MODULE 5- Loops and Arrays

Contents: Loop Control Structure, while Loop, for Loop, do – while Loop, break statement, continue statement, Arrays, Array Initialization, 1-D and 2-D Array

Loops: The versatility of the computer lies in its ability to perform a set of instructions repeatedly. This involves repeating some portion of the program either a specified number of times or until a particular condition is being satisfied. This repetitive o through a loop control instruction. There are three methods by way of which we can repeat a part of a program. They are:

(a) Using a **for** statement
(b) Using a **while** statement
(c) Using a **do-while** statement

w***hile*** **Loop:** The general form of **while** is as shown below:

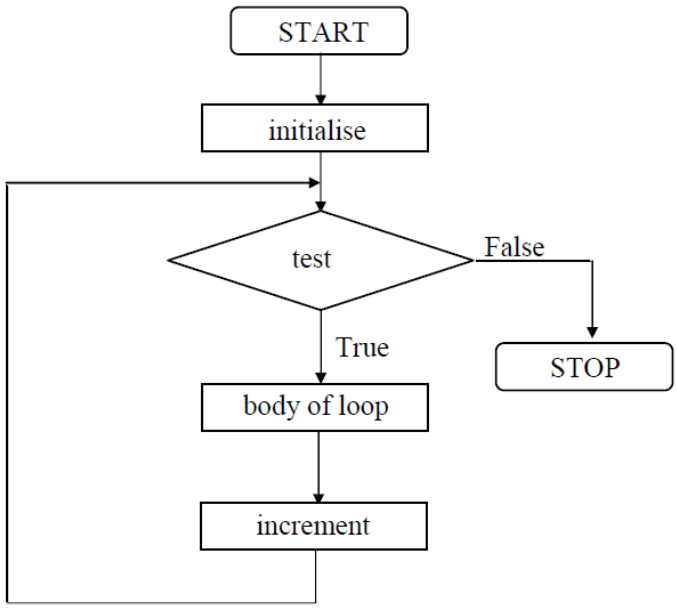
```
initialise loop counter ;
while ( test loop counter using a condition )
{
        do this ;
        and this ;
        increment loop counter ;

}
```

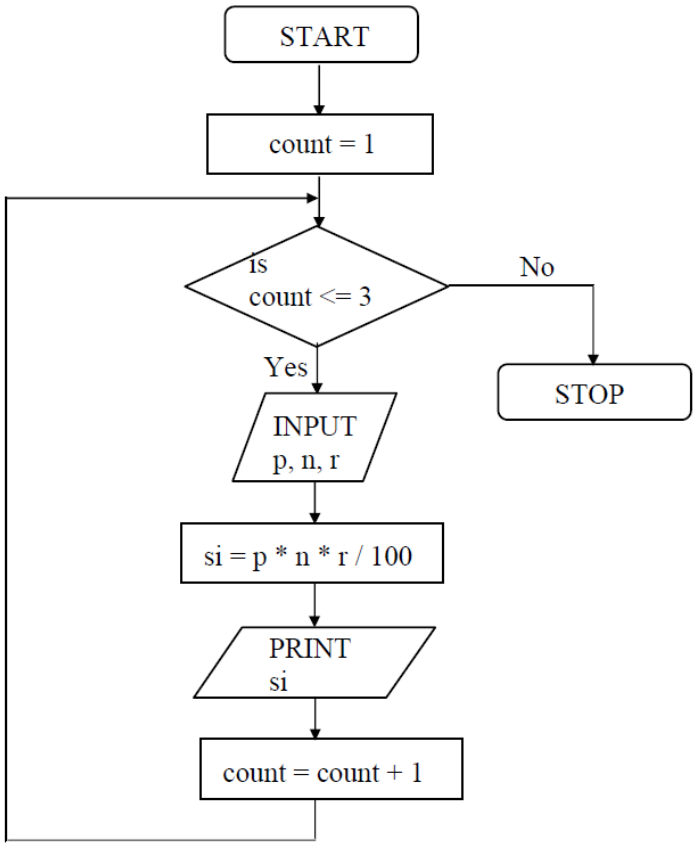
About **while...**

1. The statements within the **while** loop would keep on getting executed till the condition being tested remains true. When the

The operation of the **while** loop is illustrated in the following figure.



