main( )
{
    char source[ ] = "Sayonara" ;
    char target[20] ;

    strcpy ( target, source ) ;
    printf ( "\nsource string = %s", source ) ;
    printf ( "\ntarget string = %s", target ) ;
}
```

And here is the output...

```
source string = Sayonara
target string = Sayonara
```

On supplying the base addresses, **strcpy( )** goes on copying the characters in source string into the target string till it doesn't encounter the end of source string ('\0'). It is our responsibility to see to it that the target string's dimension is big enough to hold the string being copied into it. Thus, a string gets copied into another, piece-meal, character by character. There is no short cut for this. Let us now attempt to mimic **strcpy( )**, via our own string copy function, which we will call **xstrcpy( ).**

```
main( )
{
    char source[ ] = "Sayonara" ;
    char target[20] ;

    xstrcpy ( target, source ) ;
```

```
    printf ( "\nsource string = %s", source ) ;
    printf ( "\ntarget string = %s", target ) ;
}

xstrcpy ( char *t, char *s )
{
    while ( *s != '\0' )
    {
        *t = *s ;
        s++ ;
        t++ ;
    }
    *t = '\0' ;
}
```

The output of the program would be...

```
source string = Sayonara
target string = Sayonara
```

Note that having copied the entire source string into the target string, it is necessary to place a '\0' into the target string, to mark its end.

If you look at the prototype of **strcpy( )** standard library function, it looks like this...

```
strcpy ( char *t, const char *s ) ;
```

We didn't use the keyword **const** in our version of **xstrcpy( )** and still our function worked correctly. So what is the need of the **const** qualifier?

What would happen if we add the following lines beyond the last statement of **xstrcpy( )**?.

```
s = s - 8 ;
*s = 'K' ;
```

This would change the source string to "Kayonara". Can we not ensure that the source string doesn't change even accidentally in **xstrcpy( )**? We can, by changing the definition as follows:

```
void xstrcpy ( char *t, const char *s )
{
    while ( *s != '\0' )
    {
        *t = *s ;
        s++ ;
        t++ ;
    }
    *t = '\0' ;
}
```

By declaring **char *s** as **const** we are declaring that the source string should remain constant (should not change). Thus the **const** qualifier ensures that your program does not inadvertently alter a variable that you intended to be a constant. It also reminds anybody reading the program listing that the variable is not intended to change.

We can use **const** in several situations. The following code fragment would help you to fix your ideas about **const** further.

```
char *p = "Hello" ; /* pointer is variable, so is string */
*p = 'M' ; /* works */
p = "Bye" ; /* works */

const char *q = "Hello" ; /* string is fixed pointer is not */
*q = 'M' ; /* error */
q = "Bye" ; /* works */
```

```
char const *s = "Hello" ; /* string is fixed pointer is not */
*s = 'M' ; /* error */
s = "Bye" ; /* works */

char * const t = "Hello" ; /* pointer is fixed string is not */
*t = 'M' ; /* works */
t = "Bye" ; /* error */

const char * const u = "Hello" ; /* string is fixed so is pointer */
*u = 'M' ; /* error */
u = "Bye" ; /* error */
```

The keyword **const** can be used in context of ordinary variables like **int**, **float**, etc. The following program shows how this can be done.

```
main( )
{
    float r, a ;
    const float pi = 3.14 ;

    printf ( "\nEnter radius of circle " ) ;
    scanf ( "%f", &r ) ;
    a = pi * r * r ;
    printf ( "\nArea of circle = %f", a ) ;
}
```

strcat( )

This function concatenates the source string at the end of the target string. For example, "Bombay" and "Nagpur" on concatenation would result into a string "BombayNagpur". Here is an example of **strcat( )** at work.

```
main( )
{
```

```
    char source[ ] = "Folks!" ;
    char target[30] = "Hello" ;

    strcat ( target, source ) ;
    printf ( "\nsource string = %s", source ) ;
    printf ( "\ntarget string = %s", target ) ;
}
```

And here is the output...

```
source string = Folks!
target string = HelloFolks!
```

Note that the target string has been made big enough to hold the final string. I leave it to you to develop your own **xstrcat( )** on lines of **xstrlen( )** and **xstrcpy( ).**

strcmp( )

This is a function which compares two strings to find out whether they are same or different. The two strings are compared character by character until there is a mismatch or end of one of the strings is reached, whichever occurs first. If the two strings are identical, **strcmp( )** returns a value zero. If they're not, it returns the numeric difference between the ASCII values of the first non-matching pairs of characters. Here is a program which puts **strcmp( )** in action.

```
main( )
{
    char string1[ ] = "Jerry" ;
    char string2[ ] = "Ferry" ;
    int i, j, k ;

    i = strcmp ( string1, "Jerry" ) ;
    j = strcmp ( string1, string2 ) ;
```

```
    k = strcmp ( string1, "Jerry boy" ) ;

    printf ( "\n%d %d %d", i, j, k ) ;
}
```

And here is the output...

```
0 4 -32
```

In the first call to **strcmp( )**, the two strings are identical—"Jerry" and "Jerry"—and the value returned by **strcmp( )** is zero. In the second call, the first character of "Jerry" doesn't match with the first character of "Ferry" and the result is 4, which is the numeric difference between ASCII value of 'J' and ASCII value of 'F'. In the third call to **strcmp( )** "Jerry" doesn't match with "Jerry boy", because the null character at the end of "Jerry" doesn't match the blank in "Jerry boy". The value returned is -32, which is the value of null character minus the ASCII value of space, i.e., '\0' minus ' ', which is equal to -32.

The exact value of mismatch will rarely concern us. All we usually want to know is whether or not the first string is alphabetically before the second string. If it is, a negative value is returned; if it isn't, a positive value is returned. Any non-zero value means there is a mismatch. Try to implement this procedure into a function **xstrcmp( )**.

Two-Dimensional Array of Characters

In the last chapter we saw several examples of 2-dimensional integer arrays. Let's now look at a similar entity, but one dealing with characters. Our example program asks you to type your name. When you do so, it checks your name against a master list to see if you are worthy of entry to the palace. Here's the program...

```
#define FOUND 1
#define NOTFOUND 0
main( )
{
    char masterlist[6][10] = {
                                "akshay",
                                "parag",
                                "raman",
                                "srinivas",
                                "gopal",
                                "rajesh"
                              };
    int i, flag, a ;
    char yourname[10] ;
    printf ( "\nEnter your name " ) ;
    scanf ( "%s", yourname ) ;

    flag = NOTFOUND ;
    for ( i = 0 ; i <= 5 ; i++ )
    {
        a = strcmp ( &masterlist[i][0], yourname ) ;
        if ( a == 0 )
        {
            printf ( "Welcome, you can enter the palace" ) ;
            flag = FOUND ;
            break ;
        }
    }

    if ( flag == NOTFOUND )
        printf ( "Sorry, you are a trespasser" ) ;
}
```

And here is the output for two sample runs of this program...

```
Enter your name dinesh
Sorry, you are a trespasser
```

```
Enter your name raman
Welcome, you can enter the palace
```

Notice how the two-dimensional character array has been initialized. The order of the subscripts in the array declaration is important. The first subscript gives the number of names in the array, while the second subscript gives the length of each item in the array.

Instead of initializing names, had these names been supplied from the keyboard, the program segment would have looked like this...

```
for ( i = 0 ; i <= 5 ; i++ )
    scanf ( "%s", &masterlist[i][0] ) ;
```

While comparing the strings through **strcmp( )**, note that the addresses of the strings are being passed to **strcmp( )**. As seen in the last section, if the two strings match, **strcmp( )** would return a value 0, otherwise it would return a non-zero value.

The variable **flag** is used to keep a record of whether the control did reach inside the **if** or not. To begin with, we set **flag** to NOTFOUND. Later through the loop if the names match, **flag** is set to FOUND. When the control reaches beyond the **for** loop, if **flag** is still set to NOTFOUND, it means none of the names in the **masterlist[ ][ ]** matched with the one supplied from the keyboard.

The names would be stored in the memory as shown in Figure 9.3. Note that each string ends with a '\0'. The arrangement as you can appreciate is similar to that of a two-dimensional numeric array.

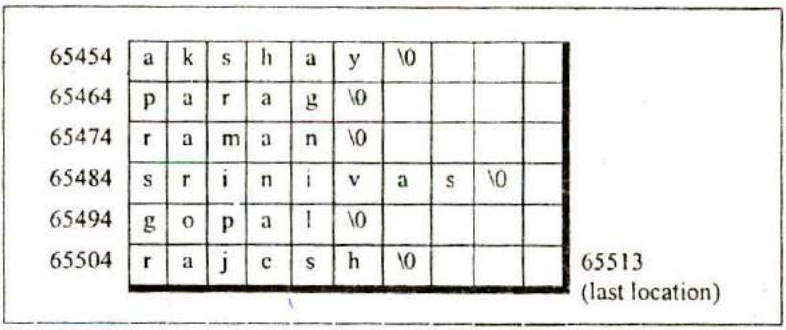


Figure 9.3

Here, 65454, 65464, 65474, etc. are the base addresses of successive names. As seen from the above pattern some of the names do not occupy all the bytes reserved for them. For example, even though 10 bytes are reserved for storing the name "akshay", it occupies only 7 bytes. Thus, 3 bytes go waste. Similarly, for each name there is some amount of wastage. In fact, more the number of names, more would be the wastage. Can this not be avoided? Yes, it can be... by using what is called an 'array of pointers', which is our next topic of discussion.

Array of Pointers to Strings

As we know, a pointer variable always contains an address. Therefore, if we construct an array of pointers it would contain a number of addresses. Let us see how the names in the earlier example can be stored in the array of pointers.

```
char *names[ ] = {
                    "akshay",
                    "parag",
                    "raman",
```

```
        "srinivas",
        "gopal",
        "rajesh"
};
```

In this declaration **names[ ]** is an array of pointers. It contains base addresses of respective names. That is, base address of "akshay" is stored in **names[0]**, base address of "parag" is stored in **names[1]** and so on. This is depicted in Figure 9.4.

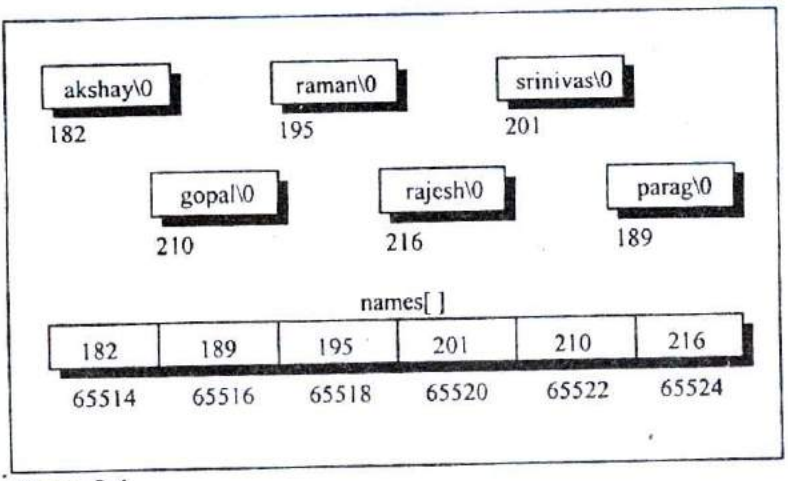


Figure 9.4

In the two-dimensional array of characters, the strings occupied 60 bytes. As against this, in array of pointers, the strings occupy only 41 bytes—a net saving of 19 bytes. A substantial saving, you would agree. But realize that actually 19 bytes are not saved, since 12 bytes are sacrificed for storing the addresses in the array **names[ ]**. Thus, one reason to store strings in an array of pointers is to make a more efficient use of available memory.

Another reason to use an array of pointers to store strings is to obtain greater ease in manipulation of the strings. This is shown by

the following programs. The first one uses a two-dimensional array of characters to store the names, whereas the second uses an array of pointers to strings. The purpose of both the programs is very simple. We want to exchange the position of the names "raman" and "srinivas".

```
/* Exchange names using 2-D array of characters */
main( )
{
    char names[ ][10] = {
                            "akshay",
                            "parag",
                            "raman",
                            "srinivas",
                            "gopal",
                            "rajesh"
                        };
    int i ;
    char t ;

    printf ( "\nOriginal: %s %s", &names[2][0], &names[3][0] ) ;

    for ( i = 0 ; i <= 9 ; i++ )
    {
        t = names[2][i] ;
        names[2][i] = names[3][i] ;
        names[3][i] = t ;
    }

    printf ( "\nNew: %s %s", &names[2][0], &names[3][0] ) ;
}
```

And here is the output...

```
Original: raman srinivas
New: srinivas raman
```

Note that in this program to exchange the names we are required to exchange corresponding characters of the two names. In effect, 10 exchanges are needed to interchange two names.

Let us see, if the number of exchanges can be reduced by using an array of pointers to strings. Here is the program...

```
main( )
{
    char *names[ ] = {
                        "akshay",
                        "parag",
                        "raman",
                        "srinivas",
                        "gopal",
                        "rajesh"
                    };
    char *temp ;

    printf ( "Original: %s %s", names[2], names[3] ) ;

    temp = names[2] ;
    names[2] = names[3] ;
    names[3] = temp ;

    printf ( "\nNew: %s %s", names[2], names[3] ) ;
}
```

And here is the output...

```
Original: raman srinivas
New: srinivas raman
```

The output is same as the earlier program. In this program all that we are required to do is exchange the addresses (of the names) stored in the array of pointers, rather than the names themselves.

Thus, by effecting just one exchange we are able to interchange names. This makes handling strings very convenient.

Thus, from the point of view of efficient memory usage and ease of programming, an array of pointers to strings definitely scores over a two-dimensional character array. That is why, even though in principle strings can be stored and handled through a two-dimensional array of characters, in actual practice it is the array of pointers to strings, which is more commonly used.

Limitation of Array of Pointers to Strings

When we are using a two-dimensional array of characters we are at liberty to either initialize the strings where we are declaring the array, or receive the strings using **scanf( )** function. However, when we are using an array of pointers to strings we can initialize the strings at the place where we are declaring the array, but we cannot receive the strings from keyboard using **scanf( )**. Thus, the following program would never work out.

```
main( )
{
    char *names[6] ;
    int i ;

    for ( i = 0 ; i <= 5 ; i++ )
    {
        printf ( "\nEnter name " ) ;
        scanf ( "%s", names[i] ) ;
    }
}
```

The program doesn't work because; when we are declaring the array it is containing garbage values. And it would be definitely

wrong to send these garbage values to **scanf( )** as the addresses where it should keep the strings received from the keyboard.

Solution

If we are bent upon receiving the strings from keyboard using **scanf( )** and then storing their addresses in an array of pointers to strings we can do it in a slightly round about manner as shown below.

```
#include "alloc.h"
main( ) .
{
    char *names[6] ;
    char n[50] ;
    int len, i ;
    char *p ;

    for ( i = 0 ; i <= 5 ; i++ )
    {
        printf ( "\nEnter name " ) ;
        scanf ( "%s", n ) ;
        len = strlen ( n ) ;
        p = malloc ( len + 1 ) ;
        strcpy ( p, n ) ;
        names[i] = p ;
    }

    for ( i = 0 ; i <= 5 ; i++ )
        printf ( "\n%s", names[i] ) ;
}
```

Here we have first received a name using **scanf( )** in a string **n[ ]**. Then we have found out its length using **strlen( )** and allocated space for making a copy of this name. This memory allocation has been done using a standard library function called **malloc( )**. This

function requires the number of bytes to be allocated and returns the base address of the chunk of memory that it allocates. The address returned by this function is always of the type **void ***. Hence it has been converted into **char *** using a feature called typecasting. Typecasting is discussed in detail in Chapter 15. The prototype of this function has been declared in the file 'alloc.h'. Hence we have **#included** this file.

But why did we not use array to allocate memory? This is because with arrays we have to commit to the size of the array at the time of writing the program. Moreover, there is no way to increase or decrease the array size during execution of the program. In other words, when we use arrays static memory allocation takes place. Unlike this, using **malloc( )** we can allocate memory dynamically, during execution. The argument that we pass to **malloc( )** can be a variable whose value can change during execution.

Once we have allocated the memory using **malloc( )** we have copied the name received through the keyboard into this allocated space and finally stored the address of the allocated chunk in the appropriate element of **names[ ]**, the array of pointers to strings.

This solution suffers in performance because we need to allocate memory and then do the copying of string for each name received through the keyboard.

Summary

(a) A string is nothing but an array of characters terminated by '\0'.
(b) Being an array, all the characters of a string are stored in contiguous memory locations.
(c) Though **scanf( )** can be used to receive multi-word strings, **gets( )** can do the same job in a cleaner way.
(d) Both **printf( )** and **puts( )** can handle multi-word strings.

(e) Strings can be operated upon using several standard library functions like **strlen( )**, **strcpy( )**, **strcat( )** and **strcmp( )** which can manipulate strings. More importantly we imitated some of these functions to learn how these standard library functions are written.

(f) Though in principle a 2-D array can be used to handle several strings, in practice an array of pointers to strings is preferred since it takes less space and is efficient in processing strings.

(g) **malloc( )** function can be used to allocate space in memory on the fly during execution of the program.

Exercise

Simple strings

[A] What would be the output of the following programs:

(a)
```
main( )
{
    char c[2] = "A" ;
    printf ( "\n%c", c[0] ) ;
    printf ( "\n%s", c ) ;
}
```

(b)
```
main( )
{
    char s[ ] = "Get organised! learn C!!" ;
    printf ( "\n%s", &s[2] ) ;
    printf ( "\n%s", s ) ;
    printf ( "\n%s", &s ) ;
    printf ( "\n%c", s[2] ) ;
}
```

(c)
```
main( )
{
    char s[ ] = "No two viruses work similarly" ;
```

```
        int i = 0 ;
        while ( s[i] != 0 )
        {
            printf ( "\n%c %c", s[i], *( s + i ) ) ;
            printf ( "\n%c %c", i[s], *( i + s ) ) ;
            i++ ;
        }
    }
```

(d)
```
    main( )
    {
        char s[ ] = "Churchgate: no church no gate" ;
        char t[25] ;
        char *ss, *tt ;
        ss = s ;
        while ( *ss != '\0' )
            *ss++ = *tt++ ;
        printf ( "\n%s", t ) ;
    }
```

(e)
```
    main( )
    {
        char str1[ ] = { 'H', 'e', 'l', 'l', 'o' } ;
        char str2[ ] = "Hello" ;

        printf ( "\n%s", str1 ) ;
        printf ( "\n%s", str2 ) ;
    }
```

(f)
```
    main( )
    {
        printf ( 5 + "Good Morning " ) ;
    }
```

(g)
```
    main( )
    {
        printf ( "%c", "abcdefgh"[4] ) ;
    }
```

(h)
```
main( )
{
    printf ( "\n%d%d", sizeof ( '3' ), sizeof ( "3" ), sizeof ( 3 ) ) ;
}
```

[B] Point out the errors, if any, in the following programs:

(a)
```
main( )
{
    char *str1 = "United" ;
    char *str2 = "Front" ;
    char *str3 ;
    str3 = strcat ( str1, str2 ) ;
    printf ( "\n%s", str3 ) ;
}
```

(b)
```
main( )
{
    int arr[ ] = { 'A', 'B', 'C', 'D' } ;
    int i ;
    for ( i = 0 ; i <= 3 ; i++ )
        printf ( "\n%d", arr[i] ) ;
}
```

(c)
```
main( )
{
    char arr[8] = "Rhombus" ;
    int i ;
    for ( i = 0 ; i <= 7 ; i++ )
        printf ( "\n%d", *arr ) ;
        arr++ ;
}
```

[C] Fill in the blanks:

(a) "A" is a _____ while 'A' is a _____.

(b) A string is terminated by a _____ character, which is written as _____.

(c) The array **char name[10]** can consist of a maximum of _____ characters.

(d) The array elements are always stored in _____ memory locations.

[D] Attempt the following:

(a) Which is more appropriate for reading in a multi-word string?

gets( ) printf( ) scanf( ) puts( )

(b) If the string "Alice in wonder land" is fed to the following **scanf( )** statement, what will be the contents of the arrays **str1**, **str2**, **str3** and **str4**?

```
scanf ( "%s%s%s%s%s", str1, str2, str3, str4 ) ;
```

(c) Write a program that converts all lowercase characters in a given string to its equivalent uppercase character.

(d) Write a program that extracts part of the given string from the specified position. For example, if the sting is "Working with strings is fun", then if from position 4, 4 characters are to be extracted then the program should return string as "king". Moreover, if the position from where the string is to be extracted is given and the number of characters to be extracted is 0 then the program should extract entire string from the specified position.

(e) Write a program that converts a string like "124" to an integer 124.

(f) Write a program that replaces two or more consecutive blanks in a string by a single blank. For example, if the input is

Grim return to the planet of apes!!

the output should be

Grim return to the planet of apes!!

Two-dimensional array, Array of pointers to strings

[E] Answer the following:

(a) How many bytes in memory would be occupied by the following array of pointers to strings? How many bytes would be required to store the same strings, if they are stored in a two-dimensional character array?

