Unit 3 – Decision making, Looping and Arrays

Decision Making

During programming, we have a number of situations where we may have to change the order of execution of statements based on certain conditions. This involves a kind of decision making to whether a particular condition has occurred or not and then direct to computer to execute certain statements accordingly.

C language has such decision making capabilities by supporting the following statements:

- If Statement
- Switch statement

These statements are popularly known as decision making statements. Since these statements control the flow of execution, they are also known as control statements.

Decision making with if statement

The if statement is a powerful decision making statement and is used to control the flow of execution of statements. It is basically a two-way decision statement and is used in conjunction with an expression. It takes the following form:

```
if(test expression)
```

It allows the computer to evaluate the expression first then, depending on whether the value of the expression is 'true' (or non-zero) or 'false' (zero), it transfers the control to a particular statement.

The if statement may be implemented in different forms depending on the complexity of conditions to be tested. The different forms are:

1. Simple if statement
2. if…else statement
3. Nested if…else statement

Simple if statement

The general form of a simple if statement is

```
if(test expression)
{
        Statement-block;
}
Statement-x;
```

The 'statement-block' may be a single statement or a group of statements. If the test expression is true, the statement-block will be executed; otherwise the statement-block will be skipped and the execution will jump to the statement-x.

Example

```
#include<stdio.h>
#include<conio.h>
```

Navtej Bhatt

```
void main()
{
        int age;
        clrscr();

        printf("\n Enter your age : ");
        scanf("%d",&age);

        if ( age < 0 )
                printf ("Negative age given…");

        if ( age > 0 )
                printf ("Your age is %d",age);

        getch();
}
```

Input & Output:
Enter your age: 30
Your age is 30

The if...else statement

The if…else statement is an extension of the simple if statement. The general form is

```
if(test expression)
{
        True-block statement(s);
}
else
{
        False-block statement(s);
}
Statement-x;
```

If the test expression is true, then the true-block statement(s), immediately following the if statements are executed; otherwise, the false-block statement(s) are executed. In either case, either true-block or false-block will be executed, not both.

Example: Write a C program to find maximum number from two integer numbers.

```
#include<stdio.h>
#include<conio.h>

void main()
{
        int a, b;
        clrscr();

        printf("Enter first number : ");
        scanf("%d",&a);
        printf("Enter second number : ");
```

Navtej Bhatt

```
        scanf("%d",&b);

        if ( a > b )
        {
                printf ("\n First number = %d is maximum",a);
        }
        Else
        {
                printf ("\n Second number = %d is maximum",b);
        }
        getch();
}
```

Output:

Enter first number: 54
Enter second number: 25
First number = 54 is maximum

Nested if...else statement

One if…else statement written inside another if…else statement is known as nested if…else statement. When a series of decisions are involved, we may have to use more than one if…else statement in nested form as shown below:

```
if(test condition-1)
{
        if(test condition-2)
        {
                Statement-1;
        }
        else
        {
                Statement-2;
        }
}
else
{
        Statement-3;
}
Statement-x;
```

If the condition-1 is false, the statement-3 will be executed; otherwise it continues to perform the second test. If the condition-2 is true, the statement-1 will be executed; otherwise the statement-2 will be evaluated and then the control is transferred to the statement-x.

Example: Write a C program to find maximum number from three integer numbers.

```
#include<stdio.h>
#include<conio.h>

void main()
{
        int a, b, c;
```

Navtej Bhatt

```
        clrscr();

        printf("Enter first number : ");
        scanf("%d",&a);

        printf("Enter second number : ");
        scanf("%d",&b);

        printf("Enter third number : ");
        scanf("%d",&c);


        if ( a > b  && a > c)
        {
                printf ("\n First number = %d is maximum",a);
        }
        else
        {
                if ( b > c )
                {
                        printf ("\n Second number = %d is maximum",b);
                }
                else
                {
                        printf ("\n Third number = %d is maximum",c);
                }
        }

        getch();
}
```