The flowchart shown below would help you to understand the operation of the **while** loop. Here simple interest is calculated 3 times for different values of p,n,r;




```
/* Calculation of simple interest for 3 sets of p, n and r */
main( )
{
        int p, n, count ;
        float r, si ;
        count = 1 ;
        while ( count <= 3 )
        {
                printf ( "\nEnter values of p, n and r " ) ;
                scanf("%d %d %f", &p, &n, &r ) ;
                si=p*n * r / 100 ;
                printf ( "Simple interest = Rs. %f", si ) ;
                count = count +1;
        }
}
```

The program executes all statements after while 3 times. The logic for calculating the simple interest is written within pair of braces immediately after the while keyword. The loop will execute till the condition written in parentheses after while keyword is true. When the condition becomes false, the control passes to the instruction after the while loop. As a rule the while must test a condition that will eventually become false, otherwise the loop would be executed forever, indefinitely.

```
main( )
```

```
{
        int i;
        while (i<=10)
        {
                printf (“%d”,i);
                i = i + 1;
        }
}
```

Instead of incrementing loop counter we can decrement it and still manage to get the body of the loop executed repeatedly.
Example:

```
main( )
{
        int i=10;
        while (i>=1)
        {
                printf (“This is line %d”,i);
                i = i - 1;
        }
}
```

for loop: The for allow us to specify three things about a loop in single line.

1) Setting a loop counter to an initial value.
2) Testing the loop c ounter to determine whether its value has reached the number of repetitions desired.
3) Increasing the value of loop counter each time the program segment within the loop has been executed.

The general form of for loop is:

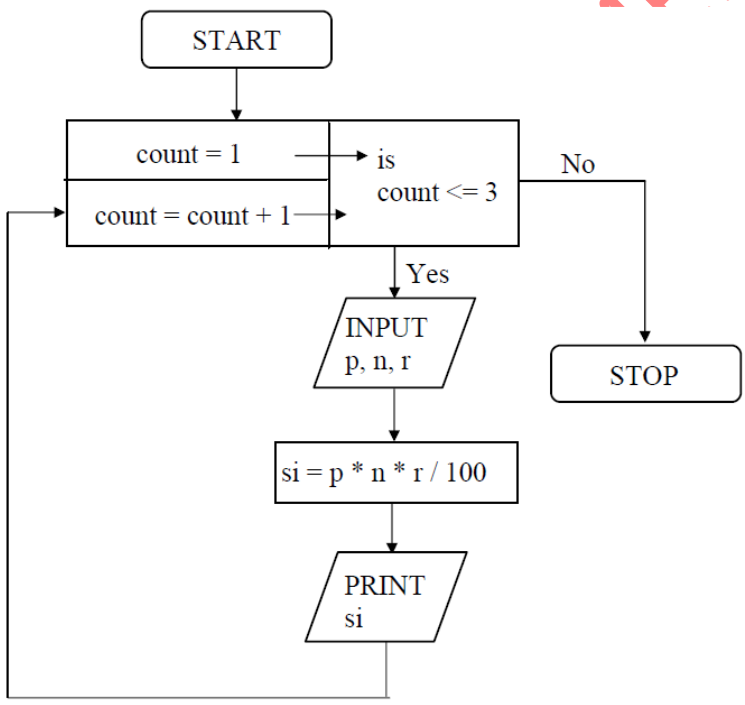
```
for ( initialise counter ; test counter ; increment counter )
{
do this ;
and this ;
and this ;
}
```

Example:

```
/* Calculation of simple interest for 3 sets of p, n and r */
main ( )
{
        int p, n, count ;
        float r, si ;
        for ( count = 1 ; count <= 3 ; count = count + 1 )
        {
                printf ( "Enter values of p, n, and r " ) ;
                scanf ( "%d %d %f", &p, &n, &r ) ;
```

```
            si = p * n * r / 100 ;
            printf ( "Simple Interest = Rs.%f\n", si ) ;
        }
}
```

Let us now examine how the **for** statement gets executed:
− When the **for** statement is executed for the first time, the value of **count** is set to an initial value 1.
− Now the condition **count <= 3** is tested. Since **count** is 1 the condition is satisfied and the body of the loop is executed for the first time.
− Upon reaching the closing brace of **for**, control is sent back to the **for** statement, where the value of **count** gets incremented by 1.
− Again the test is performed to check whether the new value of **count** exceeds 3.
− If the value of **count** is still within the range 1 to 3, the statements within the braces of **for** are executed again.
− The body of the **for** loop continues to get executed till **count** doesn't exceed the final value 3.
− When **count** reaches the value 4 the control exits from the loop and is transferred to the statement (if any) immediately after the body of **for**.
The following figure would help in further clarifying the concept of execution of the **for** loop.



It is important to note that the initialization, testing and incrementation part of a **for** loop can be replaced by any valid expression. Thus the following **for** loops are perfectly ok.