```
char *mess[ ] = {
                    "Hammer and tongs",
                    "Tooth and nail",
                    "Spit and polish",
                    "You and C"
                };
```

(b) Can an array of pointers to strings be used to collect strings from the keyboard? If not, why not?

[F] Attempt the following:

(a) Write a program that uses an array of pointers to strings **str**[ ]. Receive two strings **str1** and **str2** and check if **str1** is embedded in any of the strings in **str[ ]**. If str1 is found, then replace it with **str2**.

```
char *str[ ] ={
                "We will teach you how to...",
                "Move a mountain",
                "Level a building",
                "Erase the past",
```

```
                "Make a million",
                "...all through C!"
        } ;
```

For example if **str1** contains "mountain" and **str2** contains "car", then the second string in **str** should get changed to "Move a car".

(b) Write a program to sort a set of names stored in an array in alphabetical order.

(c) Write a program to reverse the strings stored in the following array of pointers to strings:

```
char *s[ ] = {
                "To err is human...",
                "But to really mess things up...",
                "One needs to know C!!"
        };
```

Hint: Write a function **xstrrev (string)** which should reverse the contents of one string. Call this function for reversing each string stored in **s**.

(d) Develop a program that receives the month and year from the keyboard as integers and prints the calendar in the following format.

September 2004

Mon	Tue	Wed	Thu	Fri	Sat	Sun
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30			

Note that according to the Gregorian calendar 01/01/1900 was Monday. With this as the base the calendar should be generated.

(e) Modify the above program suitably so that once the calendar for a particular month and year has been displayed on the screen, then using arrow keys the user must be able to change the calendar in the following manner:

Up arrow key : Next year, same month
Down arrow key : Previous year, same month
Right arrow key : Same year, next month
Left arrow key : Same year, previous month

If the escape key is hit then the procedure should stop.

Hint: Use the **getkey( )** function discussed in Chapter 8, problem number [L](c).

(f) A factory has 3 division and stocks 4 categories of products. An inventory table is updated for each division and for each product as they are received. There are three independent suppliers of products to the factory:

(a) Design a data format to represent each transaction.
(b) Write a program to take a transaction and update the inventory.
(c) If the cost per item is also given write a program to calculate the total inventory values.

(g) A dequeue is an ordered set of elements in which elements may be inserted or retrieved from either end. Using an array simulate a dequeue of characters and the operations retrieve left, retrieve right, insert left, insert right. Exceptional conditions such as dequeue full or empty should be indicated. Two pointers (namely, left and right) are needed in this simulation.

(h) Write a program to delete all vowels from a sentence. Assume that the sentence is not more than 80 characters long.

(i) Write a program that will read a line and delete from it all occurrences of the word 'the'.

(j) Write a program that takes a set of names of individuals and abbreviates the first, middle and other names except the last name by their first letter.

(k) Write a program to count the number of occurrences of any two vowels in succession in a line of text. For example, in the sentence

"Pleases read this application and give me gratuity"

such occurrences are ea, ea, ui.

10 Structures

- Why Use Structures
  - Declaring a Structure
  - Accessing Structure Elements
  - How Structure Elements are Stored
- Array of Structures
- Additional Features of Structures
- Uses of Structures
- Summary
- Exercise

Which mechanic is good enough who knows how to repair only one type of vehicle? None. Same thing is true about C language. It wouldn't have been so popular had it been able to handle only all **int**s, or all **float**s or all **char**s at a time. In fact when we handle real world data, we don't usually deal with little atoms of information by themselves—things like integers, characters and such. Instead we deal with entities that are collections of things, each thing having its own attributes, just as the entity we call a 'book' is a collection of things such as title, author, call number, publisher, number of pages, date of publication, etc. As you can see all this data is dissimilar, for example author is a string, whereas number of pages is an integer. For dealing with such collections, C provides a data type called 'structure'. A structure gathers together, different atoms of information that comprise a given entity. And structure is the topic of this chapter.

Why Use Structures

We have seen earlier how ordinary variables can hold one piece of information and how arrays can hold a number of pieces of information of the same data type. These two data types can handle a great variety of situations. But quite often we deal with entities that are collection of dissimilar data types.

For example, suppose you want to store data about a book. You might want to store its name (a string), its price (a float) and number of pages in it (an int). If data about say 3 such books is to be stored, then we can follow two approaches:

(a) Construct individual arrays, one for storing names, another for storing prices and still another for storing number of pages.

(b) Use a structure variable.

Let us examine these two approaches one by one. For the sake of programming convenience assume that the names of books would

be single character long. Let us begin with a program that uses arrays.

```
main( )
{
    char name[3] ;
    float price[3] ;
    int pages[3], i ;

    printf ( "\nEnter names, prices and no. of pages of 3 books\n" ) ;

    for ( i = 0 ; i <= 2 ; i++ )
        scanf ( "%c %f %d", &name[i], &price[i], &pages[i] );

    printf ( "\nAnd this is what you entered\n" ) ;
    for ( i = 0 ; i <= 2 ; i++ )
        printf ( "%c %f %d\n", name[i], price[i], pages[i] );
}
```

And here is the sample run...

```
Enter names, prices and no. of pages of 3 books
A 100.00 354
C 256.50 682
F 233.70 512

And this is what you entered
A 100.000000 354
C 256.500000 682
F 233.700000 512
```

This approach no doubt allows you to store names, prices and number of pages. But as you must have realized, it is an unwieldy approach that obscures the fact that you are dealing with a group of characteristics related to a single entity—the book.

The program becomes more difficult to handle as the number of items relating to the book go on increasing. For example, we would be required to use a number of arrays, if we also decide to store name of the publisher, date of purchase of book, etc. To solve this problem, C provides a special data type—the structure.

A structure contains a number of data types grouped together. These data types may or may not be of the same type. The following example illustrates the use of this data type.

```
main( )
{
    struct book
    {
        char name ;
        float price ;
        int pages ;
    } ;
    struct book b1, b2, b3 ;

    printf ( "\nEnter names, prices & no. of pages of 3 books\n" ) ;
    scanf ( "%c %f %d", &b1.name, &b1.price, &b1.pages ) ;
    scanf ( "%c %f %d", &b2.name, &b2.price, &b2.pages ) ;
    scanf ( "%c %f %d", &b3.name, &b3.price, &b3.pages ) ;

    printf ( "\nAnd this is what you entered" ) ;
    printf ( "\n%c %f %d", b1.name, b1.price, b1.pages ) ;
    printf ( "\n%c %f %d", b2.name, b2.price, b2.pages ) ;
    printf ( "\n%c %f %d", b3.name, b3.price, b3.pages ) ;
}
```

And here is the output...

```
Enter names, prices and no. of pages of 3 books
A 100.00 354
C 256.50 682
F 233.70 512
```

```
And this is what you entered
A 100.000000 354
C 256.500000 682
F 233.700000 512
```

This program demonstrates two fundamental aspects of structures:

(a) declaration of a structure.
(b) accessing of structure elements

Let us now look at these concepts one by one.

Declaring a Structure

In our example program, the following statement declares the structure type:

```
struct book
{
    char name ;
    float price ;
    int pages ;
} ;
```

This statement defines a new data type called **struct book**. Each variable of this data type will consist of a character variable called **name**, a float variable called **price** and an integer variable called **pages**. The general form of a structure declaration statement is given below:

```
struct <structure name>
{
    structure element 1 ;
    structure element 2 ;
    structure element 3 ;
    .....
    .....
}
```

```
};
```

Once the new structure data type has been defined one or more variables can be declared to be of that type. For example the variables **b1, b2, b3** can be declared to be of the type **struct book**, as,

```
struct book b1, b2, b3 ;
```

This statement sets aside space in memory. It makes available space to hold all the elements in the structure—in this case, 7 bytes—one for **name**, four for **price** and two for **pages**. These bytes are always in adjacent memory locations.

If we so desire, we can combine the declaration of the structure type and the structure variables in one statement.

For example,

```
struct book
{
    char name ;
    float price ;
    int pages ;
} ;
struct book b1, b2, b3 ;
```

is same as...

```
struct book
{
    char name ;
    float price ;
    int pages ;
} b1, b2, b3 ;
```

or even...

```
struct
```

```
{
    char name ;
    float price ;
    int pages ;
} b1, b2, b3 ;
```

Like primary variables and arrays, structure variables can also be initialized where they are declared. The format used is quite similar to that used to initiate arrays.

```
struct book
{
    char name[10] ;
    float price ;
    int pages ;
} ;
struct book b1 = { "Basic", 130.00, 550 } ;
struct book b2 = { "Physics", 150.80, 800 } ;
```

Note the following points while declaring a structure type:

(a) The closing brace in the structure type declaration must be followed by a semicolon.

(b) It is important to understand that a structure type declaration does not tell the compiler to reserve any space in memory. All a structure declaration does is, it defines the 'form' of the structure.

(c) Usually structure type declaration appears at the top of the source code file, before any variables or functions are defined. In very large programs they are usually put in a separate header file, and the file is included (using the preprocessor directive #include) in whichever program we want to use this structure type.

Accessing Structure Elements

Having declared the structure type and the structure variables, let us see how the elements of the structure can be accessed.

In arrays we can access individual elements of an array using a subscript. Structures use a different scheme. They use a dot (.) operator. So to refer to **pages** of the structure defined in our sample program we have to use,

```
b1.pages
```

Similarly, to refer to **price** we would use,

```
b1.price
```

Note that before the dot there must always be a structure variable and after the dot there must always be a structure element.

How Structure Elements are Stored

Whatever be the elements of a structure, they are always stored in contiguous memory locations. The following program would illustrate this:

```
/* Memory map of structure elements */
main( )
{
    struct book
    {
        char name ;
        float price ;
        int pages ;
    } ;
    struct book b1 = { 'B', 130.00, 550 } ;

    printf ( "\nAddress of name = %u", &b1.name ) ;
```

```
    printf ( "\nAddress of price = %u", &b1.price ) ;
    printf ( "\nAddress of pages = %u", &b1.pages ) ;
}
```

Here is the output of the program...

```
Address of name = 65518
Address of price = 65519
Address of pages = 65523
```

Actually the structure elements are stored in memory as shown in the Figure 10.1.

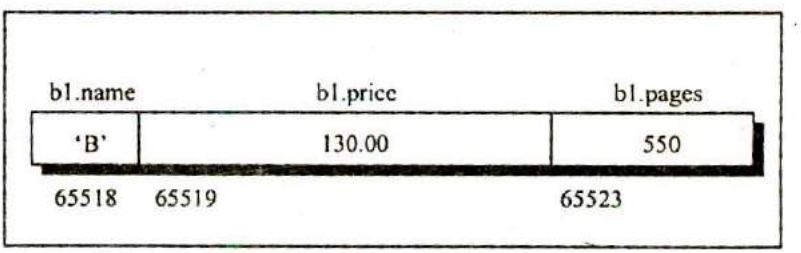


Figure 10.1

Array of Structures

Our sample program showing usage of structure is rather simple minded. All it does is, it receives values into various structure elements and output these values. But that's all we intended to do anyway... show how structure types are created, how structure variables are declared and how individual elements of a structure variable are referenced.

In our sample program, to store data of 100 books we would be required to use 100 different structure variables from **b1** to **b100**, which is definitely not very convenient. A better approach would be to use an array of structures. Following program shows how to use an array of structures.

```
/* Usage of an array of structures */
main( )
{
    struct book
    {
        char name ;
        float price ;
        int pages ;
    } ;

    struct book b[100] ;
    int i ;

    for ( i = 0 ; i <= 99 ; i++ )
    {
        printf ( "\nEnter name, price and pages " ) ;
        scanf ( "%c %f %d", &b[i].name, &b[i].price, &b[i].pages ) ;
    }

    for ( i = 0 ; i <= 99 ; i++ )
        printf ( "\n%c %f %d", b[i].name, b[i].price, b[i].pages ) ;
}

linkfloat( )
{
    float a = 0, *b ;
    b = &a ; /* cause emulator to be linked */
    a = *b ;   /* suppress the warning - variable not used */
}
```

Now a few comments about the program:

(a) Notice how the array of structures is declared...

```
struct book b[100] ;
```

This provides space in memory for 100 structures of the type **struct book**.

(b) The syntax we use to reference each element of the array **b** is similar to the syntax used for arrays of **int**s and **char**s. For example, we refer to zeroth book's price as **b[0].price**. Similarly, we refer first book's pages as **b[1].pages**.

(c) It should be appreciated what careful thought Dennis Ritchie has put into C language. He first defined array as a collection of similar elements; then realized that dissimilar data types that are often found in real life cannot be handled using arrays, therefore created a new data type called structure. But even using structures programming convenience could not be achieved, because a lot of variables (**b1** to **b100** for storing data about hundred books) needed to be handled. Therefore he allowed us to create an array of structures; an array of similar data types which themselves are a collection of dissimilar data types. Hats off to the genius!

(d) In an array of structures all elements of the array are stored in adjacent memory locations. Since each element of this array is a structure, and since all structure elements are always stored in adjacent locations you can very well visualise the arrangement of array of structures in memory. In our example, **b[0]**'s **name**, **price** and **pages** in memory would be immediately followed by **b[1]**'s **name**, **price** and **pages**, and so on.

(e) What is the function **linkfloat( )** doing here? If you don't define it you are bound to get the error "Floating Point Formats Not Linked" with majority of C Compilers. What causes this error to occur? When parsing our source file, if the compiler encounters a reference to the address of a float, it sets a flag to have the linker link in the floating-point emulator. A floating point emulator is used to manipulate floating point numbers in runtime library functions like

scanf( ) and **atof( )**. There are some cases in which the reference to the **float** is a bit obscure and the compiler does not detect the need for the emulator. The most common is using **scanf( )** to read a **float** in an array of structures as shown in our program.

How can we force the formats to be linked? That's where the **linkfloat( )** function comes in. It forces linking of the floating-point emulator into an application. There is no need to call this function, just define it anywhere in your program.

Additional Features of Structures

Let us now explore the intricacies of structures with a view of programming convenience. We would highlight these intricacies with suitable examples:

(a) The values of a structure variable can be assigned to another structure variable of the same type using the assignment operator. It is not necessary to copy the structure elements piece-meal. Obviously, programmers prefer assignment to piece-meal copying. This is shown in the following example.

```
main( )
{
    struct employee
    {
        char name[10] ;
        int age ;
        float salary ;
    } ,
    struct employee e1 = { "Sanjay", 30, 5500.50 } ;
    struct employee e2, e3 ;

    /* piece-meal copying */
    strcpy ( e2.name, e1.name ) ;
    e2.age = e1.age ;
```

```
    e2.salary = e1.salary ;

    /* copying all elements at one go */
    e3 = e2 ;

    printf ( "\n%s %d %f", e1.name, e1.age, e1.salary ) ;
    printf ( "\n%s %d %f", e2.name, e2.age, e2.salary ) ;
    printf ( "\n%s %d %f", e3.name, e3.age, e3.salary ) ;
}
```

The output of the program would be...

```
Sanjay 30 5500.500000
Sanjay 30 5500.500000
Sanjay 30 5500.500000
```

Ability to copy the contents of all structure elements of one variable into the corresponding elements of another structure variable is rather surprising, since C does not allow assigning the contents of one array to another just by equating the two. As we saw earlier, for copying arrays we have to copy the contents of the array element by element.

This copying of all structure elements at one go has been possible only because the structure elements are stored in contiguous memory locations. Had this not been so, we would have been required to copy structure variables element by element. And who knows, had this been so, structures would not have become popular at all.

(b) One structure can be nested within another structure. Using this facility complex data types can be created. The following program shows nested structures at work.

```
main( )
{
    struct address
```

```
    {
        char phone[15] ;
        char city[25] ;
        int pin ;
    } ;

    struct emp
    {
        char name[25] ;
        struct address a ;
    } ;
    struct emp e = { "jeru", "531046", "nagpur", 10 } ;

    printf ( "\nname = %s phone = %s", e.name, e.a.phone ) ;
    printf ( "\ncity = %s pin = %d", e.a.city, e.a.pin ) ;
}
```

And here is the output...

```
name = jeru phone = 531046
city = nagpur pin = 10
```

Notice the method used to access the element of a structure that is part of another structure. For this the dot operator is used twice, as in the expression,

```
e.a.pin or e.a.city
```

Of course, the nesting process need not stop at this level. We can nest a structure within a structure, within another structure, which is in still another structure and so on... till the time we can comprehend the structure ourselves. Such construction however gives rise to variable names that can be surprisingly self descriptive, for example:

```
maruti.engine.bolt.large.qty
```

This clearly signifies that we are referring to the quantity of large sized bolts that fit on an engine of a maruti car.

(c) Like an ordinary variable, a structure variable can also be passed to a function. We may either pass individual structure elements or the entire structure variable at one go. Let us examine both the approaches one by one using suitable programs.

```
/* Passing individual structure elements */
main( )
{
    struct book
    {
        char name[25] ;
        char author[25] ;
        int callno ;
    } ;
    struct book b1 = { "Let us C", "YPK", 101 } ;

    display ( b1.name, b1.author, b1.callno ) ;
}

display ( char *s, char *t, int n )
{
    printf ( "\n%s %s %d", s, t, n ) ;
}
```

And here is the output...

```
Let us C YPK 101
```

Observe that in the declaration of the structure, **name** and **author** have been declared as arrays. Therefore, when we call the function **display( )** using,

```
display ( b1.name, b1.author, b1.callno ) ;
```

we are passing the base addresses of the arrays **name** and **author**, but the value stored in **callno**. Thus, this is a mixed call—a call by reference as well as a call by value.

It can be immediately realized that to pass individual elements would become more tedious as the number of structure elements go on increasing. A better way would be to pass the entire structure variable at a time. This method is shown in the following program.

```
struct book
{
    char name[25] ;
    char author[25] ;
    int callno ;
} ;

main( )
{
    struct book b1 = { "Let us C", "YPK", 101 } ;
    display ( b1 ) ;
}

display ( struct book b )
{
    printf ( "\n%s %s %d", b.name, b.author, b.callno ) ;
}
```

And here is the output...

```
Let us C YPK 101
```

Note that here the calling of function **display( )** becomes quite compact,

```
display ( b1 ) ;
```

Having collected what is being passed to the **display( )** function, the question comes, how do we define the formal arguments in the function. We cannot say,

```
struct book b1 ;
```

because the data type **struct book** is not known to the function **display( )**. Therefore, it becomes necessary to define the structure type **struct book** outside **main( )**, so that it becomes known to all functions in the program.

(d) The way we can have a pointer pointing to an **int**, or a pointer pointing to a **char**, similarly we can have a pointer pointing to a **struct**. Such pointers are known as 'structure pointers'.

Let us look at a program that demonstrates the usage of a structure pointer.

```
main( )
{
    struct book
    {
        char name[25] ;
        char author[25] ;
        int callno ;
    } ;
    struct book b1 = { "Let us C", "YPK", 101 } ;
    struct book *ptr ;

    ptr = &b1 ;
    printf ( "\n%s %s %d", b1.name, b1.author, b1.callno ) ;
    printf ( "\n%s %s %d", ptr->name, ptr->author, ptr->callno ) ;
}
```

The first **printf( )** is as usual. The second **printf( )** however is peculiar. We can't use **ptr.name** or **ptr.callno** because **ptr** is not a structure variable but a pointer to a structure, and the dot

operator requires a structure variable on its left. In such cases C provides an operator ->, called an arrow operator to refer to the structure elements. Remember that on the left hand side of the '.' structure operator, there must always be a structure variable, whereas on the left hand side of the '->' operator there must always be a pointer to a structure. The arrangement of the structure variable and pointer to structure in memory is shown in the Figure 10.2.

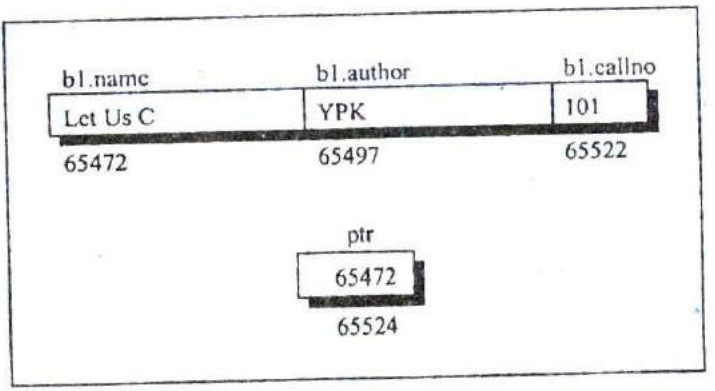


Figure 10.2

Can we not pass the address of a structure variable to a function? We can. The following program demonstrates this.

```
/* Passing address of a structure variable */
struct book
{
    char name[25] ;
    char author[25] ;
    int callno ;
} ;

main( )
{
    struct book b1 = { "Let us C", "YPK", 101 } ;
    display ( &b1 ) ;
```

```
}

display ( struct book *b )
{
    printf ( "\n%s %s %d", b->name, b->author, b->callno ) ;
}
```

And here is the output...