Output:
Enter first number: 54
Enter second number: 99
Enter second number: 85

Second number = 99 is maximum

Switch statement

C has a built-in multi way decision statement known as a switch. The switch statement tests the value of a given variable (or expression) against a list of case values and when a match is found, a block of statements associated, with that case is executed. The general form of the switch statement is as shown below:

```
switch(expression)
{
        case value-1:
                block-1;
                break;
```

Navtej Bhatt

```
        case value-2:
                block-2;
                break;
        ……
        ……

        default:
                default-block;
                break;
}
Statement-x;
```

The expression is an integer expression or characters. Value-1, value-2… are constants or constant expressions and are known as case labels. Each of these values should be unique within a switch statement. block-1, block-2…are statement list and may contains zero or more statements. There is no need to put braces around these blocks. Note that case labels end with a colon (:).

When the switch is executed, the value of the expression is successfully compared against the values value-1, value-2,… if a case is found whose value matches with the value of the expression, then the block of statements that follows the case are executed.

The break statement at the end of each block signal the end of a particular case and causes an exit from the switch statement, transferring the control to the statement-x following the switch.

The default is an optional case. When present, it will be executed if the value of the expression does not match with any of the case values. If not present, no action takes place if all matches fail and the control goes to the statement-x.

Rules for switch statement

- The switch expression must be an integral type.
- Case labels must be constants or constant expressions.
- Case labels must be unique. No two labels can have the same value.
- Case labels must end with colon.
- The default label is optional. If present, it will be executed when the expression does not find a matching case labels.
- There can be at most one default label.
- The default may be placed anywhere but usually placed at the end.
- It is permitted to nest switch statements.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int   a;
        clrscr();

        printf("\n Enter any number :");
        scanf("%d", &a);
        switch(a)
        {
```

Navtej Bhatt