1)
```
for ( i = 10 ; i ; i -- )
    printf ( "%d", i ) ;
```
2)
```
for ( i < 4 ; j = 5 ; j = 0 )
    printf ( "%d", i ) ;
```
3)
```
for ( i = 1; i <=10 ; printf ( "%d",i++ )
        ;
```

4) for (scanf ("%d", &i) ; i <= 10 ; i++)
printf ("%d", i) ;

Example: Write down the program to print numbers from 1 to 10 in different ways.

```
main( )
{
        int i ;
        for ( i = 1 ; i <= 10 ; i = i + 1 )
        printf ( "%d\n", i ) ;
}
```

Nesting of Loops

The way **if** statements can be nested, similarly **while**s and **for**s can also be nested. To understand how nested loops work, look at the program given below:

```
/* Demonstration of nested loops */
main( )
{
        int r, c, sum ;
        for ( r = 1 ; r <= 3 ; r++ ) /* outer loop */
        {
                for ( c = 1 ; c <= 2 ; c++ ) /* inner loop */
                {
                        sum = r + c ;
                        printf ( "r = %d c = %d sum = %d\n", r, c, sum ) ;
                }
        }
}
```

When you run this program you will get the following output:

```
r = 1 c = 1 sum = 2
r = 1 c = 2 sum = 3
r = 2 c = 1 sum = 3
r = 2 c = 2 sum = 4
r = 3 c = 1 sum = 4
r = 3 c = 2 sum = 5
```

Here, for each value of **r** the inner loop is cycled through twice, with the variable **c** taking values from 1 to 2. The inner loop terminates when the value of **c** exceeds 2, and the outer loop terminates when the value of **r** exceeds 3. As you can see, the body of the outer **for** loop is indented, and the body of the inner **for** loop is further indented. These multiple indentations make the program easier to understand.

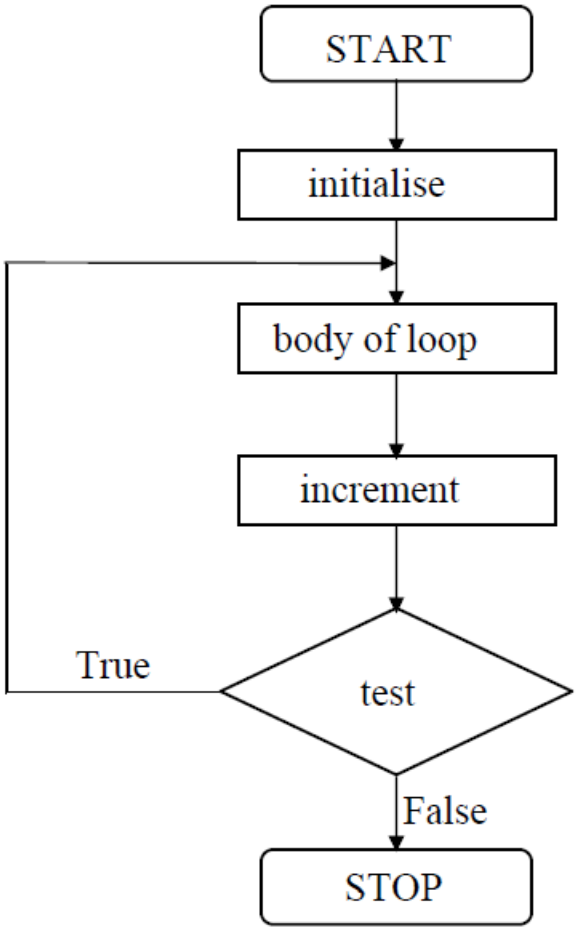
do-while loop: The **do-while** loop looks like this:

```
do
{
        this ;
        and this ;
```

and this ;
and this ;
} while (this condition is true) ;

There is a minor difference between the working of **while** and **dowhile** loops. This difference is the place where the condition is tested. The **while** tests the condition before executing any of the statements within the **while** loop. As against this, the **do-while** tests the condition after having executed the statements within the loop.

Figure would clarify the execution of **do-while** loop still further.



The loops that we have used so far executed the statements within them a finite number of times. However, in real life programming one comes across a situation when it is not known beforehand how many times the statements in the loop are to be executed. This situation can be programmed as shown below:

```
/* Execution of a loop an unknown number of times */
main( )
{
        char another ;
        int num ;
        do
        {
                printf ( "Enter a number " ) ;
                scanf ( "%d", &num ) ;
                printf ( "square of %d is %d", num, num * num ) ;
                printf ( "\nWant to enter another number y/n " ) ;
                scanf ( " %c", &another ) ;
```

```
        } while ( another == 'y' ) ;
}
```

And here is the sample output...

```
Enter a number 5
square of 5 is 25
Want to enter another number y/n y
Enter a number 7
square of 7 is 49
Want to enter another number y/n n
```

In this program the **do-while** loop would keep getting executed till the user continues to answer y. The moment he answers n, the loop terminates, since the condition (**another == 'y'**) fails. Note that this loop ensures that statements within it are executed at least once even if **n** is supplied first time itself.