```
Let us C YPK 101
```

Again note that to access the structure elements using pointer to a structure we have to use the '->' operator.

Also, the structure **struct book** should be declared outside **main( )** such that this data type is available to **display( )** while declaring pointer to the structure.

(e) Consider the following code snippet:

```
struct emp
{
    int a ;
    char ch ;
    float s ;
} ;
struct emp e ;
printf ( "%u %u %u", &e.a, &e.ch, &e.s ) ;
```

If we execute this program using TC/TC++ compiler we get the addresses as:

```
65518 65520 65521
```

As expected, in memory the **char** begins immediately after the **int** and **float** begins immediately after the **char**.

However, if we run the same program using VC++ compiler then the output turns but to be:

```
1245044 1245048 1245052
```

It can be observed from this output that the **float** doesn't get stored immediately after the **char**. In fact there is a hole of three bytes after the **char**. Let us understand the reason for this. VC++ is a 32-bit compiler targeted to generate code for a 32-bit microprocessor. The architecture of this microprocessor is such that it is able to fetch the data that is present at an address, which is a multiple of four much faster than the data present at any other address. Hence the VC++ compiler aligns every element of a structure at an address that is multiple of four. That's the reason why there were three holes created between the **char** and the **float**.

However, some programs need to exercise precise control over the memory areas where data is placed. For example, suppose we wish to read the contents of the boot sector (first sector on the floppy/hard disk) into a structure. For this the byte arrangement of the structure elements must match the arrangement of various fields in the boot sector of the disk. The **#pragma pack** directive offers a way to fulfill this requirement. This directive specifies packing alignment for structure members. The pragma takes effect at the first structure declaration after the pragma is seen. Turbo C/C++ compiler doesn't support this feature, VC++ compiler does. The following code shows how to use this directive.

```
#pragma pack(1)
struct emp
{
    int a ;
    char ch ;
    float s ;
} ;
```

```
#pragma pack( )

struct emp e ;
printf ( "%u %u %u", &e.a, &e.ch, &e.s ) ;
```

Here, **#pragma pack (1)** lets each structure element to begin on a 1-byte boundary as justified by the output of the program given below:

```
1245044  1245048  1245049
```

Uses of Structures

Where are structures useful? The immediate application that comes to the mind is Database Management. That is, to maintain data about employees in an organization, books in a library, items in a store, financial accounting transactions in a company etc. But mind you, use of structures stretches much beyond database management. They can be used for a variety of purposes like:

(a) Changing the size of the cursor
(b) Clearing the contents of the screen
(c) Placing the cursor at an appropriate position on screen
(d) Drawing any graphics shape on the screen
(e) Receiving a key from the keyboard
(f) Checking the memory size of the computer
(g) Finding out the list of equipment attached to the computer
(h) Formatting a floppy
(i) Hiding a file from the directory
(j) Displaying the directory of a disk
(k) Sending the output to printer
(l) Interacting with the mouse

And that is certainly a very impressive list! At least impressive enough to make you realize how important a data type a structure is and to be thorough with it if you intend to program any of the

above applications. Some of these applications would be discussed in Chapters 16 to 19.

Summary

(a) A structure is usually used when we wish to store dissimilar data together.
(b) Structure elements can be accessed through a structure variable using a dot (.) operator.
(c) Structure elements can be accessed through a pointer to a structure using the arrow (->) operator.
(d) All elements of one structure variable can be assigned to another structure variable using the assignment (=) operator.
(e) It is possible to pass a structure variable to a function either by value or by address.
(f) It is possible to create an array of structures.

Exercise

[A] What would be the output of the following programs:

(a)
```
main( )
{
    struct gospel
    {
        int num ;
        char mess1[50] ;
        char mess2[50] ;
    } m ;

    m.num = 1 ;
    strcpy ( m.mess1, "If all that you have is hammer" ) ;
    strcpy ( m.mess2, "Everything looks like a nail" ) ;

    /* assume that the strucure is located at address 1004 */
    printf ( "\n%u %s %u", &m.num, m.mess1, m.mess2 ) ;
}
```

(b)
```
struct gospel
{
    int num ;
    char mess1[50] ;
    char mess2[50] ;
} m1 = { 2, "If you are driven by success",
           "make sure that it is a quality drive"
       } ;
main( )
{
    struct gospel m2, m3 ;
    m2 = m1 ;
    m3 = m2 ;
    printf ( "\n%d %s %s", m1.num, m2.mess1, m3.mess2 ) ;
}
```

[B] Point out the errors, if any, in the following programs:

(a)
```
main( )
{
    struct employee
    {
        char name[25] ;
        int age ;
        float bs ;
    } ;
    struct employee e ;
    strcpy ( e.name, "Hacker" ) ;
    age = 25 ;
    printf ( "\n%s %d", e.name, age ) ;
}
```

(b)
```
main( )
{
    struct
    {
        char name[25] ;
```

```
        char language[10] ;
    } ;
    struct employee e = { "Hacker", "C" } ;
    printf ( "\n%s %d", e.name, e.language ) ;
}
```

(c)
```
struct virus
{
    char signature[25] ;
    char status[20] ;
    int size ;
} v[2] = {
            "Yankee Doodle", "Deadly", 1813,
            "Dark Avenger", "Killer", 1795
        } ;
main( )
{
    int i ;
    for ( i = 0 ; i <=1 ; i++ )
        printf ( "\n%s %s", v.signature, v.status ) ;
}
```

(d)
```
struct s
{
    char ch ;
    int i ;
    float a ;
} ;
main( )
{
    struct s var = { 'C', 100, 12.55 } ;
    f ( var ) ;
    g ( &var ) ;
}
f ( struct s v )
{
    printf ( "\n%c %d %f", v -> ch, v -> i, v -> a ) ;
}
```

```
g ( struct s *v )
{
    printf ( "\n%c %d %f", v.ch, v.i, v.a ) ;
}
```

(e)
```
struct s
{
    int i ;
    struct s *p ;
} ;
main( )
{
    struct s var1, var2 ;

    var1.i = 100 ;
    var2.i = 200 ;
    var1.p = &var2 ;
    var2.p = &var1 ;
    printf ( "\n%d %d", var1.p -> i, var2.p -> i ) ;
}
```

[C] Answer the following:

(a) Ten floats are to be stored in memory. What would you prefer, an array or a structure?

(b) Given the statement,

```
maruti.engine.bolts = 25 ;
```

which of the following is True?

1. structure bolts is nested within structure engine
2. structure engine is nested within structure maruti
3. structure maruti is nested within structure engine
4. structure maruti is nested within structure bolts

(c) State True or False:

1. All structure elements are stored in contiguous memory locations.

2. An array should be used to store dissimilar elements, and a structure to store similar elements.

3. In an array of structures, not only are all structures stored in contiguous memory locations, but the elements of individual structures are also stored in contiguous locations.

(d)
```
struct time
{
    int hours ;
    int minutes ;
    int seconds ;
} t ;
struct time *tt ;
tt = &t ;
```

Looking at the above declarations, which of the following refers to **seconds** correctly:

1. tt.seconds
2. (*tt).seconds
3. time.t
4. tt -> seconds

[D] Attempt the following:

(a) Create a structure to specify data on students given below:

Roll number, Name, Department, Course, Year of joining

Assume that there are not more than 450 students in the collage.

(a) Write a function to print names of all students who joined in a particular year.

(b) Write a function to print the data of a student whose roll number is given.

(b) Create a structure to specify data of customers in a bank. The data to be stored is: Account number, Name, Balance in account. Assume maximum of 200 customers in the bank.

(a) Write a function to print the Account number and name of each customer with balance below Rs. 100.

(b) If a customer request for withdrawal or deposit, it is given in the form:

Acct. no, amount, code (1 for deposit, 0 for withdrawal)

Write a program to give a message, "The balance is insufficient for the specified withdrawal".

(c) An automobile company has serial number for engine parts starting from AA0 to FF9. The other characteristics of parts to be specified in a structure are: Year of manufacture, material and quantity manufactured.

(a) Specify a structure to store information corresponding to a part.

(b) Write a program to retrieve information on parts with serial numbers between BB1 and CC6.

(d) A record contains name of cricketer, his age, number of test matches that he has played and the average runs that he has scored in each test match. Create an array of structure to hold records of 20 such cricketer and then write a program to read these records and arrange them in ascending order by average runs. Use the **qusort( )** standard library function.

(e) There is a structure called **employee** that holds information like employee code, name, date of joining. Write a program to create an array of the structure and enter some data into it. Then ask the user to enter current date. Display the names of those employees whose tenure is 3 or more than 3 years according to the given current date.

(f) Write a menu driven program that depicts the working of a library. The menu options should be:

1. Add book information
2. Display book information
3. List all books of given author
4. List the title of specified book
5. List the count of books in the library
6. List the books in the order of accession number
7. Exit

Create a structure called **library** to hold accession number, title of the book, author name, price of the book, and flag indicating whether book is issued or not.

(g) Write a program that compares two given dates. To store date use structure say **date** that contains three members namely date, month and year. If the dates are equal then display message as "Equal" otherwise "Unequal".

(h) Linked list is a very common data structure often used to store similar data in memory. While the elements of an array occupy contiguous memory locations, those of a linked list are not constrained to be stored in adjacent location. The individual elements are stored "somewhere" in memory, rather like a family dispersed, but still bound together. The order of the elements is maintained by explicit links between them. Thus, a linked list is a collection of elements called nodes, each of which stores two item of information—an element of the list, and a link, i.e., a pointer or an address that indicates explicitly the location of the node containing the successor of this list element.

Write a program to build a linked list by adding new nodes at the beginning, at the end or in the middle of the linked list. Also write a function **display( )** which display all the nodes present in the linked list.

(i) A stack is a data structure in which addition of new element or deletion of existing element always takes place at the same

end. This end is often known as 'top' of stack. This situation can be compared to a stack of plates in a cafeteria where every new plate taken off the stack is also from the 'top' of the stack. There are several application where stack can be put to use. For example, recursion, keeping track of function calls, evaluation of expressions, etc. Write a program to implement a stack using a linked list.

(j) Unlike a stack, in a queue the addition of new element takes place at the end (called 'rear' of queue) whereas deletion takes place at the other end (called 'front' of queue). Write a program to implement a queue using a linked list.

11 Console Input/Output

- Types of I/O
- Console I/O Functions
  - Formatted Console I/O Functions
  - *sprintf*( ) and *sscanf*( ) Functions
  - Unformatted Console I/O Functions
- Summary
- Exercise

As mentioned in the first chapter, Dennis Ritchie wanted C to remain compact. In keeping with this intention he deliberately omitted everything related with Input/Output (I/O) from his definition of the language. Thus, C simply has no provision for receiving data from any of the input devices (like say keyboard, disk, etc.), or for sending data to the output devices (like say VDU, disk, etc.). Then how do we manage I/O, and how is it that we were we able to use **printf( )** and **scanf( )** if C has nothing to offer for I/O? This is what we intend to explore in this chapter.

Types of I/O

Though C has no provision for I/O, it of course has to be dealt with at some point or the other. There is not much use writing a program that spends all its time telling itself a secret. Each Operating System has its own facility for inputting and outputting data from and to the files and devices. It's a simple matter for a system programmer to write a few small programs that would link the C compiler for particular Operating system's I/O facilities.

The developers of C Compilers do just that. They write several standard I/O functions and put them in libraries. These libraries are available with all C compilers. Whichever C compiler you are using it's almost certain that you have access to a library of I/O functions.

Do understand that the I/O facilities with different operating systems would be different. Thus, the way one OS displays output on screen may be different than the way another OS does it. For example, the standard library function **printf( )** for DOS-based C compiler has been written keeping in mind the way DOS outputs characters to screen. Similarly, the **printf( )** function for a Unix-based compiler has been written keeping in mind the way Unix outputs characters to screen. We as programmers do not have to bother about which **printf( )** has been written in what manner. We should just use **printf( )** and it would take care of the rest of the

details that are OS dependent. Same is true about all other standard library functions available for I/O.

There are numerous library functions available for I/O. These can be classified into three broad categories:

(a) Console I/O functions - Functions to receive input from keyboard and write output to VDU.

(b) File I/O functions - Functions to perform I/O operations on a floppy disk or hard disk.

In this chapter we would be discussing only Console I/O functions. File I/O functions would be discussed in Chapter 12.

Console I/O Functions

The screen and keyboard together are called a console. Console I/O functions can be further classified into two categories—formatted and unformatted console I/O functions. The basic difference between them is that the formatted functions allow the input read from the keyboard or the output displayed on the VDU to be formatted as per our requirements. For example, if values of average marks and percentage marks are to be displayed on the screen, then the details like where this output would appear on the screen, how many spaces would be present between the two values, the number of places after the decimal points, etc. can be controlled using formatted functions. The functions available under each of these two categories are shown in Figure 11.1. Now let us discuss these console I/O functions in detail.

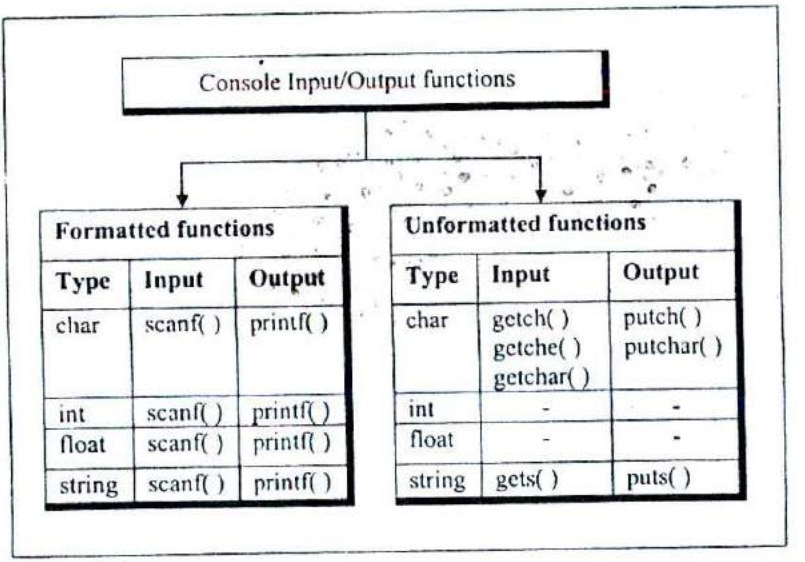


Figure 11.1

Formatted Console I/O Functions

As can be seen from Figure 11.1 the functions **printf( )**, and **scanf( )** fall under the category of formatted console I/O functions. These functions allow us to supply the input in a fixed format and let us obtain the output in the specified form. Let us discuss these functions one by one.

We have talked a lot about **printf( )**, used it regularly, but without having introduced it formally. Well, better late than never. Its general form looks like this...

```
printf ( "format string", list of variables ) ;
```

The format string can contain:

(a) Characters that are simply printed as they are
(b) Conversion specifications that begin with a % sign

(c) Escape sequences that begin with a \ sign

For example, look at the following program:

```
main( )
{
    int avg = 346 ;
    float per = 69.2 ;
    printf ( "Average = %d\nPercentage = %f", avg, per ) ;
}
```

The output of the program would be...

```
Average = 346
Percentage = 69.200000
```

How does **printf( )** function interpret the contents of the format string. For this it examines the format string from left to right. So long as it doesn't come across either a **%** or a \ it continues to dump the characters that it encounters, on to the screen. In this example **Average =** is dumped on the screen. The moment it comes across a conversion specification in the format string it picks up the first variable in the list of variables and prints its value in the specified format. In this example, the moment **%d** is met the variable **avg** is picked up and its value is printed. Similarly, when an escape sequence is met it takes the appropriate action. In this example, the moment **\n** is met it places the cursor at the beginning of the next line. This process continues till the end of format string is not reached.

Format Specifications

The **%d** and **%f** used in the **printf( )** are called format specifiers. They tell **printf( )** to print the value of **avg** as a decimal integer and the value of per as a float. Following is the list of format specifiers that can be used with the **printf( )** function.

Data type		Format specifier
Integer	short signed	%d or %I
	short unsigned	%u
	long singed	%ld
	long unsigned	%lu
	unsigned hexadecimal	%x
	unsigned octal	%o
Real	float	%f
	double	%lf
Character	signed character	%c
	unsigned character	%c
String		%s

Figure 11.2

We can provide following optional specifiers in the format specifications.

Specifier	Description
dd	Digits specifying field width
	Decimal point separating field width from precision (precision stands for the number of places after the decimal point)
dd	Digits specifying precision
	Minus sign for left justifying the output in the specified field width

Figure 11.3

Now a short explanation about these optional format specifiers. The field-width specifier tells **printf( )** how many columns on screen should be used while printing a value. For example, **%10d** says, "print the variable as a decimal integer in a field of 10 columns". If the value to be printed happens not to fill up the entire field, the value is right justified and is padded with blanks on the left. If we include the minus sign in format specifier (as in **%-10d**), this means left justification is desired and the value will be padded with blanks on the right. Here is an example that should make this point clear.

```
main( )
{
    int weight = 63 ;

    printf ( "\nweight is %d kg", weight ) ;
    printf ( "\nweight is %2d kg", weight ) ;
    printf ( "\nweight is %4d kg", weight ) ;
    printf ( "\nweight is %6d kg", weight ) ;
    printf ( "\nweight is %-6d kg", weight ) ;
}
```

The output of the program would look like this ...

```
Columns     01234567890123456789012345678901234567890
            weight is 63 kg
            weight is 63 kg
            weight is 63  kg
            weight is 63 kg
            weight is 63 kg
```

Specifying the field width can be useful in creating tables of numeric values, as the following program demonstrates.

```
main( )
{
    printf ( "\n%f %f %f", 5.0, 13.5, 133.9 ) ;
```

```
    printf ( "\n%f %f %f", 305.0, 1200.9, 3005.3 ) ;
}
```

And here is the output...

```
5.000000 13.500000 133.900000
305.000000 1200.900000 3005.300000
```

Even though the numbers have been printed, the numbers have not been lined up properly and hence are hard to read. A better way would be something like this...

```
main( )
{
    printf ( "\n%10.1f %10.1f %10.1f", 5.0, 13.5, 133.9 ) ;
    printf ( "\n%10.1f %10.1f %10.1f", 305.0, 1200.9, 3005.3 );
}
```

This results into a much better output...

```
01234567890123456789012345678901
       5.0        13.5       133.9
     305.0      1200.9      3005.3
```

The format specifiers could be used even while displaying a string of characters. The following program would clarify this point:

```
/* Formatting strings with printf( ) */
main( )
{
    char firstname1[ ] = "Sandy" ;
    char surname1[ ] = "Malya" ;
    char firstname2[ ] = "AjayKumar" ;
    char surname2[ ] = "Gurubaxani" ;

    printf ( "\n%20s%20s", firstname1, surname1 ) ;
    printf ( "\n%20s%20s", firstname2, surname2 ) ;
```

```
}
```

And here's the output...

```
01234567890123456789012345678901234567890123456789012345678901234567890
                    Sandy                   Malya
                AjayKumar               Gurubaxani
```

The format specifier **%20s** reserves 20 columns for printing a string and then prints the string in these 20 columns with right justification. This helps lining up names of different lengths properly. Obviously, the format **%-20s** would have left justified the string.

Escape Sequences

We saw earlier how the newline character, **\n**, when inserted in a **printf( )**'s format string, takes the cursor to the beginning of the next line. The newline character is an 'escape sequence', so called because the backslash symbol (\) is considered as an 'escape' character—it causes an escape from the normal interpretation of a string, so that the next character is recognized as one having a special meaning.

The following example shows usage of **\n** and a new escape sequence \t, called 'tab'. A \t moves the cursor to the next tab stop. A 80-column screen usually has 10 tab stops. In other words, the screen is divided into 10 zones of 8 columns each. Printing a tab takes the cursor to the beginning of next printing zone. For example, if cursor is positioned in column 5, then printing a tab takes it to column 8.

```
main( )
{
    printf ( "You\tmust\tbe\tcrazy\nto\thate\tthis\tbook" ) ;
}
```

And here's the output...

```
          1         2         3         4
01234567890123456789012345678901234567890
You       must      be        crazy
to        hate      this      book
```

The **\n** character causes a new line to begin following 'crazy'. The tab and newline are probably the most commonly used escape sequences, but there are others as well. Figure 11.4 shows a complete list of these escape sequences.

Esc. Seq.	Purpose	Esc. Seq.	Purpose
\n	New line	\t	Tab
\b	Backspace	\r	Carriage return
\f	Form feed	\a	Alert
\'	Single quote	\"	Double quote
\\	Backslash		

Figure 11.4

The first few of these escape sequences are more or less self-explanatory. **\b** moves the cursor one position to the left of its current position. **\r** takes the cursor to the beginning of the line in which it is currently placed. **\a** alerts the user by sounding the speaker inside the computer. Form feed advances the computer stationery attached to the printer to the top of the next page. Characters that are ordinarily used as delimiters... the single quote, double quote, and the backslash can be printed by preceding them with the backslash. Thus, the statement,

```
printf ( "He said, \"Let's do it!\"" ) ;
```

will print...

He said, "Let's do it!"

So far we have been describing **printf( )**'s specification as if we are forced to use only **%d** for an integer, only **%c** for a char, only **%s** for a string and so on. This is not true at all. In fact, **printf( )** uses the specification that we mention and attempts to perform the specified conversion, and does its best to produce a proper result. Sometimes the result is nonsensical, as in case when we ask it to print a string using **%d**. Sometimes the result is useful, as in the case we ask **printf( )** to print ASCII value of a character using **%d**. Sometimes the result is disastrous and the entire program blows up.

The following program shows a few of these conversions, some sensible, some weird.

```
main( )
{
    char ch = 'z' ;
    int i = 125 ;
    float a = 12.55 ;
    char s[ ] = "hello there !" ;

    printf ( "\n%c %d %f", ch, ch, ch ) ;
    printf ( "\n%s %d %f", s, s, s ) ;
    printf ( "\n%c %d %f",i ,i, i ) ;
    printf ( "\n%f %d\n", a, a ) ;
}
```

And here's the output ...

```
z 122 -936283178250178300000000000000000000000000000.000000
hello there ! 3280 -
936283178250178300000000000000000000000000000.000000
} 125 -936283178250178300000000000000000000000000000.000000
```

```
12.550000 0
```

I would leave it to you to analyze the results by yourselves. Some of the conversions you would find are quite sensible.

Let us now turn our attention to **scanf( )**. **scanf( )** allows us to enter data from keyboard that will be formatted in a certain way.

The general form of **scanf( )** statement is as follows:

```
scanf ( "format string", list of addresses of variables ) ;
```

For example:

```
scanf ( "%d %f %c", &c, &a, &ch ) ;
```

Note that we are sending addresses of variables (addresses are obtained by using '**&**' the 'address of' operator) to **scanf( )** function. This is necessary because the values received from keyboard must be dropped into variables corresponding to these addresses. The values that are supplied through the keyboard must be separated by either blank(s), tab(s), or newline(s). Do not include these escape sequences in the format string.

All the format specifications that we learnt in **printf( )** function are applicable to **scanf( )** function as well.

sprintf( ) and *sscanf( )* Functions

The **sprintf( )** function works similar to the **printf( )** function except for one small difference. Instead of sending the output to the screen as **printf( )** does, this function writes the output to an array of characters. The following program illustrates this.

```
main( )
{
```

```
    int i = 10 ;
    char ch = 'A' ;
    float a = 3.14 ;
    char str[20] ;

    printf ( "\n%d %c %f", i, ch, a ) ;
    sprintf ( str, "%d %c %f", i, ch, a ) ;
    printf ( "\n%s", str ) ;
}
```

In this program the **printf( )** prints out the values of **i**, **ch** and **a** on the screen, whereas **sprintf( )** stores these values in the character array **str**. Since the string **str** is present in memory what is written into **str** using **sprintf( )** doesn't get displayed on the screen. Once **str** has been built, its contents can be displayed on the screen. In our program this was achieved by the second **printf( )** statement.

The counterpart of **sprintf( )** is the **sscanf( )** function. It allows us to read characters from a string and to convert and store them in C variables according to specified formats. The **sscanf( )** function comes in handy for in-memory conversion of characters to values. You may find it convenient to read in strings from a file and then extract values from a string by using **sscanf( )**. The usage of **sscanf( )** is same as **scanf( )**, except that the first argument is the string from which reading is to take place.

Unformatted Console I/O Functions

There are several standard library functions available under this category—those that can deal with a single character and those that can deal with a string of characters. For openers let us look at those which handle one character at a time.

So far for input we have consistently used the **scanf( )** function. However, for some situations the **scanf( )** function has one glaring weakness... you need to hit the Enter key before the function can

digest what you have typed. However, we often want a function that will read a single character the instant it is typed without waiting for the Enter key to be hit. **getch( )** and **getche( )** are two functions which serve this purpose. These functions return the character that has been most recently typed. The 'e' in **getche( )** function means it echoes (displays) the character that you typed to the screen. As against this **getch( )** just returns the character that you typed without echoing it on the screen. **getchar( )** works similarly and echo's the character that you typed on the screen, but unfortunately requires Enter key to be typed following the character that you typed. The difference between **getchar( )** and **fgetchar( )** is that the former is a macro whereas the latter is a function. Here is a sample program that illustrates the use of these functions.

```
main( )
{
    char ch ;

    printf ( "\nPress any key to continue" ) ;
    getch( ) ; /* will not echo the character */

    printf ( "\nType any character" ) ;
    ch = getche( ) ; /* will echo the character typed */

    printf ( "\nType any character" ) ;
    getchar( ) ; /* will echo character, must be followed by enter key */
    printf ( "\nContinue Y/N" ) ;
    fgetchar( ) ; /* will echo character, must be followed by enter key */
}
```

And here is a sample run of this program...

```
Press any key to continue
Type any character B
Type any character W
Continue Y/N Y
```

putch( ) and **putchar( )** form the other side of the coin. They print a character on the screen. As far as the working of **putch( ) putchar( )** and **fputchar( )** is concerned it's exactly same. The following program illustrates this.

```
main( )
{
    char ch = 'A' ;

    putch ( ch ) ;
    putchar ( ch ) ;
    fputchar ( ch ) ;
    putch ( 'Z' ) ;
    putchar ( 'Z' ) ;
    fputchar ( 'Z' ) ;
}
```

And here is the output...

```
AAAZZZ
```

The limitation of **putch( ), putchar( )** and **fputchar( )** is that they can output only one character at a time.

gets( ) and *puts( )*

gets( ) receives a string from the keyboard. Why is it needed? Because **scanf( )** function has some limitations while receiving string of characters, as the following example illustrates...

```
main( )
{
    char name[50] ;

    printf ( "\nEnter name " ) ;
    scanf ( "%s", name ) ;
    printf ( "%s", name ) ;
```

```
}
```

And here is the output...

```
Enter name Jonty Rhodes
Jonty
```

Surprised? Where did "Rhodes" go? It never got stored in the array **name[ ]**, because the moment the blank was typed after "Jonty" **scanf( )** assumed that the name being entered has ended. The result is that there is no way (at least not without a lot of trouble on the programmer's part) to enter a multi-word string into a single variable (**name** in this case) using **scanf( )**. The solution to this problem is to use **gets( )** function. As said earlier, it gets a string from the keyboard. It is terminated when an Enter key is hit. Thus, spaces and tabs are perfectly acceptable as part of the input string. More exactly, **gets( )** gets a newline (**\n**) terminated string of characters from the keyboard and replaces the **\n** with a **\0**.

The **puts( )** function works exactly opposite to **gets( )** function. It outputs a string to the screen.

Here is a program which illustrates the usage of those functions:

```
main( )
{
    char footballer[40] ;

    puts ( "Enter name" ) ;
    gets ( footballer ) ; /* sends base address of array */
    puts ( "Happy footballing!" ) ;
    puts ( footballer ) ;
}
```

Following is the sample output:

```
Enter name
```

```
Jonty Rhodes
Happy footballing!
Jonty Rhodes
```

Why did we use two **puts( )** functions to print "Happy footballing!" and "Jonty Rhodes"? Because, unlike **printf( ), puts( )** can output only one string at a time. If we attempt to print two strings using **puts( )**, only the first one gets printed. Similarly, unlike **scanf( ), gets( )** can be used to read only one string at a time.

Summary

(a) There is no keyword available in C for doing input/output.
(b) All I/O in C is done using standard library functions.
(c) There are several functions available for performing console input/output.
(d) The formatted console I/O functions can force the user to receive the input in a fixed format and display the output in a fixed format.
(e) There are several format specifiers and escape sequences available to format input and output.
(f) Unformatted console I/O functions work faster since they do not have the overheads of formatting the input or output.

Exercise

[A] What would be the output of the following programs:

(a)
```
main( )
{
    char ch ;
    ch = getchar( ) ;
    if ( islower ( ch ) )
        putchar ( toupper ( ch ) ) ;
    else
        putchar ( tolower ( ch ) ) ;
```

```
    }
(b) main( )
    {
        int i = 2 ;
        float f = 2.5367 ;
        char str[ ] = "Life is like that" ;

        printf ( "\n%4d\t%3.3f\t%4s", i, f, str ) ;
    }
(c) main( )
    {
        printf ( "More often than \b\b not \rthe person who \
                  wins is the one who thinks he can!" ) ;
    }
(d) char p[ ] = "The sixth sick sheikh's sixth ship is sick" ;
    main( )
    {
        int i = 0 ;
        while ( p[i] != '\0' )
        {
            putch ( p[i] ) ;
            i++ ;
        }
    }
```

[B] Point out the errors, if any, in the following programs:

```
(a) main( )
    {
        int i ;
        char a[ ] = "Hello" ;
        while ( a != '\0' )
        {
            printf ( "%c", *a ) ;
            a++ ;
        }
    }
```

(b)
```
main( )
{
    double dval ;
    scanf ( "%f", &dval ) ;
    printf ( "\nDouble Value = %lf", dval ) ;
}
```

(c)
```
main( )
{
    int ival ;
    scanf ( "%d\n", &n ) ;
    printf ( "\nInteger Value = %d", ival ) ;
}
```

(d)
```
main( )
{
    char *mess[5] ;
    for ( i = 0 ; i < 5 ; i++ )
        scanf ( "%s", mess[i] ) ;
}
```

(e)
```
main( )
{
    int dd, mm, yy ;
    printf ( "\nEnter day, month and year\n" ) ;
    scanf ( "%d%*c%d%*c%d", &dd, &mm, &yy ) ;
    printf ( "The date is: %d - %d - %d", dd, mm, yy ) ;
}
```

(f)
```
main( )
{
    char text ;
    sprintf ( text, "%4d\t%2.2f\n%s", 12, 3.452, "Merry Go Round" ) ;
    printf ( "\n%s", text ) ;
}
```

(g)
```
main( )
{
    char buffer[50] ;
```

```
    int no = 97;
    double val = 2.34174 ;
    char name[10] = "Shweta" ;

    sprintf ( buffer, "%d %lf %s", no, val, name ) ;
    printf ( "\n%s", buffer ) ;
    sscanf ( buffer "%4d %2.2lf %s", &no, &val, name ) ;
    printf ( "\n%s", buffer ) ;
    printf ( "\n%d %lf %s", no, val, name ) ;

}
```

[C] Answer the following:

(a) To receive the string "We have got the guts, you get the glory!!" in an array **char str[100]** which of the following functions would you use?

1. scanf ("%s", str) ;
2. gets (str) ;
3. getche (str) ;
4. fgetchar (str) ;

(b) Which function would you use if a single key were to be received through the keyboard?

1. scanf( )
2. gets( )
3. getche( )
4. getchar( )

(c) If an integer is to be entered through the keyboard, which function would you use?

1. scanf( )
2. gets( )
3. getche( )
4. getchar( )

(d) If a character string is to be received through the keyboard which function would work faster?

1. scanf( )
2. gets( )

(e) What is the difference between **getchar( )**, **fgetchar( )**, **getch( )** and **getche( )**?

(f) The format string of a **printf( )** function can contain:

1. Characters, format specifications and escape sequences
2. Character, integers and floats
3. Strings, integers and escape sequences
4. Inverted commas, percentage sign and backslash character

(g) A field-width specifier in a **printf( )** function:

1. Controls the margins of the program listing
2. Specifies the maximum value of a number
3. Controls the size of type used to print numbers
4. Specifies how many columns will be used to print the number

[D] Answer the following:

(a) Write down two functions **xgets( )** and **xputs( )** which work similar to the standard library functions **gets( )** and **puts( )**.

(b) Write down a function **getint( )**, which would receive a numeric string from the keyboard, convert it to an integer number and return the integer to the calling function. A sample usage of **getint( )** is shown below:

```
main( )
{
   int a ;
```

```
    a = getint( ) ;
    printf ( "you entered %d", a )
}
```

12 File Input/Output

- Data Organization
- File Operations
  - Opening a File
  - Reading from a File
  - Trouble in Opening a File
  - Closing the File
- Counting Characters, Tabs, Spaces, ...
- A File-copy Program
  - Writing to a File
- File Opening Modes
- String (line) I/O in Files
  - The Awkward Newline
- Record I/O in Files
- Text Files and Binary Files
- Record I/O Revisited
- Database Management
- Low Level Disk I/O
  - A Low Level File-copy Program
- I/O Under Windows
- Summary
- Exercise

Often it is not enough to just display the data on the screen. This is because if the data is large, only a limited amount of it can be stored in memory and only a limited amount of it can be displayed on the screen. It would be inappropriate to store this data in memory for one more reason. Memory is volatile and its contents would be lost once the program is terminated. So if we need the same data again it would have to be either entered through the keyboard again or would have to be regenerated programmatically. Obviously both these operations would be tedious. At such times it becomes necessary to store the data in a manner that can be later retrieved and displayed either in part or in whole. This medium is usually a 'file' on the disk. This chapter discusses how file I/O operations can be performed.

Data Organization

Before we start doing file input/output let us first find out how data is organized on the disk. All data stored on the disk is in binary form. How this binary data is stored on the disk varies from one OS to another. However, this does not affect the C programmer since he has to use only the library functions written for the particular OS to be able to perform input/output. It is the compiler vendor's responsibility to correctly implement these library functions by taking the help of OS. This is illustrated in Figure 12.1.

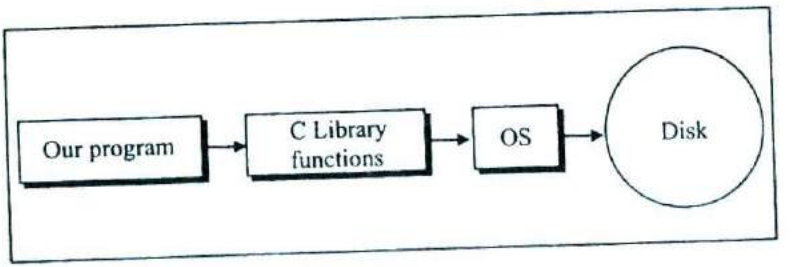


Figure 12.1

File Operations.

There are different operations that can be carried out on a file. These are:

(a) Creation of a new file
(b) Opening an existing file
(c) Reading from a file
(d) Writing to a file
(e) Moving to a specific location in a file (seeking)
(f) Closing a file

Let us now write a program to read a file and display its contents on the screen. We will first list the program and show what it does, and then dissect it line by line. Here is the listing...

```
/* Display contents of a file on screen. */
# include "stdio.h"
main( )
{
    FILE *fp ;
    char ch ;

    fp = fopen ( "PR1.C", "r" ) ;

    while ( 1 )
    {
        ch = fgetc ( fp ) ;

        if ( ch == EOF )
            break ;

        printf ( "%c", ch ) ;
    }

    fclose ( fp ) ;
}
```

On execution of this program it displays the contents of the file 'PR1.C' on the screen. Let us now understand how it does the same.

Opening a File

Before we can read (or write) information from (to) a file on a disk we must open the file. To open the file we have called the function **fopen( )**. It would open a file "PR1.C" in 'read' mode, which tells the C compiler that we would be reading the contents of the file. Note that "r" is a string and not a character; hence the double quotes and not single quotes. In fact **fopen( )** performs three important tasks when you open the file in "r" mode:

(a) Firstly it searches on the disk the file to be opened.
(b) Then it loads the file from the disk into a place in memory called buffer.
(c) It sets up a character pointer that points to the first character of the buffer.

Why do we need a buffer at all? Imagine how inefficient it would be to actually access the disk every time we want to read a character from it. Every time we read something from a disk, it takes some time for the disk drive to position the read/write head correctly. On a floppy disk system, the drive motor has to actually start rotating the disk from a standstill position every time the disk is accessed. If this were to be done for every character we read from the disk, it would take a long time to complete the reading operation. This is where a buffer comes in. It would be more sensible to read the contents of the file into the buffer while opening the file and then read the file character by character from the buffer rather than from the disk. This is shown in Figure 12.2.

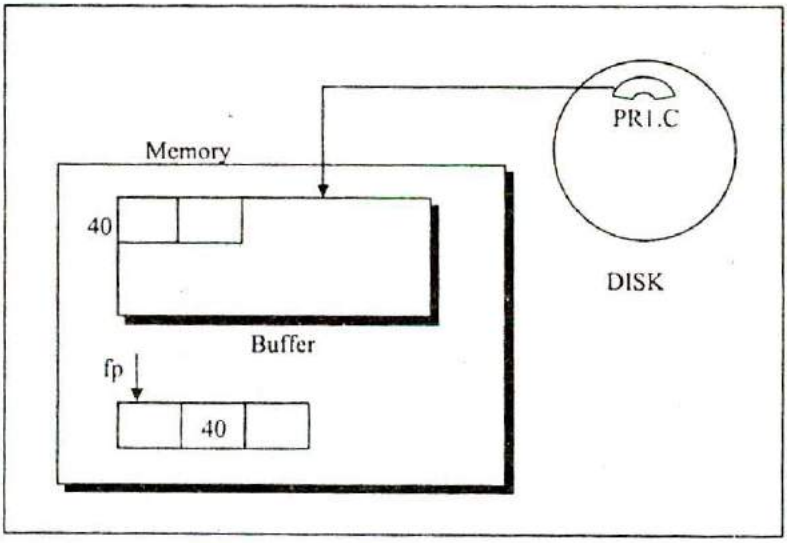


Figure 12.2

Same argument also applies to writing information in a file. Instead of writing characters in the file on the disk one character at a time it would be more efficient to write characters in a buffer and then finally transfer the contents from the buffer to the disk.

To be able to successfully read from a file information like mode of opening, size of file, place in the file from where the next read operation would be performed, etc. has to be maintained. Since all this information is inter-related, all of it is gathered together by **fopen( )** in a structure called **FILE**. **fopen( )** returns the address of this structure, which we have collected in the structure pointer called **fp**. We have declared **fp** as

```
FILE *fp ;
```

The **FILE** structure has been defined in the header file "stdio.h" (standing for standard input/output header file). Therefore, it is necessary to **#include** this file.

Reading from a File

Once the file has been opened for reading using **fopen( )**, as we have seen, the file's contents are brought into buffer (partly or wholly) and a pointer is set up that points to the first character in the buffer. This pointer is one of the elements of the structure to which **fp** is pointing (refer Figure 12.2).

To read the file's contents from memory there exists a function called **fgetc( )**. This has been used in our program as,

```
ch = fgetc ( fp ) ;
```

fgetc( ) reads the character from the current pointer position, advances the pointer position so that it now points to the next character, and returns the character that is read, which we collected in the variable **ch**. Note that once the file has been opened, we no longer refer to the file by its name, but through the file pointer **fp**.

We have used the function **fgetc( )** within an indefinite **while** loop. There has to be a way to break out of this **while**. When shall we break out... the moment we reach the end of file. But what is end of file? A special character, whose ASCII value is 26, signifies end of file. This character is inserted beyond the last character in the file, when it is created.

While reading from the file, when **fgetc( )** encounters this special character, instead of returning the character that it has read, it returns the macro EOF. The EOF macro has been defined in the file "stdio.h". In place of the function **fgetc( )** we could have as well used the macro **getc( )** with the same effect.

In our program we go on reading each character from the file till end of file is not met. As each character is read we display it on the screen. Once out of the loop, we close the file.

Trouble in Opening a File

There is a possibility that when we try to open a file using the function **fopen( )**, the file may not be opened. While opening the file in "r" mode, this may happen because the file being opened may not be present on the disk at all. And you obviously cannot read a file that doesn't exist. Similarly, while opening the file for writing, **fopen( )** may fail due to a number of reasons, like, disk space may be insufficient to open a new file, or the disk may be write protected or the disk is damaged and so on.

Crux of the matter is that it is important for any program that accesses disk files to check whether a file has been opened successfully before trying to read or write to the file. If the file opening fails due to any of the several reasons mentioned above, the **fopen( )** function returns a value NULL (defined in "stdio.h" as **#define NULL 0**). Here is how this can be handled in a program...

```
#include "stdio.h"
main( )
{
    FILE *fp ;

    fp = fopen ( "PR1.C", "r" ) ;
    if ( fp == NULL )
    {
        puts ( "cannot open file" ) ;
        exit( ) ;
    }
}
```

Closing the File

When we have finished reading from the file, we need to close it. This is done using the function **fclose( )** through the statement,

```
fclose ( fp ) ;
```

Once we close the file we can no longer read from it using **getc( )** unless we reopen the file. Note that to close the file we don't use the filename but the file pointer **fp**. On closing the file the buffer associated with the file is removed from memory.

In this program we have opened the file for reading. Suppose we open a file with an intention to write characters into it. This time too a buffer would get associated with it. When we attempt to write characters into this file using **fputc( )** the characters would get written to the buffer. When we close this file using **fclose( )** three operations would be performed:

(a) The characters in the buffer would be written to the file on the disk.
(b) At the end of file a character with ASCII value 26 would get written.
(c) The buffer would be eliminated from memory.

You can imagine a possibility when the buffer may become full before we close the file. In such a case the buffer's contents would be written to the disk the moment it becomes full. All this buffer management is done for us by the library functions.

Counting Characters, Tabs, Spaces, ...

Having understood the first file I/O program in detail let us now try our hand at one more. Let us write a program that will read a file and count how many characters, spaces, tabs and newlines are present in it. Here is the program...

```
/* Count chars, spaces, tabs and newlines in a file */
# include "stdio.h"
main( )
{
    FILE *fp ;
    char ch ;
    int nol = 0, not = 0, nob = 0, noc = 0 ;

    fp = fopen ( "PR1.C", "r" ) ;

    while ( 1 )
    {
        ch = fgetc ( fp ) ;

        if ( ch == EOF )
            break ;

        noc++ ;

        if ( ch == ' ' )
            nob++ ;

        if ( ch == '\n' )
            nol++ ;

        if ( ch == '\t' )
            not++ ;
    }

    fclose ( fp ) ;
    printf ( "\nNumber of characters = %d", noc ) ;
    printf ( "\nNumber of blanks = %d", nob ) ;
    printf ( "\nNumber of tabs = %d", not ) ;
    printf ( "\nNumber of lines = %d", nol ) ;
}
```

Here is a sample run...