The *break* Statement

We often come across situations where we want to jump out of a loop instantly, without waiting to get back to the conditional test. The keyword **break** allows us to do this. When **break** is encountered inside any loop, control automatically passes to the first statement after the loop. A **break** is usually associated with an **if**. As an example, let's consider the following example.

Example: Write a program to determine whether a number is prime or not. A prime number is one, which is divisible only by 1 or itself. All we have to do to test whether a number is prime or not, is to divide it successively by all numbers from 2 to one less than itself.

If remainder of any of these divisions is zero, the number is not a prime. If no division yields a zero then the number is a prime number. Following program implements this logic.

```
main( )
{
        int num, i ;
        printf ( "Enter a number " ) ;
        scanf ( "%d", &num ) ;
        i = 2 ;
        while ( i <= num - 1 )
        {
                if ( num % i == 0 )
                {
                        printf ( "Not a prime number" ) ;
                        break ;
                }
                i++ ;
        }
        if ( i == num )
        printf ( "Prime number" ) ;
}
```

In this program the moment **num % i** turns out to be zero, (i.e. **num** is exactly divisible by **i**) the message "Not a prime number" is printed and the control breaks out of the **while** loop. Why does the program require the **if** statement after the **while** loop at all?

Well, there are two ways the control could have reached outside the **while** loop:

- It jumped out because the number proved to be not a prime.
- The loop came to an end because the value of **i** became equal to **num**.

When the loop terminates in the second case, it means that there was no number between 2 to **num - 1** that could exactly divide **num**. That is, **num** is indeed a prime. If this is true, the program should print out the message "Prime number". The keyword **break**, breaks the control only from the **while** in which it is placed. Consider the following program, which illustrates this fact.

```
main( )
{
        int i = 1 , j = 1 ;
        while ( i++ <= 100 )
        {
                while ( j++ <= 200 )
                {
                        if ( j == 150 )
                                break ;
                        else
                                printf ( "%d %d\n", i, j ) ;
                }
        }
}
```

In this program when **j** equals 150, **break** takes the control outside the inner **while** only, since it is placed inside the inner **while**.

The *continue* Statement

In some programming situations we want to take the control to the beginning of the loop, bypassing the statements inside the loop, which have not yet been executed. The keyword **continue** allows us to do this. When **continue** is encountered inside any loop, control automatically passes to the beginning of the loop. A **continue** is usually associated with an **if**. As an example, let's consider the following program.

```
main( )
{
        int i, j ;
        for ( i = 1 ; i <= 2 ; i++ )
        {
                for ( j = 1 ; j <= 2 ; j++ )
                {
                        if ( i == j )
                                continue ;
                        printf ( "\n%d %d\n", i, j ) ;
                }
        }
}
```

The output of the above program would be...

```
1 2
2 1
```

Note that when the value of **i** equals that of **j**, the **continue** statement takes the control to the **for** loop (inner) bypassing rest of the statements pending execution in the **for** loop (inner).

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:

Construct 100 variables to store percentage marks obtained by 100 different students, i.e. each variable containing one student's marks. Construct one variable (called array or subscripted variable) capable of storing or holding all the hundred values. 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 ) ;
}
```

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:

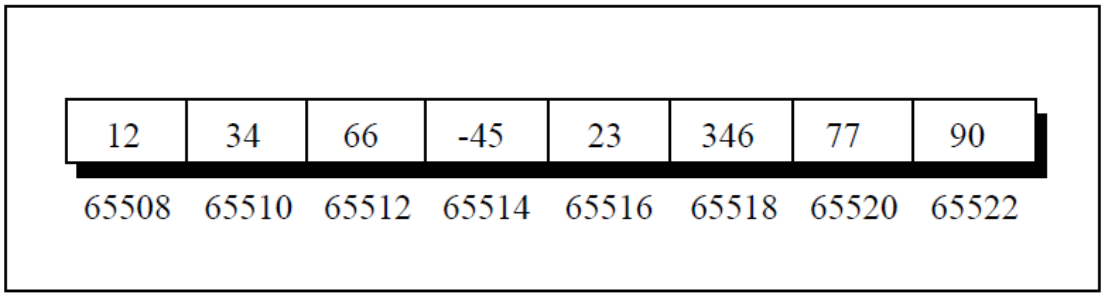
- An array is a collection of similar elements.
- The first element in the array is numbered 0, so the last element is 1 less than the size of the array.
- An array is also known as a subscripted variable.
- Before using an array its type and dimension must be declared.
- However big an array its elements are always stored in contiguous memory locations.

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. 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



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.

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

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.

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

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 above figure is only conceptually true. This is because memory doesn't contain rows and columns.

In memory whether it is a one-dimensional or a two-dimensional array the array elements are stored in one continuous chain. The arrangement of array elements of a two-dimensional array in memory is shown below:

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

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

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