```
Number of characters = 125
Number of blanks = 25
Number of tabs = 13
Number of lines = 22
```

The above statistics are true for a file "PR1.C", which I had on my disk. You may give any other filename and obtain different results. I believe the program is self-explanatory.

In this program too we have opened the file for reading and then read it character by character. Let us now try a program that needs to open a file for writing.

A File-copy Program

We have already used the function **fgetc( )** which reads characters from a file. Its counterpart is a function called **fputc( )** which writes characters to a file. As a practical use of these character I/O functions we can copy the contents of one file into another, as demonstrated in the following program. This program takes the contents of a file and copies them into another file, character by character.

```
#include "stdio.h"
main( )
{
    FILE *fs, *ft ;
    char ch ;

    fs = fopen ( "pr1.c", "r" ) ;
    if ( fs == NULL )
    {
        puts ( "Cannot open source file" ) ;
        exit( ) ;
```

```
    }

    ft = fopen ( "pr2.c", "w" ) ;
    if ( ft == NULL )
    {
        puts ( "Cannot open target file" ) ;
        fclose ( fs ) ;
        exit( ) ;
    }

    while ( 1 )
    {
        ch = fgetc ( fs ) ;

        if ( ch == EOF )
            break ;
        else
            fputc ( ch, ft ) ;
    }

    fclose ( fs ) ;
    fclose ( ft ) ;
}
```

I hope most of the stuff in the program can be easily understood, since it has already been dealt with in the earlier section. What is new is only the function **fputc( )**. Let us see how it works.

Writing to a File

The **fputc( )** function is similar to the **putch( )** function, in the sense that both output characters. However, **putch( )** function always writes to the VDU, whereas, **fputc( )** writes to the file. Which file? The file signified by **ft**. The writing process continues till all characters from the source file have been written to the target file, following which the **while** loop terminates.

Note that our sample file-copy program is capable of copying only text files. To copy files with extension .EXE or .COM, we need to open the files in binary mode, a topic that would be dealt with in sufficient detail in a later section.

File Opening Modes

In our first program on disk I/O we have opened the file in read ("r") mode. However, "r" is but one of the several modes in which we can open a file. Following is a list of all possible modes in which a file can be opened. The tasks performed by **fopen( )** when a file is opened in each of these modes are also mentioned.

"r" Searches file. If the file is opened successfully **fopen( )** loads it into memory and sets up a pointer which points to the first character in it. If the file cannot be opened **fopen( )** returns NULL.

Operations possible – reading from the file.

"w" Searches file. If the file exists, its contents are overwritten. If the file doesn't exist, a new file is created. Returns NULL, if unable to open file.

Operations possible – writing to the file.

"a" Searches file. If the file is opened successfully **fopen( )** loads it into memory and sets up a pointer that points to the last character in it. If the file doesn't exist, a new file is created. Returns NULL, if unable to open file.

Operations possible - adding new contents at the end of file.

"r+" Searches file. If is opened successfully **fopen( )** loads it into memory and sets up a pointer which points to the first character in it. Returns NULL, if unable to open the file.

Operations possible - reading existing contents, writing new contents, modifying existing contents of the file.

"w+" Searches file. If the file exists, its contents are overwritten. If the file doesn't exist a new file is created. Returns NULL, if unable to open file.

Operations possible - writing new contents, reading them back and modifying existing contents of the file.

"a+" Searches file. If the file is opened successfully **fopen( )** loads it into memory and sets up a pointer which points to the first character in it. If the file doesn't exist, a new file is created. Returns NULL, if unable to open file.

Operations possible - reading existing contents, appending new contents to end of file. Cannot modify existing contents.

String (line) I/O in Files

For many purposes, character I/O is just what is needed. However, in some situations the usage of functions that read or write entire strings might turn out to be more efficient.

Reading or writing strings of characters from and to files is as easy as reading and writing individual characters. Here is a program that writes strings to a file using the function **fputs( )**.

```
/* Receives strings from keyboard and writes them to file */
#include "stdio.h"
main( )
{
    FILE *fp ;
    char s[80] ;
```

```
    fp = fopen ( "POEM.TXT", "w" ) ;
    if ( fp == NULL )
    {
        puts ( "Cannot open file" ) ;
        exit( ) ;
    }

    printf ( "\nEnter a few lines of text:\n" ) ;
    while ( strlen ( gets ( s ) ) > 0 )
    {
        fputs ( s, fp ) ;
        fputs ( "\n", fp ) ;
    }

    fclose ( fp ) ;
}
```

And here is a sample run of the program...

```
Enter a few lines of text:
Shining and bright, they are forever,
so true about diamonds,
more so of memories,
especially yours !
```

Note that each string is terminated by hitting enter. To terminate the execution of the program, hit enter at the beginning of a line. This creates a string of zero length, which the program recognizes as the signal to close the file and exit.

We have set up a character array to receive the string; the **fputs( )** function then writes the contents of the array to the disk. Since **fputs( )** does not automatically add a newline character to the end of the string, we must do this explicitly to make it easier to read the string back from the file.

Here is a program that reads strings from a disk file.

```
/* Reads strings from the file and displays them on screen */
#include "stdio.h"
main( )
{
    FILE *fp ;
    char s[80] ;

    fp = fopen ( "POEM.TXT", "r" ) ;
    if ( fp == NULL )
    {
        puts ( "Cannot open file" ) ;
        exit( ) ;
    }

    while ( fgets ( s, 79, fp ) != NULL )
        printf ( "%s" , s ) ;

    fclose ( fp ) ;
}
```

And here is the output...

```
Shining and bright, they are forever,
so true about diamonds,
more so of memories,
especially yours !
```

The function **fgets( )** takes three arguments. The first is the address where the string is stored, and the second is the maximum length of the string. This argument prevents **fgets( )** from reading in too long a string and overflowing the array. The third argument, as usual, is the pointer to the structure **FILE**. When all the lines from the file have been read, we attempt to read one more line, in which case **fgets( )** returns a **NULL**.

The Awkward Newline

We had earlier written a program that counts the total number of characters present in a file. If we use that program to count the number of characters present in the above poem (stored in the file "POEM.TXT"), it would give us the character count as 101. The same file if seen in the directory, would be reported to contain 105 characters.

This discrepancy occurs because when we attempt to write a "\n" to the file using **fputs( )**, **fputs( )** converts the **\n** to **\r\n** combination. Here **\r** stands for carriage return and **\n** for linefeed. If we read the same line back using **fgets( )** the reverse conversion happens. Thus when we write the first line of the poem and a "\n" using two calls to **fputs( )**, what gets written to the file is

```
Shining and bright, they are forever,\r\n
```

When the same line is read back into the array **s[ ]** using **fgets( )**, the array contains

```
Shining and bright, they are forever,\n\0
```

Thus conversion of **\n** to **\r\n** during writing and **\r\n** conversion to **\n** during reading is a feature of the standard library functions and not that of the OS. Hence the OS counts **\r** and **\n** as separate characters. In our poem there are four lines, therefore there is a discrepancy of four characters (105 - 101).

Record I/O in Files

So far we have dealt with reading and writing only characters and strings. What if we want to read or write numbers from/to file? Furthermore, what if we desire to read/write a combination of characters, strings and numbers? For this first we would organize this dissimilar data together in a structure and then use **fprintf( )**

and **fscanf( )** library functions to read/write data from/to file. Following program illustrates the use of structures for writing records of employees.

```
/* Writes records to a file using structure */
#include "stdio.h"
main( )
{
    FILE *fp ;
    char another = 'Y' ;
    struct emp
    {
        char name[40] ;
        int age ;
        float bs ;
    } ;
    struct emp e ;

    fp = fopen ( "EMPLOYEE.DAT", "w" ) ;

    if ( fp == NULL )
    {
        puts ( "Cannot open file" ) ;
        exit( ) ;
    }

    while ( another == 'Y' )
    {
        printf ( "\nEnter name, age and basic salary: " ) ;
        scanf ( "%s %d %f", e.name, &e.age, &e.bs ) ;
        fprintf ( fp, "%s %d %f\n", e.name, e.age, e.bs ) ;

        printf ( "Add another record (Y/N) " ) ;
        fflush ( stdin ) ;
        another = getche( ) ;
    }

    fclose ( fp ) ;
```

}

And here is the output of the program...

```
Enter name, age and basic salary: Sunil 34 1250.50
Add another record (Y/N) Y
Enter name, age and basic salary: Sameer 21 1300.50
Add another record (Y/N) Y
Enter name, age and basic salary: Rahul 34 1400.55
Add another record (Y/N) N
```

In this program we are just reading the data into a structure variable using **scanf( )**, and then dumping it into a disk file using **fprintf( )**. The user can input as many records as he desires. The procedure ends when the user supplies 'N' for the question 'Add another record (Y/N)'.

The key to this program is the function **fprintf( )**, which writes the values in the structure variable to the file. This function is similar to **printf( )**, except that a **FILE** pointer is included as the first argument. As in **printf( )**, we can format the data in a variety of ways, by using **fprintf( )**. In fact all the format conventions of **printf( )** function work with **fprintf( )** as well.

Perhaps you are wondering what for have we used the function **fflush( )**. The reason is to get rid of a peculiarity of **scanf( )**. After supplying data for one employee, we would hit the enter key. What **scanf( )** does is it assigns name, age and salary to appropriate variables and keeps the enter key unread in the keyboard buffer. So when it's time to supply Y or N for the question 'Another employee (Y/N)', **getch( )** will read the enter key from the buffer thinking that user has entered the enter key. To avoid this problem we use the function **fflush( )**. It is designed to remove or 'flush out' any data remaining in the buffer. The argument to **fflush( )** must be the buffer which we want to flush out. Here we have used 'stdin', which means buffer related with standard input device—keyboard.

Let us now write a program that reads the employee records created by the above program. Here is how it can be done...

```
/* Read records from a file using structure */
#include "stdio.h"
main( )
{
    FILE *fp ;
    struct emp
    {
        char name[40] ;
        int age ;
        float bs ;
    } ;
    struct emp e ;

    fp = fopen ( "EMPLOYEE.DAT", "r" ) ;

    if ( fp == NULL )
    {
        puts ( "Cannot open file" ) ;
        exit( ) ;
    }

    while ( fscanf ( fp, "%s %d %f", e.name, &e.age, &e.bs ) != EOF )
        printf ( "\n%s %d %f", e.name, e.age, e.bs ) ;

    fclose ( fp ) ;
}
```

And here is the output of the program...

```
Sunil 34 1250.500000
Sameer 21 1300.500000
Rahul 34 1400.500000
```

Text Files and Binary Files

All the programs that we wrote in this chapter so far worked on text files. Some of them would not work correctly on binary files. A text file contains only textual information like alphabets, digits and special symbols. In actuality the ASCII codes of these characters are stored in text files. A good example of a text file is any C program, say PR1.C.

As against this, a binary file is merely a collection of bytes. This collection might be a compiled version of a C program (say PR1.EXE), or music data stored in a wave file or a picture stored in a graphic file. A very easy way to find out whether a file is a text file or a binary file is to open that file in Turbo C/C++. If on opening the file you can make out what is displayed then it is a text file, otherwise it is a binary file.

As mentioned while explaining the file-copy program, the program cannot copy binary files successfully. We can improve the same program to make it capable of copying text as well as binary files as shown below.

```
#include "stdio.h"
main( )
{
    FILE *fs, *ft ;
    int ch ;

    fs = fopen ( "pr1.exe", "rb" ) ;
    if ( fs == NULL )
    {
        puts ( "Cannot open source file" ) ;
        exit( ) ;
    }

    ft = fopen ( "newpr1.exe", "wb" ) ;
```

```
    if ( ft == NULL )
    {
        puts ( "Cannot open target file" ) ;
        fclose ( fs ) ;
        exit( ) ;
    }

    while ( 1 )
    {
        ch = fgetc ( fs ) ;

        if ( ch == EOF )
            break ;
        else
            fputc ( ch, ft ) ;
    }

    fclose ( fs ) ;
    fclose ( ft ) ;
}
```

Using this program we can comfortably copy text as well as binary files. Note that here we have opened the source and target files in "rb" and "wb" modes respectively. While opening the file in text mode we can use either "r" or "rt", but since text mode is the default mode we usually drop the 't'.

From the programming angle there are three main areas where text and binary mode files are different. These are:

(a) Handling of newlines
(b) Representation of end of file
(c) Storage of numbers

Let us explore these three differences.

Text versus Binary Mode: Newlines

We have already seen that, in text mode, a newline character is converted into the carriage return-linefeed combination before being written to the disk. Likewise, the carriage return-linefeed combination on the disk is converted back into a newline when the file is read by a C program. However, if a file is opened in binary mode, as opposed to text mode, these conversions will not take place.

Text versus Binary Mode: End of File

The second difference between text and binary modes is in the way the end-of-file is detected. In text mode, a special character, whose ASCII value is 26, is inserted after the last character in the file to mark the end of file. If this character is detected at any point in the file, the read function would return the EOF signal to the program.

As against this, there is no such special character present in the binary mode files to mark the end of file. The binary mode files keep track of the end of file from the number of characters present in the directory entry of the file.

There is a moral to be derived from the end of file marker of text mode files. If a file stores numbers in binary mode, it is important that binary mode only be used for reading the numbers back, since one of the numbers we store might well be the number 26 (hexadecimal 1A). If this number is detected while we are reading the file by opening it in text mode, reading would be terminated prematurely at that point.

Thus the two modes are not compatible. See to it that the file that has been written in text mode is read back only in text mode. Similarly, the file that has been written in binary mode must be read back only in binary mode.

Text versus Binary Mode: Storage of Numbers

The only function that is available for storing numbers in a disk file is the **fprintf( )** function. It is important to understand how numerical data is stored on the disk by **fprintf( ).** Text and characters are stored one character per byte, as we would expect. Are numbers stored as they are in memory, two bytes for an integer, four bytes for a float, and so on? No.

Numbers are stored as strings of characters. Thus, 1234, even though it occupies two bytes in memory, when transferred to the disk using **fprintf( )**, would occupy four bytes, one byte per character. Similarly, the floating-point number 1234.56 would occupy 7 bytes on disk. Thus, numbers with more digits would require more disk space.

Hence if large amount of numerical data is to be stored in a disk file, using text mode may turn out to be inefficient. The solution is to open the file in binary mode and use those functions (**fread( )** and **fwrite( )** which are discussed later) which store the numbers in binary format. It means each number would occupy same number of bytes on disk as it occupies in memory.

Record I/O Revisited

The record I/O program that we did in an earlier section has two disadvantages:

(a) The numbers (basic salary) would occupy more number of bytes, since the file has been opened in text mode. This is because when the file is opened in text mode, each number is stored as a character string.

(b) If the number of fields in the structure increase (say, by adding address, house rent allowance etc.), writing structures

using **fprintf( )**, or reading them using **fscanf( )**, becomes quite clumsy.

Let us now see a more efficient way of reading/writing records (structures). This makes use of two functions **fread( )** and **fwrite( )**. We will write two programs, first one would write records to the file and the second would read these records from the file and display them on the screen.

```
/* Receives records from keyboard and writes them to a file in binary mode */
#include "stdio.h"
main( )
{
    FILE *fp ;
    char another = 'Y' ;
    struct emp
    {
        char name[40] ;
        int age ;
        float bs ;
    } ;
    struct emp e ;

    fp = fopen ( "EMP.DAT", "wb" ) ;

    if ( fp == NULL )
    {
        puts ( "Cannot open file" ) ;
        exit( ) ;
    }

    while ( another == 'Y' )
    {
        printf ( "\nEnter name, age and basic salary: " ) ;
        scanf ( "%s %d %f", e.name, &e.age, &e.bs ) ;
        fwrite ( &e, sizeof ( e ), 1, fp ) ;

        printf ( "Add another record (Y/N) " ) ;
```

```
        fflush ( stdin ) ;
        another = getche( ) ;
    }

    fclose ( fp ) ;
}
```

And here is the output...

```
Enter name, age and basic salary: Suresh 24 1250.50
Add another record (Y/N) Y
Enter name, age and basic salary: Ranjan 21 1300.60
Add another record (Y/N) Y
Enter name, age and basic salary: Harish 28 1400.70
Add another record (Y/N) N
```

Most of this program is similar to the one that we wrote earlier, which used **fprintf( )** instead of **fwrite( )**. Note, however, that the file "EMP.DAT" has now been opened in binary mode.

The information obtained from the keyboard about the employee is placed in the structure variable e. Then, the following statement writes the structure to the file:

```
fwrite ( &e, sizeof ( e ), 1, fp ) ;
```

Here, the first argument is the address of the structure to be written to the disk.

The second argument is the size of the structure in bytes. Instead of counting the bytes occupied by the structure ourselves, we let the program do it for us by using the **sizeof( )** operator. The **sizeof( )** operator gives the size of the variable in bytes. This keeps the program unchanged in event of change in the elements of the structure.

The third argument is the number of such structures that we want to write at one time. In this case, we want to write only one structure at a time. Had we had an array of structures, for example, we might have wanted to write the entire array at once.

The last argument is the pointer to the file we want to write to.

Now, let us write a program to read back the records written to the disk by the previous program.

```
/* Reads records from binary file and displays them on VDU */
#include "stdio.h"
main( )
{
    FILE *fp ;
    struct emp
    {
        char name[40] ;
        int age ;
        float bs ;
    } ;
    struct emp e ;

    fp = fopen ( "EMP.DAT", "rb" ) ;

    if ( fp == NULL )
    {
        puts ( "Cannot open file" ) ;
        exit( ) ;
    }

    while ( fread ( &e, sizeof ( e ), 1, fp ) == 1 )
        printf ( "\n%s %d %f", e.name, e.age, e.bs ) ;

    fclose ( fp ) ;
}
```

Here, the **fread( )** function causes the data read from the disk to be placed in the structure variable **e**. The format of **fread( )** is same as that of **fwrite( )**. The function **fread( )** returns the number of records read. Ordinarily, this should correspond to the third argument, the number of records we asked for... 1 in this case. If we have reached the end of file, since **fread( )** cannot read anything, it returns a 0. By testing for this situation, we know when to stop reading.

As you can now appreciate, any database management application in C must make use of **fread( )** and **fwrite( )** functions, since they store numbers more efficiently, and make writing/reading of structures quite easy. Note that even if the number of elements belonging to the structure increases, the format of **fread( )** and **fwrite( )** remains same.

Database Management

So far we have learnt record I/O in bits and pieces. However, in any serious database management application, we will have to combine all that we have learnt in a proper manner to make sense. I have attempted to do this in the following menu driven program. There is a provision to Add, Modify, List and Delete records, the operations that are imperative in any database management. Following comments would help you in understanding the program easily:

- Addition of records must always take place at the end of existing records in the file, much in the same way you would add new records in a register manually.
- Listing records means displaying the existing records on the screen. Naturally, records should be listed from first record to last record.
- While modifying records, first we must ask the user which record he intends to modify. Instead of asking the record

number to be modified, it would be more meaningful to ask for the name of the employee whose record is to be modified. On modifying the record, the existing record gets overwritten by the new record.

- In deleting records, except for the record to be deleted, rest of the records must first be written to a temporary file, then the original file must be deleted, and the temporary file must be renamed back to original.
- Observe carefully the way the file has been opened, first for reading & writing, and if this fails (the first time you run this program it would certainly fail, because that time the file is not existing), for writing and reading. It is imperative that the file should be opened in binary mode.
- Note that the file is being opened only once and closed only once, which is quite logical.
- **clrscr( )** function clears the contents of the screen and **gotoxy( )** places the cursor at appropriate position on the screen. The parameters passed to **gotoxy( )** are column number followed by row number.

Given below is the complete listing of the program.

```
/* A menu-driven program for elementary database management */
#include "stdio.h"
main( )
{
    FILE *fp, *ft ;
    char another, choice ;
    struct emp
    {
        char name[40] ;
        int age ;
        float bs ;
    } ;
```

```
struct emp  e ;
char  empname[40] ;
long int  recsize ;

fp = fopen ( "EMP.DAT", "rb+" ) ;

if ( fp == NULL )
{
     fp = fopen ( "EMP.DAT", "wb+" ) ;

     if ( fp == NULL )
     {
          puts ( "Cannot open file" ) ;
          exit( ) ;
     }
}

recsize = sizeof ( e ) ;

while ( 1 )
{
     clrscr( ) ;

     gotoxy ( 30, 10 ) ;
     printf ( "1. Add Records" ) ;
     gotoxy ( 30, 12 ) ;
     printf ( "2. List Records" ) ;
     gotoxy ( 30, 14 ) ;
     printf ( "3. Modify Records" ) ;
     gotoxy ( 30, 16 ) ;
     printf ( "4. Delete Records" ) ;
     gotoxy ( 30, 18 ) ;
     printf ( "0. Exit" ) ;
     gotoxy ( 30, 20 ) ;
     printf ( "Your choice" ) ;

     fflush ( stdin ) ;
     choice = getche( ) ;
```

```
switch ( choice ),
{
    case '1' :

        fseek ( fp, 0 , SEEK_END ) ;
        another = 'Y' ;

        while ( another == 'Y' )
        {
            printf ( "\nEnter name, age and basic sal. " ) ;
            scanf ( "%s %d %f", e.name, &e.age, &e.bs ) ;
            fwrite ( &e, recsize, 1, fp ) ;
            printf ( "\nAdd another Record (Y/N) " ) ;
            fflush ( stdin ) ;
            another = getche( ) ;
        }

        break ;

    case '2' :

        rewind ( fp ) ;

        while ( fread ( &e, recsize, 1, fp ) == 1 )
            printf ( "\n%s %d %f", e.name, e.age, e.bs ) ;

        break ;

    case '3' :

        another = 'Y' ;
        while ( another == 'Y' )
        {
            printf ( "\nEnter name of employee to modify " ) ;
            scanf ( "%s", empname ) ;

            rewind ( fp ) ;
            while ( fread ( &e, recsize, 1, fp ) == 1 )
```

```
            {
                if ( strcmp ( e.name, empname ) == 0 )
                {
                    printf ( "\nEnter new name, age & bs" ) ;
                    scanf ( "%s %d %f", e.name, &e.age,
                                &e.bs ) ;
                    fseek ( fp, - recsize, SEEK_CUR ) ;
                    fwrite ( &e, recsize, 1, fp ) ;
                    break ;
                }
            }

            printf ( "\nModify another Record (Y/N) " ) ;
            fflush ( stdin ) ;
            another = getche( ) ;
        }

        break ;

    case '4' :

        another = 'Y' ;
        while ( another == 'Y' )
        {
            printf ( "\nEnter name of employee to delete " ) ;
            scanf ( "%s", empname ) ;

            ft = fopen ( "TEMP.DAT", "wb" ) ;

            rewind ( fp ) ;
            while ( fread ( &e, recsize, 1, fp ) == 1 )
            {
                if ( strcmp ( e.name, empname ) != 0 )
                    fwrite ( &e, recsize, 1, ft ) ;
            }

            fclose ( fp ) ;
            fclose ( ft ) ;
```

```
                remove ("EMP.DAT") ;
                rename ("TEMP.DAT EMP. DAT");

                fp = fopen ("EMP.DAT," rb+") ;

                printf ("Delete another Record (Y/N)") ;
                fflush (stdin),
                another = getche ( ),

            }
            break ;
        case '0' :
            fclose (fp),
            exit ( ) ;
    }
  }
}
```

To understand how this program works, you need to be familiar with the concept of pointers.A pointer is initiated whenever we open a file. On opening a file a pointer is set up which points to the first record in the file. To be precise this pointer is present in the structure to which the file pointer returned by fopen ( ) points to. On using the functions fread ( ) or fwrite ( ) the pointer moves to the beginning of the next record. On closing a file the pointer is deactivated. Note that the pointer movement is of utmost importance since fread ( ) alwas reads that record where the pointer is currently placed. Similarly, fwrite ( ) always writes the record where the pointer is currently placed

The rewind ( ) function places the pointer to the beginning of the file, irrespective of where it is present right now.

The fseek ( ) function lets us move the pointer from one record to another. In the program above, to move the pointer to the previous record from its current position, we used the function,

fseek (fp,-recsize SEEK CUR) ;

Here, **recsize** moves the pointer back by **recsize** bytes from the current position. **SEEK CUR** is a macro defined in "stdio.h".

Similarly, the following **fseek( )** would place the pomter beyond the last record in the file.

fseek (fp. 0, SEEK_ END) ;

In fact **recsize** or **0** are just the offsets that tell the compiler by how many bytes should the pointer be moved from a particular position, The third argument could be **SEEK_END, SEEK_CUR** or **SEEK SET** All these act as a reference from which the pointer should be offset. **SEEK_END** means move the pointer from the end of the file, **SEEK_CUR** means move the pointer with reference to its current position and **SEEK_SET** means move the pointer with reference to the beginning of the file.

If we wish to know where the pointer is positioned right now, we can use the function ftell( ). It returns this position as a long int which is an offset from the beginning of the file. The value returned by ftell( ) can be used in subsequent calls to fseek( ). A sample call to ftell( ) is shown below.

position=ftell (fp)

where position is a long int

Low Level Disk I/O

In low level disk I/O, data cannot be written as individual characters, or as strings or as formatted data. There is only one way date can be written or read in low level disk I/O fanctions as a buffer full of bytes.

Writing a buffer full of data resembles the **fwrite( )** function. However, unlike **fwrite(.)**, the programmer must set up the buffer for the data, place the appropriate values in it before writing, and take them out after writing. Thus, the buffer in the low level I/O functions is very much a part of the program, rather than being invisible as in high level disk I/O functions.

Low level disk I/O functions offer following advantages:

(a) Since these functions parallel the methods that the OS uses to write to the disk, they are more efficient than the high level disk I/O functions.

(b) Since there are fewer layers of routines to go through, low level I/O functions operate faster than their high level counterparts.

Let us now write a program that uses low level disk input/output functions.

A Low Level File-copy Program

Earlier we had written a program to copy the contents of one file to another. In that program we had read the file character by character using **fgetc( )**. Each character that was read was written into the target file using **fputc( )**. Instead of performing the I/O on a character by character basis we can read a chunk of bytes from the source file and then write this chunk into the target file. While doing so the chunk would be read into the buffer and would be written to the file from the buffer. While doing so we would manage the buffer ourselves, rather than relying on the library functions to do so. This is what is low-level about this program. Here is a program which shows how this can be done.

```
/* File-copy program which copies text, .com and .exe files */
#include "fcntl.h"
#include "types.h"  /* if present in sys directory use
```

```
                    "c:tc\\include\\sys\\types.h" */
#include "stat.h"  /* if present in sys directory use
                    "c:\\tc\\include\\sys\\stat.h" */

main ( int argc, char *argv[ ] )
{
    char buffer[ 512 ], source [ 128 ], target [ 128 ] ;
    int inhandle, outhandle, bytes ;

    printf ( "\nEnter source file name" ) ;
    gets ( source ) ;

    inhandle = open ( source, O_RDONLY | O_BINARY ) ;
    if ( inhandle == -1 )
    {
        puts ( "Cannot open file" ) ;
        exit( ) ;
    }

    printf ( "\nEnter target file name" ) ;
    gets ( target ) ;

    outhandle = open ( target, O_CREAT | O_BINARY | O_WRONLY,
                    S_IWRITE ) ;
    if ( inhandle == -1 )
    {
        puts ( "Cannot open file" ) ;
        close ( inhandle ) ;
        exit( ) ;
    }

    while ( 1 )
    {
        bytes = read ( inhandle, buffer, 512 ) ;

        if ( bytes > 0 )
            write ( outhandle, buffer, bytes ) ;
        else
```

```
        break ;
    }
    close (inhandle) ;
    close (outhandle) :
}
```

Declaring the Buffer

The first difference that you will notice in this program is that we declare a character buffer,

char buffer[512] ;

This is the buffer in which the data read from the disk will be placed. The size of this buffer is important for effrcient operation. Depending on the operating system, buffers of certain sizes are handled more efficiently than others.

Opening a File

We have opened two files in our program, one is the source file from which we read the information, and the other is the target file into which we write the information read from the source file.

As in high level disk I/O, the file must be opened before we can access it. This is done using the statement,

inhandle=open (source, O_RDONLY | O_BINARY) ;

We open the file for the same reason as we did earlier-to establish communication with operating system about the file. As usual, we have to supply to **open( )**, the filename and the mode in which we want to open the file. The possible file opening modes are given below :

O_APPEND - Opens a file for appending

O_CREAT - Creates a new file for writing (has no effect if file already exists)
O_RDONLY - Creates a new file for reading only
O_ RDWR - Creates a file for both reading and writing
O_WRONLY - Creates a file for writing only
O_BINARY - Creates a file in binary mode
O_ TEXT - Creates a file in text mode

These 'O-flags' are defined in the file "fcntl.h". So this file must be included in the program while usng low level disk I/O. Note that the file "stdio.h" is not necessary for low level disk I/O. When two or more O-flags are used together, they are combined using the bitwise OR operator (|). Chapter 14 discusses bitwise operators in detail.

The other statement used in our program to open the file is,

```
outhandle = open (target, O_CREAT | O_ BINARY | O_WRONLY,
                  S-IWRITE),
```

Note that since the target file is not existing when it is being opened we have used the O_CREAT flag, and since we want to write to the file and not read from it, there fore we have used O_WRONLY. And finally, since we want to open the file in binary mode we have used O_ BINARY.

Whenever O_ CREAT flag is used, another argument must be added to **open( )** function to indicate the read/write status of the file to be created. This argument is called `permission argument'. Permission arguments could be any of the following :

S_IWRITE - Writing to the file permitted
S_IREAD - Reading from the file permitted

To use these permissions, both the files "types.h" and "stat.h" must be **#include**d in the program alongwith "fcntl.h".

File Handles

Instead of returning a FILE pointer as **fopen( )** did, in low level disk I/O, **open( )** returns an integer value called 'file handle'. This is a number assigned to a particular file, which is used thereafter to refer to the file. If **open( )** returns a value of -1, it means that the file couldn't be successfully opened.

Interaction between Buffer and File

The following statement reads the file or as much of it as will fit into the buffer:

```
bytes = read ( inhandle, buffer, 512 ) ;
```

The **read( )** function takes three arguments. The first argument is the file handle, the second is the address of the buffer and the third is the maximum number of bytes we want to read.

The **read( )** function returns the number of bytes actually read. This is an important number, since it may very well be less than the buffer size (512 bytes), and we will need to know just how full the buffer is before we can do anything with its contents. In our program we have assigned this number to the variable **bytes**.

For copying the file, we must use both the **read( )** and the **write( )** functions in a **while** loop. The **read( )** function returns the number of bytes actually read. This is assigned to the variable **bytes**. This value will be equal to the buffer size (512 bytes) until the end of file, when the buffer will only be partially full. The variable **bytes** therefore is used to tell **write( )**, as to how many bytes to write from the buffer to the target file.

Note that when large buffers are used they must be made global variables otherwise stack overflow occurs.

I/O Under Windows

As said earlier I/O in C is carried out using functions present in the library that comes with the C compiler targeted for a specific OS. Windows permits several applications to use the same screen simultaneously. Hence there is a possibility that what is written by one application to the console may get overwritten by the output sent by another application to the console. To avoid such situations Windows has completely abandoned console I/O functions. It uses a separate mechanism to send output to a window representing an application. The details of this mechanism are discussed in Chapter 17.

Though under Windows console I/O functions are not used, still functions like **fprintf( )**, **fscanf( )**, **fread( )**, **fwrite( )**, **sprintf( )**, **sscanf( )** work exactly same under Windows as well.

Summary

(a) File I/O can be performed on a character by character basis, a line by line basis, a record by record basis or a chunk by chunk basis.

(b) Different operations that can be performed on a file are—creation of a new file, opening an existing file, reading from a file, writing to a file, moving to a specific location in a file (seeking) and closing a file.

(c) File I/O is done using a buffer to improve the efficiency.

(d) A file can be a text file or a binary file depending upon its contents.

(e) Library functions convert **\n** to **\r\n** or vice versa while writing/reading to/from a file.

(f) Many library functions convert a number to a numeric string before writing it to a file, thereby using more space on disk. This can be avoided using functions **fread( )** and **fwrite( )**.
(g) In low level file I/O we can do the buffer management ourselves.

Exercise

(A) Point out the errors, if any, in the following programs :

(a)
```
# include "stdio.h"
main ( )
{
FILE *fp ;
openfile ("Myfile txt", fp) ;
if (fp ==NULL)
printf ("Unable to open file...") ;
}
openfile ( char *fn, FILE **f)
{
*f= fopen (fn, "r") ;
}
```

(b)
```
# include "stdio.h"
main( )
{
FILE *fp,
Char c;
fp=fopen ("TRY.C", "r") ;
if (fp == null)
{
puts ("Cannot open file")
exit( ) ;
}
while (( c = getc (fp)) != EOF)
putch (c) ;
fclose ( fp) ;
```

```
}
(c) main ( )
{
        char fname[ ] = "c:\\students.dat";
        FILE *fp ;
        fp = fopen (fname, "tr");
        if (fp = = NULL)
        printf ("\nUnable to open file...") ;
}
(d) main( )
{
        FILE *fp ;
        char str[80] ;
        fp = fopen ("TRY.C", "r") ;
        while (fgets (str, 80, fp) ! = EOF)
        fputs (str) ;
        fclose (fp) ;
}
(e) # include "stdio.h"
{
        unsigned char ;
        FILE *fp ;

        fp = fopen ("trial", "r") ;
        while ((ch = getc (fp)) ! = EOF)
        printf ("%c",ch) ;
        fclose (fp) ;
}
(f) main ( )
{
        FILE *fp ;
        char name [25] ;
        int age ;

        fp = fopen (" YOURS", "r") ;
```

```
    while ( fscanf ( fp, "%s %d", name, &age ) != NULL )
    fclose ( fp ) ;
}
```

(g)
```
main( )
{
    FILE *fp ;
    char names[20] ;
    int i ;
    fp = fopen ( "students.c", "wb" ) ;
    for ( i = 0 ; i <= 10 ; i++ )
    {
        puts ( "\nEnter name " ) ;
        gets ( name ) ;
        fwrite ( name, size of ( name ), 1, fp ) ;
    }
    close ( fp ) ;
}
```

(h)
```
main( )
{
    FILE *fp ;
    char name[20] = "Ajay" ;
    int i ;
    fp = fopen ( "students.c", "r" ) ;
    for ( i = 0 ; i <= 10 ; i++ )
        fwrite ( name, sizeof ( name ), 1, fp ) ;
    close ( fp ) ;
}
```

(i)
```
#include "fcntl.h"
main( )
{
    int fp ;
    fp = open ( "pr22.c" , "r" ) ;
    if ( fp == -1 )
        puts ( "cannot open file" ) ;
    else
        close ( fp ) ;
```

```
}
```

(j)
```
main( )
{
    int fp ;
    fp = fopen ( "students.c", READ | BINARY ) ;
    if ( fp == -1 )
        puts ( "cannot open file" ) ;
    else
        close ( fp ) ;
}
```

[B] Answer the following:

(a) The macro FILE is defined in which of the following files:

1. stdlib.h
2. stdio.c
3. io.h
4. stdio.h

(b) If a file contains the line "I am a boy\r\n" then on reading this line into the array **str[ ]** using **fgets( )** what would **str[ ]** contain?

1. I am a boy\r\n\0
2. I am a boy\r\0
3. I am a boy\n\0
4. I am a boy

(c) State True or False:

1. The disadvantage of High Level Disk I/O functions is that the programmer has to manage the buffers.
2. If a file is opened for reading it is necessary that the file must exist.
3. If a file opened for writing already exists its contents would be overwritten.

4. For opening a file in append mode it is necessary that the file should exist.

(d) On opening a file for reading which of the followingactivities are performed :

1. The disk is searched for existence of the file.
2. The file is brought into memory.
3. A pointer is set up which points to the first character in the file.
4. All the above.

(e) Is it necessary that a file created in text mode must always be opened in text mode for subsequent operations ?

(f) State True or False :

A file opened in binary mode and read using fgetc ( ) would report the same number of characters in the file as reported by DOS's DIR command.

(g) While using the statement,
fp= fopen ("myfile c") "r")
what happens if,

- 'myfile.c' does not exist on the disk
- 'myfile.c' exists on the disk

(h) What is the purpose of the library function fflush ( )?

(i) While using the statement,

fp = fopen (" myfile.c" "wb")
what happens if,

- 'myfile.c' does not exist on the disk
- 'myfile.c' exists on the disk

(i) A floating-Point array contains percentage marks obtained by students in an examination, To store these marks in a file 'marks.c', in which mode would you open the file and why ?

[C] Attempt the following :
(a) Write a program to read a file and display contents with its line numbers.

(b) write a program to find the size of a text file without traversing it character by character.
(c) Write a program to add the contents of one file at the end of another.

(d) Suppose a file contains student's records with each record containing name and age of a student. Write a program to read these records and display them in sorted order by name.
(e) Write a program to copy one file to another. While doing so replace all lowercase characters to their equivalent uppercase characters.

(f) Write a program that merges Lines alternately from two files and writes the results to new file If one file has less number of lines than the other, the remaining lines from the larger file should be simply copied into the target file.

(g) Write a program to display the contents of a text file on the screen. Make following provisions :

Display the contents inside a box drawn with opposite corner co-ordinates being (0, 1) and (79,23) Disply the name of the file whose contents are being displyed, and the page numbers in the zeroth now. The moment one screenful of file has been displayed, flash a message 'Press any key...' in 24th row. When a key is hit, the next page's contents should be displayed, and so on till the end of file.

(h) Write a program to encrypt/decrypt a file using.

(1) An offset cipher: In an offset cipher each character from the source file is offset with a fixed value and then written to the target file.

For example, if character read from the source file is 'A', then convert this into a new character by offsetting 'A' by a fixed value, say 128, and then writing the new character to the target file.

(2) A substitution cipher: In this each character read from the source file is substituted by a corresponding predetermined character and this character is written to the target file.

For example, if character 'A' is read from the source file, and if we have decided that every 'A' is to be substituted by '!', then a '!' would be written to the target file in place of every 'A' Similarly, every 'B' would be substituted by '5' and so on.

(i) In the file 'CUSTOMER.DAT' there are 100 records with the following structure:

```
struct customer
{
    int accno ;
    char name[30] ;
    float balance ;
} ;
```

In another file 'TRANSACTIONS.DAT' there are several records with the following structure:

```
struct trans
{
    int accno ,
    char trans_type ;
```

```
    float amount ;
};
```

The parameter **trans_type** contains D/W indicating deposit or withdrawal of amount. Write a program to update 'CUSTOMER.DAT' file, i.e. if the **trans_type** is 'D' then update the **balance** of 'CUSTOMER.DAT' by adding **amount** to balance for the corresponding **accno**. Similarly, if **trans_type** is 'W' then subtract the **amount** from **balance**. However, while subtracting the amount make sure that the amount should not get overdrawn, i.e. at least 100 Rs. Should remain in the account.

(j) There are 100 records present in a file with the following structure:

```
struct date
{
    int d, m, y ;
};

struct employee
{
    int empcode[6] ;
    char empname[20] ;
    struct date join_date ;
    float salary ;
};
```

Write a program to read these records, arrange them in ascending order of **join_date** and write them in to a target file.

(k) A hospital keeps a file of blood donors in which each record has the format:
Name: 20 Columns
Address: 40 Columns

Age: 2 Columns
Blood Type: 1 Column (Type 1, 2, 3 or 4)

Write a program to read the file and print a list of all blood donors whose age is below 25 and blood is type 2.

(l) Given a list of names of students in a class, write a program to store the names in a file on disk. Make a provision to display the n^{th} name in the list (**n** is data to be read) and to display all names starting with S.

(m) Assume that a Master file contains two fields, Roll no. and name of the student. At the end of the year, a set of students join the class and another set leaves. A Transaction file contains the roll numbers and an appropriate code to add or delete a student.

Write a program to create another file that contains the updated list of names and roll numbers. Assume that the Master file and the Transaction file are arranged in ascending order by roll numbers. The updated file should also be in ascending order by roll numbers.

(n) In a small firm employee numbers are given in serial numerical order, that is 1, 2, 3, etc.

- Create a file of employee data with following information: employee number, name, sex, gross salary.
- If more employees join, append their data to the file.
- If an employee with serial number 25 (say) leaves, delete the record by making gross salary 0.
- If some employee's gross salary increases, retrieve the record and update the salary.

Write a program to implement the above operations.

(o) Given a text file, write a program to create another text file deleting the words "a", "the", "an" and replacing each one of them with a blank space.

(p) You are given a data file EMPLOYEE.DAT with the following record structure:

```
struct employee {
                    int empno ;
                    char name[30] ;
                    int basic, grade ;
                } ;
```

Every employee has a unique **empno** and there are supposed to be no gaps between employee numbers. Records are entered into the data file in ascending order of employee number, **empno**. It is intended to check whether there are missing employee numbers. Write a program segment to read the data file records sequentially and display the list of missing employee numbers.

(q) Write a program to carry out the following:

- To read a text file "TRIAL.TXT" consisting of a maximum of 50 lines of text, each line with a maximum of 80 characters.
- Count and display the number of words contained in the file.
- Display the total number of four letter words in the text file.

Assume that the end of a word may be a space, comma or a full-stop followed by one or more spaces or a newline character.

(r) Write a program to read a list of words, sort the words in alphabetical order and display them one word per line. Also give the total number of words in the list. Output format should be:

Total Number of words in the list is _____
Alphabetical listing of words is:

Assume the end of the list is indicated by **ZZZZZZ** and there are maximum in 25 words in the Text file.

(s) Write a program to carry out the following:

(a) Read a text file 'INPUT.TXT'
(b) Print each word in reverse order

Example,

Input: INDIA IS MY COUNTRY
Output: AIDNI SI YM YRTNUOC

Assume that each word length is maximum of 10 characters and each word is separated by newline/blank characters.

(t) Write a C program to read a large text file 'NOTES.TXT' and print it on the printer in cut-sheets, introducing page breaks at the end of every 50 lines and a pause message on the screen at the end of every page for the user to change the paper.

13 More Issues In Input/Output

- Using *argc* and *argv*
- Detecting Errors in Reading/Writing
- Standard I/O Devices
- I/O Redirection
  - Redirecting the Output
  - Redirecting the Input
  - Both Ways at Once
- Summary
- Exercise

In Chapters 11 and 12 we saw how Console I/O and File I/O are done in C. There are still some more issues related with input/output that remain to be understood. These issues help in making the I/O operations more elegant.

Using *argc* and *argv*

To execute the file-copy programs that we saw in Chapter 12 we are required to first type the program, compile it, and then execute it. This program can be improved in two ways:

(a) There should be no need to compile the program every time to use the file-copy utility. It means the program must be executable at command prompt (A> or C> if you are using MS-DOS, Start | Run dialog if you are using Windows and $ prompt if you are using Unix).

(b) Instead of the program prompting us to enter the source and target filenames, we must be able to supply them at command prompt, in the form:

```
filecopy PR1.C PR2.C
```

where, PR1.C is the source filename and PR2.C is the target filename.

The first improvement is simple. In MS-DOS, the executable file (the one which can be executed at command prompt and has an extension .EXE) can be created in Turbo C/C++ by using the key F9 to compile the program. In VC++ compiler under Windows same can be done by using F7 to compile the program. Under Unix this is not required since in Unix every time we compile a program we always get an executable file.

The second improvement is possible by passing the source filename and target filename to the function **main( )**. This is illustrated below:

```
#include "stdio.h"
main ( int  argc, char  *argv[ ] )
{
    FILE  *fs, *ft ;
    char  ch ;

    if ( argc != 3 )
    {
        puts ( "Improper number of arguments" ) ;
        exit( ) ;
    }

    fs = fopen ( argv[1], "r" ) ;
    if ( fs == NULL )
    {
        puts ( "Cannot open source file" ) ;
        exit( ) ;
    }

    ft = fopen ( argv[2], "w" ) ;
    if ( ft == NULL )
    {
        puts ( "Cannot open target file" ) ;
        fclose ( fs ) ;
        exit( ) ;
    }

    while ( 1 )
    {
        ch = fgetc ( fs ) ;

        if ( ch == EOF )
            break ;
        else
            fputc ( ch, ft ) ;
    }
```

```
    fclose ( fs ) ;
    fclose ( ft ) ;
}
```

The arguments that we pass on to **main( )** at the command prompt are called command line arguments. The function **main( )** can have two arguments, traditionally named as **argc** and **argv**. Out of these, **argv** is an array of pointers to strings and **argc** is an **int** whose value is equal to the number of strings to which **argv** points. When the program is executed, the strings on the command line are passed to **main( )**. More precisely, the strings at the command line are stored in memory and address of the first string is stored in **argv[0]**, address of the second string is stored in **argv[1]** and so on. The argument **argc** is set to the number of strings given on the command line. For example, in our sample program, if at the command prompt we give,

```
filecopy PR1.C PR2.C
```

then,

argc would contain 3
argv[0] would contain base address of the string "filecopy"
argv[1] would contain base address of the string "PR1.C"
argv[2] would contain base address of the string "PR2.C"

Whenever we pass arguments to **main( )**, it is a good habit to check whether the correct number of arguments have been passed on to **main( )** or not. In our program this has been done through,

```
if ( argc != 3 )
{
    printf ( "Improper number of arguments" ) ;
    exit( ) ;
}
```

Rest of the program is same as the earlier file-copy program. This program is better than the earlier file-copy program on two counts:

(a) There is no need to recompile the program every time we want to use this utility. It can be executed at command prompt.
(b) We are able to pass source file name and target file name to **main( )**, and utilize them in **main( )**.

One final comment... the **while** loop that we have used in our program can be written in a more compact form, as shown below:

```
while ( ( ch = fgetc ( fs ) ) != EOF )
    fputc ( ch, ft ) ;
```

This avoids the usage of an indefinite loop and a **break** statement to come out of this loop. Here, first **fgetc (fs)** gets the character from the file, assigns it to the variable **ch**, and then **ch** is compared against **EOF**. Remember that it is necessary to put the expression

```
ch = fgetc ( fs )
```

within a pair of parentheses, so that first the character read is assigned to variable **ch** and then it is compared with **EOF**.

There is one more way of writing the **while** loop. It is shown below:

```
while ( !feof ( fs ) )
{
    ch = fgetc ( fs ) ;
    fputc ( ch, ft ) ;
}
```

Here, **feof( )** is a macro which returns a 0 if end of file is not reached. Hence we use the ! operator to negate this 0 to the truth value. When the end of file is reached **feof( )** returns a non-zero

value, ! makes it 0 and since now the condition evaluates to false the **while** loop gets terminated.

Note that in each one of them the following three methods for opening a file are same, since in each one of them, essentially a base address of the string (pointer to a string) is being passed to **fopen( )**.

```
fs = fopen ( "PR1.C" , "r" ) ;
fs = fopen ( filename, "r" ) ;
fs = fopen ( argv[1] , "r" ) ;
```

Detecting Errors in Reading/Writing

Not at all times when we perform a read or write operation on a file are we successful in doing so. Naturally there must be a provision to test whether our attempt to read/write was successful or not.

The standard library function **ferror( )** reports any error that might have occurred during a read/write operation on a file. It returns a zero if the read/write is successful and a non-zero value in case of a failure. The following program illustrates the usage of **ferror( )**.

```
#include "stdio.h"
main( )
{
    FILE *fp ;
    char ch ;

    fp = fopen ( "TRIAL", "w" ) ;

    while ( !feof ( fp ) )
    {
        ch = fgetc ( fp ) ;
        if ( ferror( ) )
        {
```

```
            printf ( "Error in reading file" ) ;
            break ;
        }
        else
            printf ( "%c", ch ) ;
    }

    fclose ( fp ) ;
}
```

In this program the **fgetc( )** function would obviously fail first time around since the file has been opened for writing, whereas **fgetc( )** is attempting to read from the file. The moment the error occurs **ferror( )** returns a non-zero value and the **if** block gets executed. Instead of printing the error message using **printf( )** we can use the standard library function **perror( )** which prints the error message specified by the compiler. Thus in the above program the **perror( )** function can be used as shown below.

```
if ( ferror( ) )
{
    perror ( "TRIAL" ) ;
    break ;
}
```

Note that when the error occurs the error message that is displayed is:

```
TRIAL: Permission denied
```

This means we can precede the system error message with any message of our choice. In our program we have just displayed the filename in place of the error message.

Standard I/O Devices

To perform reading or writing operations on a file we need to use the function **fopen( )**, which sets up a file pointer to refer to this file. Most OSs also predefine pointers for three standard files. To access these pointers we need not use **fopen( )**. These standard file pointers are shown in Figure 13.1

Standard File pointer	Description
stdin	standard input device (Keyboard)
stdout	standard output device (VDU)
stderr	standard error device (VDU)

Figure 13.1

Thus the statement **ch = fgetc (stdin)** would read a character from the keyboard rather than from a file. We can use this statement without any need to use **fopen( )** or **fclose( )** function calls.

Note that under MS-DOS two more standard file pointers are available—**stdprn** and **stdaux**. They stand for standard printing device and standard auxiliary device (serial port). The following program shows how to use the standard file pointers. It reads a file from the disk and prints it on the printer.

```
/* Prints file contents on printer */
#include "stdio.h"
main( )
{
    FILE *fp ;
    char ch ;
```

```
    fp = fopen ( "poem.txt", "r" ) ;

    if ( fp == NULL )
    {
        printf ( "Cannot open file" ) ;
        exit( ) ;
    }

    while ( ( ch = fgetc ( fp ) ) != EOF )
        fputc ( ch, stdprn ) ;

    fclose ( fp ) ;
}
```

The statement **fputc (ch, stdprn)** writes a character read from the file to the printer. Note that although we opened the file on the disk we didn't open **stdprn**, the printer. Standard files and their use in redirection have been dealt with in more details in the next section.

Note that these standard file pointers have been defined in the file "stdio.h". Therefore, it is necessary to include this file in the program that uses these standard file pointers.

I/O Redirection

Most operating systems incorporate a powerful feature that allows a program to read and write files, even when such a capability has not been incorporated in the program. This is done through a process called 'redirection'.

Normally a C program receives its input from the standard input device, which is assumed to be the keyboard, and sends its output to the standard output device, which is assumed to be the VDU. In other words, the OS makes certain assumptions about where input

should come from and where output should go. Redirection permits us to change these assumptions.

For example, using redirection the output of the program that normally goes to the VDU can be sent to the disk or the printer without really making a provision for it in the program. This is often a more convenient and flexible approach than providing a separate function in the program to write to the disk or printer. Similarly, redirection can be used to read information from disk file directly into a program, instead of receiving the input from keyboard.

To use redirection facility is to execute the program from the command prompt, inserting the redirection symbols at appropriate places. Let us understand this process with the help of a program.

Redirecting the Output

Let's see how we can redirect the output of a program, from the screen to a file. We'll start by considering the simple program shown below:

```
/* File name: util.c */
#include "stdio.h"<+>
main( )
{
    char ch ;
    while ( ( ch = getc ( stdin ) ) != EOF )
        putc ( ch, stdout ) ;
}
```

On compiling this program we would get an executable file UTIL.EXE. Normally, when we execute this file, the **putc( )** function will cause whatever we type to be printed on screen, until we don't type Ctrl-Z, at which point the program will terminate, as

shown in the following sample run. The Ctrl-Z character is often called end of file character.

```
C>UTIL.EXE
perhaps I had a wicked childhood,
perhaps I had a miserable youth,
but somewhere in my wicked miserable past,
there must have been a moment of truth ^Z
C>
```

Now let's see what happens when we invoke this program from in a different way, using redirection:

```
C>UTIL.EXE > POEM.TXT
C>
```

Here we are causing the output to be redirected to the file POEM.TXT. Can we prove that this the output has indeed gone to the file POEM.TXT? Yes, by using the TYPE command as follows:

```
C>TYPE POEM.TXT
perhaps I had a wicked childhood,
perhaps I had a miserable youth,
but somewhere in my wicked miserable past,
there must have been a moment of truth
C>
```

There's the result of our typing sitting in the file. The redirection operator, '>', causes any output intended for the screen to be written to the file whose name follows the operator.

Note that the data to be redirected to a file doesn't need to be typed by a user at the keyboard; the program itself can generate it. Any output normally sent to the screen can be redirected to a disk file. As an example consider the following program for generating the ASCII table on screen:

```
/* File name: ascii.c*/
main( )
{
    int ch ;

    for ( ch = 0 ; ch <= 255 ; ch++ )
        printf ( "\n%d %c", ch, ch ) ;
}
```

When this program is compiled and then executed at command prompt using the redirection operator,

```
C>ASCII.EXE > TABLE.TXT
```

the output is written to the file. This can be a useful capability any time you want to capture the output in a file, rather than displaying it on the screen.

DOS predefines a number of filenames for its own use. One of these names in PRN, which stands for the printer. Output can be redirected to the printer by using this filename. For example, if you invoke the "ascii.exe" program this way:

```
C>ASCII.EXE > PRN
```

the ASCII table will be printed on the printer.

Redirecting the Input

We can also redirect input to a program so that, instead of reading a character from the keyboard, a program reads it from a file. Let us now see how this can be done.

To redirect the input, we need to have a file containing something to be displayed. Suppose we use a file called NEWPOEM.TXT containing the following lines:

```
Let's start at the very beginning,
A very good place to start!
```

We'll assume that using some text editor these lines have been placed in the file NEWPOEM.TXT. Now, we use the input redirection operator '<' before the file, as shown below:

```
C>UTIL.EXE < NEWPOEM.TXT
Let's start at the very beginning,
A very good place to start!
C>
```

The lines are printed on the screen with no further effort on our part. Using redirection we've made our program UTIL.C perform the work of the TYPE command.

Both Ways at Once

Redirection of input and output can be used together; the input for a program can come from a file via redirection, at the same time its output can be redirected to a file. Such a program is called a filter. The following command demonstrates this process.

```
C>UTIL < NEWPOEM.TXT > POETRY.TXT
```

In this case our program receives the redirected input from the file NEWPOEM.TXT and instead of sending the output to the screen it would redirect it to the file POETRY.TXT.

Similarly to send the contents of the file NEWPOEM.TXT to the printer we can use the following command:

```
C>UTIL < NEWPOEM.TXT > PRN
```

While using such multiple redirections don't try to send output to the same file from which you are receiving input. This is because

the output file is erased before it's written to. So by the time we manage to receive the input from a file it is already erased.

Redirection can be a powerful tool for developing utility programs to examine or alter data in files. Thus, redirection is used to establish a relationship between a program and a file. Another OS operator can be used to relate two programs directly, so that the output of one is fed directly into another, with no files involved. This is called 'piping', and is done using the operator '|', called pipe. We won't pursue this topic, but you can read about it in the OS help/manual.

Summary

(a) We can pass parameters to a program at command line using the concept of 'command line arguments'.
(b) The command line argument **argv** contains values passed to the program, whereas, **argc** contains number of arguments.
(c) We can use the standard file pointer **stdin** to take input from standard input device such as keyboard.
(d) We can use the standard file pointer **stdout** to send output to the standard output device such as a monitor.
(e) We can use the standard file pointers **stdprn** and **stdaux** to interact with printer and auxiliary devices respectively.
(f) Redirection allows a program to read from or write to files at command prompt.
(g) The operators < and > are called redirection operators.

Exercise

[A] Answer the following:

(a) How will you use the following program to

- Copy the contents of one file into another.
- Print a file on the printer.
- Create a new file and add some text to it.

– Display the contents of an existing file.

```
#include "stdio.h"
main( )
{
    char ch, str[10] ;
    while ( ( ch = getc ( stdin ) ) != -1 )
    putc ( ch, stdout ) ;
}
```

(b) State True or False:

1. We can send arguments at command line even if we define **main( )** function without parameters.
2. To use standard file pointers we don't need to open the file using **fopen( )**.
3. Using **stdaux** we can send output to the printer if printer is attached to the serial port.
4. The zeroth element of the **argv** array is always the name of the exe file.

(c) Point out the errors, if any, in the following program

```
main ( int  ac, char  ( * ) av[ ] )
{
    printf ( "\n%d", ac ) ;
    printf ( "\n%s", av[0] ) ;
}
```

[B] Attempt the following:

(a) Write a program to carry out the following:

(a) Read a text file provided at command prompt
(b) Print each word in reverse order

For example if the file contains

INDIA IS MY COUNTRY

Output should be

AIDNI SI YM YRTNÙOC

(b) Write a program using command line arguments to search for a word in a file and replace it with the specified word. The usage of the program is shown below.

C> change <old word> <new word> <filename>

(c) Write a program that can be used at command prompt as a calculating utility. The usage of the program is shown below.

C> calc <switch> <n> <m>

Where, **n** and **m** are two integer operands. **switch** can be any one of the arithmetic or comparison operators. If arithmetic operator is supplied, the output should be the result of the operation. If comparison operator is supplied then the output should be **True** or **False**.

14 Operations On Bits

- Bitwise Operators
  - One's Complement Operator
  - Right Shift Operator
  - Left Shift Operator
  - Bitwise AND Operator
  - Bitwise OR Operator
  - Bitwise XOR Operator
- The *showbits( )* Function
- Summary
- Exercise

So far we have dealt with characters, integers, floats and their variations. The smallest element in memory on which we are able to operate as yet is a byte; and we operated on it by making use of the data type **char**. However, we haven't attempted to look within these data types to see how they are constructed out of individual bits, and how these bits can be manipulated. Being able to operate on a bit level, can be very important in programming, especially when a program must interact directly with the hardware. This is because, the programming languages are byte oriented, whereas hardware tends to be bit oriented. Let us now delve inside the byte and see how it is constructed and how it can be manipulated effectively. So let us take apart the byte... bit by bit.

Bitwise Operators

One of C's powerful features is a set of bit manipulation operators. These permit the programmer to access and manipulate individual bits within a piece of data. The various Bitwise Operators available in C are shown in Figure 14.1.

Operator	Meaning
~	One's complement
>>	Right shift
<<	Left shift
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR(Exclusive OR)

Figure 14.1

These operators can operate upon **int**s and **char**s but not on **float**s and **double**s. Before moving on to the details of the operators, let

us first take a look at the bit numbering scheme in integers and characters. Bits are numbered from zero onwards, increasing from right to left as shown below:

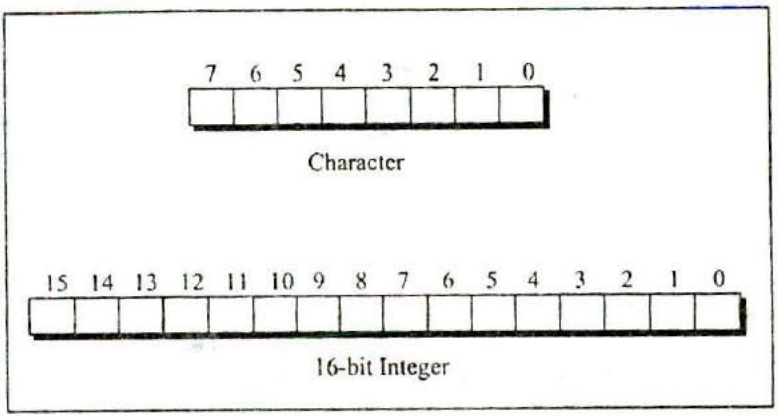


Figure 14.2

Throughout this discussion of bitwise operators we are going to use a function called **showbits( )**, but we are not going to show you the details of the function immediately. The task of **showbits( )** is to display the binary representation of any integer or character value.

We begin with a plain-jane example with **showbits( )** in action.

```
/* Print binary equivalent of integers using showbits( ) function */
main( )
{
    int j ;

    for ( j = 0 ; j <<= 5 ; j++ )
    {
        printf ( "\nDecimal %d is same as binary ", j ) ;
        showbits ( j ) ;
    }
```

```
}
```

And here is the output...

```
Decimal 0 is same as binary 0000000000000000
Decimal 1 is same as binary 0000000000000001
Decimal 2 is same as binary 0000000000000010
Decimal 3 is same as binary 0000000000000011
Decimal 4 is same as binary 0000000000000100
Decimal 5 is same as binary 0000000000000101
```

Let us now explore the various bitwise operators one by one.

One's Complement Operator

On taking one's complement of a number, all 1's present in the number are changed to 0's and all 0's are changed to 1's. For example one's complement of 1010 is 0101. Similarly, one's complement of 1111 is 0000. Note that here when we talk of a number we are talking of binary equivalent of the number. Thus, one's complement of 65 means one's complement of 0000 0000 0100 0001, which is binary equivalent of 65. One's complement of 65 therefore would be, 1111 1111 1011 1110. One's complement operator is represented by the symbol ~. Following program shows one's complement operator in action.

```
main( )
{
    int  j, k ;

    for ( j = 0 ; j <= 3 ; j++ )
    {
        printf ( "\nDecimal %d is same as binary ", j ) ;
        showbits ( j ) ;

        k = ~j ;
        printf ( "\nOne's complement of %d is ", j ) ;
```

```
        showbits ( k ) ;
    }
}
```

And here is the output of the above program...

```
Decimal 0 is same as binary 0000000000000000
One's complement of 0 is 1111111111111111
Decimal 1 is same as binary 0000000000000001
One's complement of 1 is 1111111111111110
Decimal 2 is same as binary 0000000000000010
One's complement of 2 is 1111111111111101
Decimal 3 is same as binary 0000000000000011
One's complement of 3 is 1111111111111100
```

In real-world situations where could the one's complement operator be useful? Since it changes the original number beyond recognition, one potential place where it can be effectively used is in development of a file encryption utility as shown below:

```
/* File encryption utility */
#include "stdio.h"
main( )
{
    encrypt( ) ;
}

encrypt( )
{
    FILE *fs, *ft ;
    char  ch ;

    fs = fopen ( "SOURCE.C", "r" ) ;  /* normal file */
    ft = fopen ( "TARGET.C", "w" ) ;  /* encrypted file */

    if ( fs == NULL || ft == NULL )
    {
```

```
        printf ( "\nFile opening error!" ) ;
        exit ( 1 ) ;
    }

    while ( ( ch = getc ( fs ) ) != EOF )
        putc ( ~ch, ft ) ;

    fclose ( fs ) ;
    fclose ( ft ) ;
}
```

How would you write the corresponding decrypt function? Would there be any problem in tackling the end of file marker? It may be recalled here that the end of file in text files is indicated by a character whose ASCII value is 26.

Right Shift Operator

The right shift operator is represented by >>. It needs two operands. It shifts each bit in its left operand to the right. The number of places the bits are shifted depends on the number following the operator (i.e. its right operand).

Thus, **ch >> 3** would shift all bits in **ch** three places to the right. Similarly, **ch >> 5** would shift all bits 5 places to the right.

For example, if the variable **ch** contains the bit pattern 11010111, then, **ch >> 1** would give 01101011 and **ch >> 2** would give 00110101.

Note that as the bits are shifted to the right, blanks are created on the left. These blanks must be filled somehow. They are always filled with zeros. The following program demonstrates the effect of right shift operator.

```
main( )
{
```

```
    int i = 5225, j, k ;

    printf ( "\nDecimal %d is same as binary ", i ) ;
    showbits ( i ) ;

    for ( j = 0 ; j <= 5 ; j++ )
    {
        k = i >>j ;
        printf ( "\n%d right shift %d gives ", i, j ) ;
        showbits ( k ) ;
    }
}
```

The output of the above program would be...

```
Decimal 5225 is same as binary 0001010001101001
5225 right shift 0 gives 0001010001101001
5225 right shift 1 gives 0000101000110100
5225 right shift 2 gives 0000010100011010
5225 right shift 3 gives 0000001010001101
5225 right shift 4 gives 0000000101000110
5225 right shift 5 gives 0000000010100011
```

Note that if the operand is a multiple of 2 then shifting the operand one bit to right is same as dividing it by 2 and ignoring the remainder. Thus,

```
64 >> 1 gives 32
64 >> 2 gives 16
128 >> 2 gives 32
```

but,

```
27 >> 1 is 13
49 >> 1 is 24 .
```

A Word of Caution

In the explanation **a >> b** if **b** is negative the result is unpredictable. If **a** is negative than its left most bit (sign bit) would be 1. On some computer right shifting **a** would result in extending the sign bit. For example, if **a** contains -1, its binary representation would be 1111111111111111. Without sign extension, the operation **a >> 4** would be 0000111111111111. However, on the machine on which we executed this expression the result turns out to be 1111111111111111. Thus the sign bit 1 continues to get extended.

Left Shift Operator

This is similar to the right shift operator, the only difference being that the bits are shifted to the left, and for each bit shifted, a 0 is added to the right of the number. The following program should clarify my point.

```
main( )
{
    int i = 5225, j, k ;

    printf ( "\nDecimal %d is same as ", i ) ;
    showbits ( i ) ;

    for ( j = 0 ; j <= 4 ; j++ )
    {
        k = i <<j ;
        printf ( "\n%d left shift %d gives ", i, j ) ;
        showbits ( k ) ;
    }
}
```

The output of the above program would be...

```
Decimal 5225 is same as binary 0001010001101001
```

```
5225 left shift 0 gives 0001010001101001
5225 left shift 1 gives 0010100011010010
5225 left shift 2 gives 0101000110100100
5225 left shift 3 gives 1010001101001000
5225 left shift 4 gives 0100011010010000
```

Having acquainted ourselves with the left shift and right shift operators, let us now find out the practical utility of these operators.

In DOS/Windows the date on which a file is created (or modified) is stored as a 2-byte entry in the 32 byte directory entry of that file. Similarly, a 2-byte entry is made of the time of creation or modification of the file. Remember that DOS/Windows doesn't store the date (day, month, and year) of file creation as a 8 byte string, but as a codified 2 byte entry, thereby saving 6 bytes for each file entry in the directory. The bitwise distribution of year, month and date in the 2-byte entry is shown in Figure 14.3.

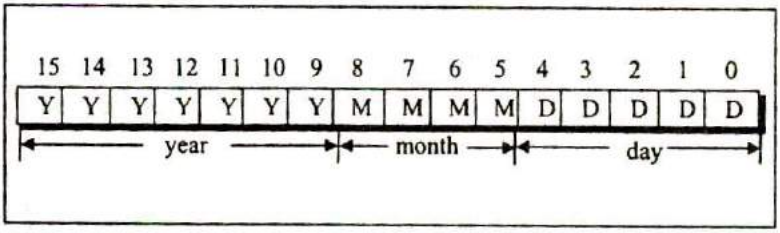


Figure 14.3

DOS/Windows converts the actual date into a 2-byte value using the following formula:

```
date = 512 * ( year - 1980 ) + 32 * month + day
```

Suppose 09/03/1990 is the date, then on conversion the date will be,

```
date = 512 * ( 1990 - 1980 ) + 32 * 3 + 9 = 5225
```

The binary equivalent of 5225 is 0001 0100 0110 1001. This binary value is placed in the date field in the directory entry of the file as shown below.

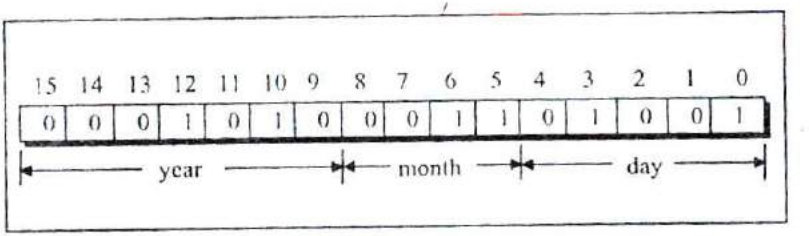


Figure 14.4

Just to verify this bit distribution, let us take the bits representing the month,

```
month = 0011
      = 1 * 2 + 1 * 1
      = 3
```

Similarly, the year and the day can also be verified.

When we issue the command DIR or use Windows Explorer to list the files, the file's date is again presented on the screen in the usual date format of mm/dd/yy. How does this integer to date conversion take place? Obviously, using left shift and right shift operators.

When we take a look at Figure 14.4 depicting the bit pattern of the 2- byte date field, we see that the year, month and day exist as a bunch of bits in contiguous locations. Separating each of them is a matter of applying the bitwise operators.

For example, to get year as a separate entity from the two bytes entry we right shift the entry by 9 to get the year. Just see, how...

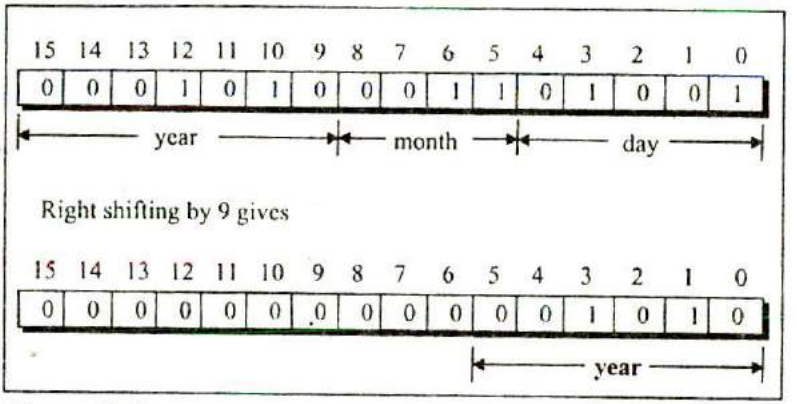


Figure 14.5

On similar lines, left shifting by 7, followed by right shifting by 12 yields month.

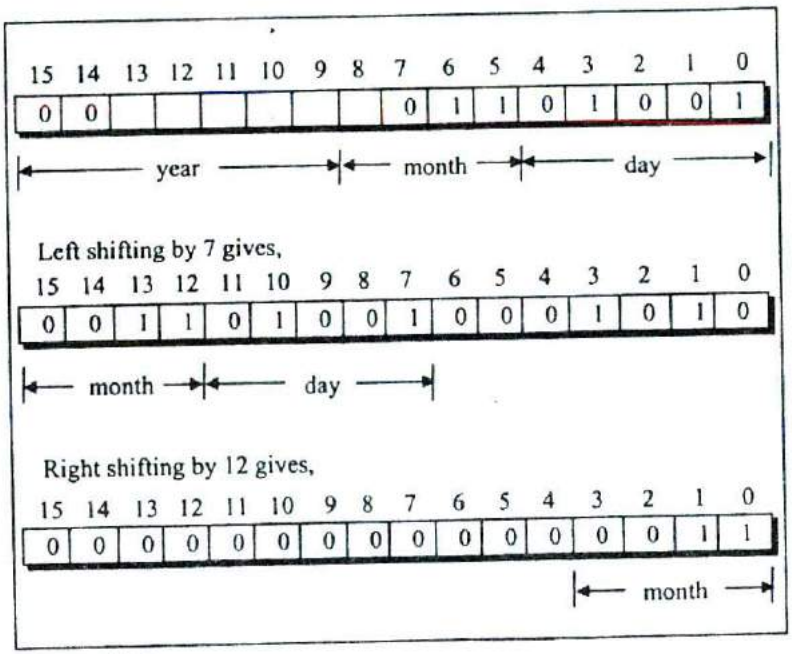


Figure 14.6

Finally, for obtaining the day, left shift date by 11 and then right shift the result by 11. Left shifting by 11 gives 0100100000000000. Right shifting by 11 gives 0000000000001001.

This entire logic can be put into a program as shown below:

```
/* Decoding date field in directory entry using bitwise operators */
main( )
{
    unsigned int  d = 9, m = 3, y = 1990, year, month, day, date ;

    date = ( y - 1980 ) * 512 + m * 32 + d ;
    printf ( "\nDate = %u", date ) ;
```

```
    year = 1980 + ( date >> 9 ) ;
    month = ( (date << 7 ) >> 12 ) ;
    day = ( (date << 11 ) >> 11 ) ;
    printf ( "\nYear = %u ", year ) ;
    printf ( "Month = %u ", month ) ;
    printf ( "Day = %u", day ) ;
}
```

And here is the output...

```
Date = 5225
Year = 1990  Month = 3  Day = 9
```

Bitwise AND Operator

This operator is represented as **&**. Remember it is different than **&&**, the logical AND operator. The **&** operator operates on two operands. While operating upon these two operands they are compared on a bit-by-bit basis. Hence both the operands must be of the same type (either **char** or **int**). The second operand is often called an AND mask. The **&** operator operates on a pair of bits to yield a resultant bit. The rules that decide the value of the resultant bit are shown below:

First bit	Second bit	First bit & Second bit
0	0	0
0	1	0
1	0	0
1	1	1

Figure 14.7

This can be represented in a more understandable form as a 'Truth Table' shown in Figure 14.8.

&	0	1
0	0	0
1	0	1

Figure 14.8

The example given below shows more clearly what happens while ANDing one operand with another. The rules given in the Figure 14.8 are applied to each pair of bits one by one.

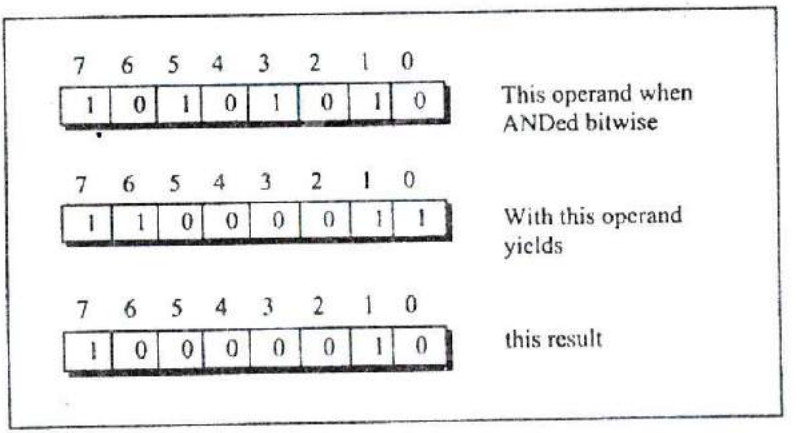


Figure 14.9

Work through the Truth Table and confirm that the result obtained is really correct.

Thus, it must be clear that the operation is being performed on individual bits, and the operation performed on one pair of bits is

completely independent of the operation performed on the other pairs.

Probably, the best use of the AND operator is to check whether a particular bit of an operand is ON or OFF. This is explained in the following example.

Suppose, from the bit pattern 10101101 of an operand, we want to check whether bit number 3 is ON (1) or OFF (0). Since we want to check the bit number 3, the second operand for the AND operation should be $1 * 2^3$, which is equal to 8. This operand can be represented bitwise as 00001000.

Then the ANDing operation would be,

```
10101101        Original bit pattern
00001000        AND mask
--------------
00001000        Resulting bit pattern
```

The resulting value we get in this case is 8, i.e the value of the second operand. The result turned out to be 8 since the third bit of the first operand was ON. Had it been OFF, the bit number 3 in the resulting bit pattern would have evaluated to 0 and the complete bit pattern would have been 00000000.

Thus, depending upon the bit number to be checked in the first operand we decide the second operand, and on ANDing these two operands the result decides whether the bit was ON or OFF. If the bit is ON (1), the resulting value turns out to be a non-zero value which is equal to the value of second operand, and if the bit is OFF (0) the result is zero as seen above. The following program puts this logic into action.

```
/* To test whether a bit in a number is ON or OFF */
main( )
{
```

```
    int i = 65, j ;

    printf ( "\nvalue of i = %d"; i ) ;
    j = i & 32 ;

    if ( j == 0 )
        printf ( "\nand its fifth bit is off" ) ;
    else
        printf ( "\nand its fifth bit is on" ) ;

    j = i & 64 ;

    if ( j == 0 )
        printf ( "\nwhereas its sixth bit is off" ) ;
    else
        printf ( "\nwhereas its sixth bit is on" ) ;
}
```

And here is the output...

```
Value of i = 65
and its fifth bit is off
whereas its sixth bit is on
```

In every file entry present in the directory, there is an attribute byte. The status of a file is governed by the value of individual bits in this attribute byte. The AND operator can be used to check the status of the bits of this attribute byte. The meaning of each bit in the attribute byte is shown in Figure 14.10.

Bit numbers								Meaning
7	6	5	4	3	2	1	0	
.	.	.	.	.	.	.	1	Read only
.	.	.	.	.	.	1	.	Hidden
.	.	.	.	.	1	.	.	System
.	.	.	.	1	.	.	.	Volume label entry
.	.	.	1	.	.	.	.	Sub-directory entry
.	.	1	.	.	.	.	.	Archive bit
.	1	.	.	.	.	.	.	Unused
1	.	.	.	.	.	.	.	Unused

Figure 14.10

Now, suppose we want to check whether a file is a hidden file or not. A hidden file is one, which is never shown in the directory, even though it exists on the disk. From the above bit classification of attribute byte, we only need to check whether bit number 1 is ON or OFF.

So, our first operand in this case becomes the attribute byte of the file in question, whereas the second operand is the $1 * 2^1 = 2$, as discussed earlier. Similarly, it can be checked whether the file is a system file or not, whether the file is read-only file or not, and so on.

The second, and equally important use of the AND operator is in changing the status of the bit, or more precisely to switch OFF a particular bit.

If the first operand happens to be 00000111, then to switch OFF bit number 1, our AND mask bit pattern should be 11111101. On applying this mask, we get,

```
00000111    Original bit pattern
11111101    AND mask
--------
00000101    Resulting bit pattern
```

Here in the AND mask we keep the value of all other bits as 1 except the one which is to be switched OFF (which is purposefully kept as 0). Therefore, irrespective of whether the first bit is ON or OFF previously, it is switched OFF. At the same time the value 1 provided in all the other bits of the AND mask (second operand) keeps the bit values of the other bits in the first operand unaltered.

Let's summarize the uses of bitwise AND operator:

(a) It is used to check whether a particular bit in a number is ON or OFF.
(b) It is used to turn OFF a particular bit in a number.

Bitwise OR Operator

Another important bitwise operator is the OR operator which is represented as |. The rules that govern the value of the resulting bit obtained after ORing of two bits is shown in the truth table below.

|	0	1
0	0	1
1	1	1

Figure 14.11

Using the Truth table confirm the result obtained on ORing the two operands as shown below.

```
11010000    Original bit pattern
00000111    OR mask
--------
11010111    Resulting bit pattern
```

Bitwise OR operator is usually used to put ON a particular bit in a number.

Let us consider the bit pattern 11000011. If we want to put ON bit number 3, then the OR mask to be used would be 00001000. Note that all the other bits in the mask are set to 0 and only the bit, which we want to set ON in the resulting value is set to 1.

Bitwise XOR Operator

The XOR operator is represented as ^ and is also called an Exclusive OR Operator. The OR operator returns 1, when any one of the two bits or both the bits are 1, whereas XOR returns 1 only if one of the two bits is 1. The truth table for the XOR operator is given below.

^	0	1
0	0	1
1	1	0

Figure 14.12

XOR operator is used to toggle a bit ON or OFF. A number XORed with another number twice gives the original number. This is shown in the following program.

```
main( )
{
    int b = 50 ;

    b = b ^ 12 ;
    printf ( "\n%d", b ) ; /* this will print 62 */

    b = b ^ 12 ;
    printf ( "\n%d", b ) ; /* this will print 50 */
}
```

The *showbits( )* Function

We have used this function quite often in this chapter. Now we have sufficient knowledge of bitwise operators and hence are in a position to understand it. The function is given below followed by a brief explanation.

```
showbits ( int n )
{
    int i, k, andmask ;

    for ( i = 15 ; i >= 0 ; i-- )
    {
        andmask = 1 << i ;
        k = n & andmask ;

        k == 0 ? printf ( "0" ) : printf ( "1" ) ;
    }
}
```

All that is being done in this function is using an AND operator and a variable **andmask** we are checking the status of individual bits. If the bit is OFF we print a 0 otherwise we print a 1.

First time through the loop, the variable **andmask** will contain the value 1000000000000000, which is obtained by left-shifting 1,

fifteen places. If the variable **n**'s most significant bit is 0, then **k** would contain a value 0, otherwise it would contain a non-zero value. If **k** contains 0 then **printf( )** will print out 0 otherwise it will print out 1.

On the second go-around of the loop, the value of **i** is decremented and hence the value of **andmask** changes, which will now be 0100000000000000. This checks whether the next most significant bit is 1 or 0, and prints it out accordingly. The same operation is repeated for all bits in the number.

Summary

(a) To help manipulate hardware oriented data—individual bits rather than bytes a set of bitwise operators are used.
(b) The bitwise operators include operators like one's complement, right-shift, left-shift, bitwise AND, OR, and XOR.
(c) The one's complement converts all zeros in its operand to 1s and all 1s to 0s.
(d) The right-shift and left-shift operators are useful in eliminating bits from a number—either from the left or from the right.
(e) The bitwise AND operators is useful in testing whether a bit is on/off and in putting off a particular bit.
(f) The bitwise OR operator is used to turn on a particular bit.
(g) The XOR operator works almost same as the OR operator except one minor variation.

Exercise

[A] Answer the following:

(a) The information about colors is to be stored in bits of a **char** variable called **color**. The bit number 0 to 6, each represent 7 colors of a rainbow, i.e. bit 0 represents violet, 1 represents

indigo, and so on. Write a program that asks the user to enter a number and based on this number it reports which colors in the rainbow does the number represents.

(b) A company planning to launch a new newspaper in market conducts a survey. The various parameters considered in the survey were, the economic status (upper, middle, and lower class) the languages readers prefer (English, Hindi, Regional language) and category of paper (daily, supplement, tabloid). Write a program, which reads data of 10 respondents through keyboard, and stores the information in an array of integers. The bit-wise information to be stored in an integer is given below:

Bit Number	Information
0	Upper class
1	Middle class
2	Lower class
3	English
4	Hindi
5	Regional Language
6	Daily
7	Supplement
8	Tabloid

At the end give the statistical data for number of persons who read English daily, number of upper class people who read tabloid and number of regional language readers.

(c) In an inter-college competition, various sports and games are played between different colleges like cricket, basketball, football, hockey, lawn tennis, table tennis, carom and chess. The information regarding the games won by a particular college is stored in bit numbers 0, 1, 2, 3, 4, 5, 6, 7 and 8 respectively of an integer variable called **game**. The college

that wins in 5 or more than 5 games is awarded the Champion of Champions trophy. If a number is entered through the keyboard, then write a program to find out whether the college won the Champion of the Champions trophy or not, along with the names of the games won by the college.

(d) An animal could be either a canine (dog, wolf, fox, etc.), a feline (cat, lynx, jaguar, etc.), a cetacean (whale, narwhal, etc.) or a marsupial (koala, wombat, etc.). The information whether a particular animal is canine, feline, cetacean, or marsupial is stored in bit number 0, 1, 2 and 3 respectively of a integer variable called **type**. Bit number 4 of the variable **type** stores the information about whether the animal is Carnivore or Herbivore.

For the following animal, complete the program to determine whether the animal is a herbivore or a carnivore. Also determine whether the animal is a canine, feline, cetacean or a marsupial.

```
struct animal
{
    char name[30] ;
    int type ;
}
struct animal a = { "OCELOT", 18 } ;
```

(e) The time field in the directory entry is 2 bytes long. Distribution of different bits which account for hours, minutes and seconds is given below. Write a function which would receive the two-byte time entry and return to the calling function, the hours, minutes and seconds.

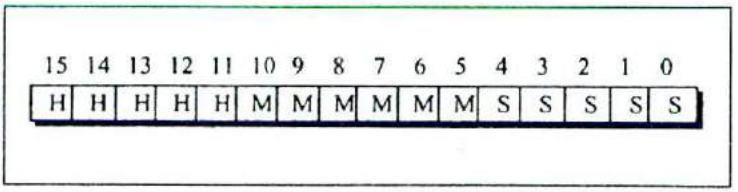


Figure 14.13

(f) In order to save disk space information about student is stored in an integer variable. If bit number 0 is on then it indicates Ist year student, bit number 1 to 3 stores IInd year, IIIrd year and IVth year student respectively. The bit number 4 to 7 stores stream Mechanical, Chemical, Electronics and IT. Rest of the bits store room number. Based on the given data, write a program that asks for the room number and displays the information about the student, if its data exists in the array. The contents of array are,

```
int data[ ] = { 273, 548, 786, 1096 } ;
```

(g) What will be the output of the following program:

```
main( )
{
    int i = 32, j = 65, k, l, m, n, o, p ;
    k = i | 35 ;  l = ~k ;  m = i & j ;
    n = j ^ 32 ;  o = j << 2 ;  p = i >> 5 ;
    printf ( "\nk = %d l = %d m = %d", k, l, m ) ;
    printf ( "\nn = %d o = %d p = %d", n, o, p ) ;
}
```