```
            case 1 :
                printf("\n a is 1");
                break;
            case 2 :
                printf("\n a is 2");
                break;
            default :
                printf("\n Number other than 1 and 2");
                break;
        }
        getch();
}
```

Looping

*"When sequences of statements are executed repeatedly up to some condition then it is known as **Program Loop**."*

A loop consists mainly two parts:

Body of loop
This part contains the sequences of statements, which are to be executed repeatedly.

Control statement:
This part contains certain conditions, which indicates how many times we have to execute the body of loop. The control statement should be writing carefully otherwise it may possible your loop is executed infinite times which is known as ***Infinite Loop***. Loop contains one or more variable, which control the execution of loop that is known as the ***Loop Counter***.

Loop is executed basically in following steps:

1. Setting and initialization of Loop Counter.
2. Test the condition.
3. Executing the body of loop.
4. Incrementing or decrementing the Loop Counter value.

In above steps, 2nd and 3rd steps may be executed interchangeably that means either we can perform step 2 first or step 3 first.

In general there are two types of loops: Entry Controlled loop and Exit controlled loop.

Entry-controlled loop
In entry-controlled loop we first check the condition and then the body of loop is executed. If at first time condition is false then the body of loop and not executed any time. In C programming language *while* loop and *for* loop is example of entry-controlled loop.

Navtej Bhatt

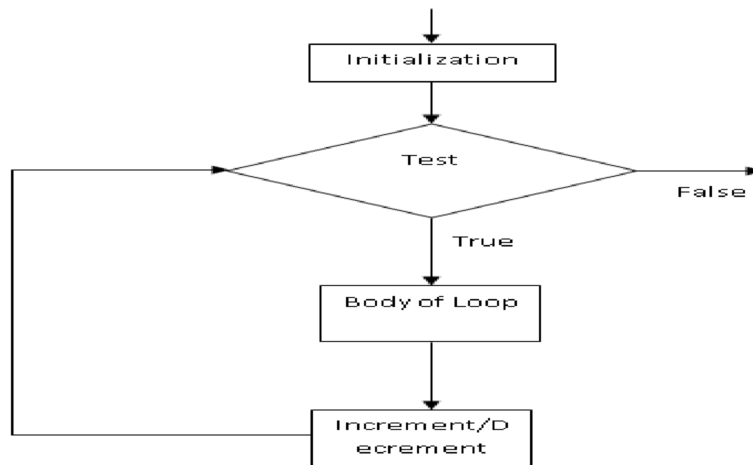


Fig. 1 Entry Controlled Loop

Exit-controlled loop

In exit-controlled loop we first execute the body of the loop and then check the condition and if condition is true then next again executed the body of the loop otherwise not. In exit-controlled loop it is sure the body of loop is executed once. In C programming language *do-while* loop is example of the exit-controlled loop.

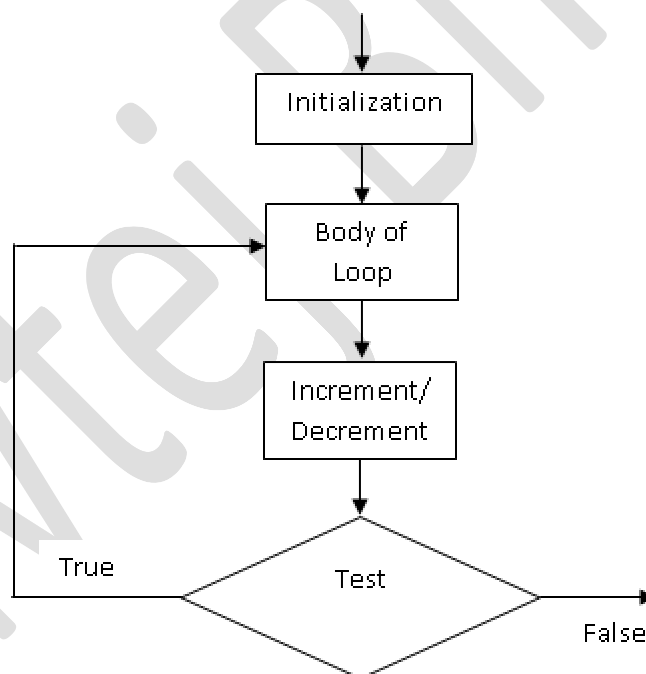


Fig. 2 Exit controlled Loop

In 'C' language we have three loops format:

- ***while*** loop
- ***for*** loop
- ***do-while*** loop

While Loop

while loop is entry-controlled loop, in which first condition is evaluated and if condition is true the body of loop is executed. After executing the body of loop the test condition is again evaluated and if it is true then again body of loop is executed. This process is going to

continue up to your test condition is true. If the test condition is false on initial stage then body of loop is never executed. The format of *while* loop is:

```
while(test condition)
{
        body of loop
}
```

Body of loop has one or more statements. Here the curly bracket is not necessary if body of loop contains only one statement. Consider following example to print 10 numbers start from 1.

```
void main( )
{
        int i;
        i=1;                    // Initialization of Loop Counter
        while(i <= 10) {        // Test condition
                printf("I = %d\n",i);   // Body of loop
                i++;            // Increment of Loop counter
        }
}
```

Here above program print the number from 1-10. First loop counter 'i' is 1. Out test condition is 'i <=10' that means the value of 'i' is less than or equal to 10. If test condition is true then we are getting the value of 'i'. After printing the 'i' value the value of 'i' is incremented by 1 and again check the test condition. This process is continuing till test condition is true.

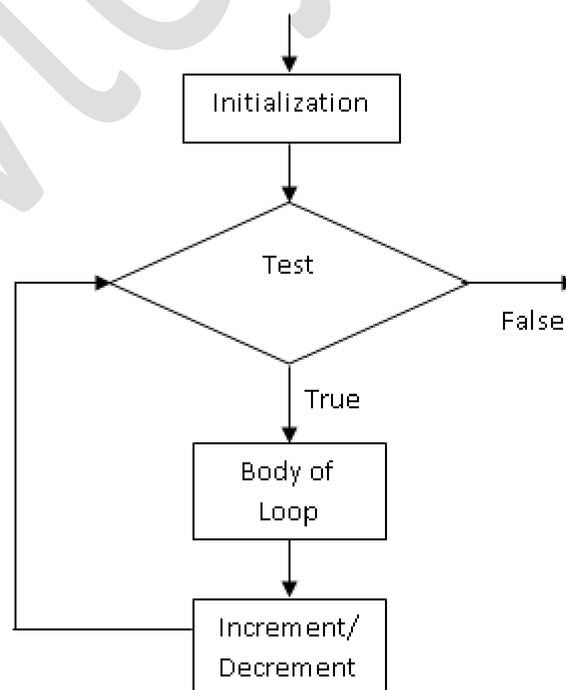


Fig. 3 *while* Loop flow chart

Navtej Bhatt

do…while loop

As we known that 'while' loop is entry-controlled loop in which test condition is evaluated first and if test condition is true then body of loop is executed. If test condition is false on first time then body of loop is never executed. But sometimes we have such a condition in which it is mandatory to execute the body of loop once whether test condition is true or false. At that time we have to use *do-while* loop. *do-while* loop is exit-controlled loop, in which first condition body of loop is executed first and then the test condition is evaluated and if the test condition is true then again body of loop is executed. This process is repeated continuously till test condition is true. Here the body of loop must be executed at least once.

The format of *do-while* loop is:

```
do
{
        body of loop
} while (test condition);
```

Consider one example of *do-while* loop in which we have to read the data till the number is less than 100 and if number is greater than 100 then we have to terminate the loop.

```
void main( )
{
        int number;
        do
        {
                printf("Enter any number : ");
                scanf("%d", &number);
                printf("You have Entered %d\n",number)
        } while(number <=100) ;
```

Here first body of loop is executed mean we are reading number and then test condition is evaluated (number is less than or equal to 100) if it is true then again we reading another number otherwise the loop is terminated.

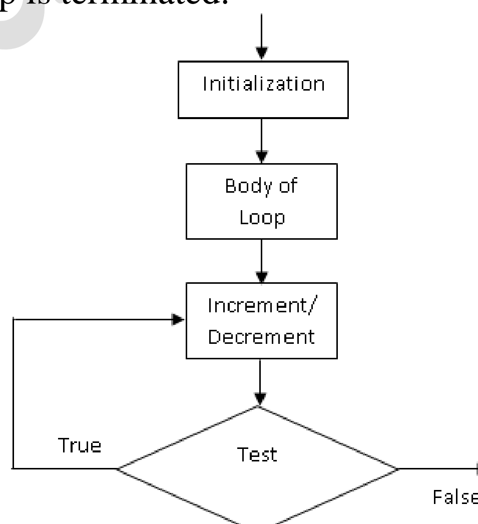


Fig. 4 *do-while* loop flow chart

Navtej Bhatt

for loop

for loop is entry-controlled loop, in which first condition is evaluated and if condition is true the body of loop is executed. After executing the body of loop the test condition is again evaluated and if it is true then again body of loop is executed. This process is going to continue up to your test condition is true. If the test condition is false on initial stage then body of loop is never executed. The format of *for* loop is:

```
for (initialization ; test-condition ; increment-decrement)
{
        body of loop
}
```

In above format the first line contain initialization, test-condition, and increment-decrement portion that means in 'for' loop initialization, condition and increment/decrement is done in only one line. First the loop counter is initialized. After initialization the test condition is evaluated and if it true then body of loop is executed and after execution of body of loop increment or decrement of loop counter is done. Again the test condition is evaluated and it is true then the same process is done again. This is continuing till the test condition is true. The flow chart of the *for* loop can be drawn in Fig. 5

Consider the example of *for* loop in which we are determining the factorial of given number.

```
void main( )
{
        int fact=1, n, i;
        n=5;            // Let assume we are determining the factorial of 5
        for ( i = 1 ; i < = n ; i++ )
        {
                fact = fact * i;
        }

        printf("The factorial = %d\n", fact);
}
```

Here the loop counter 'i' is initialized by 1 at first time. Then test condition (The value of 'i' is less than or equal to 'n') is evaluated and if it is true then 'fact' is multiplied with 'i' and again assign to 'fact'. After executing the body of loop the value of 'i' is incremented by 1. Next again test condition is evaluated and if it is true then we are proceeding in same way.

Navtej Bhatt

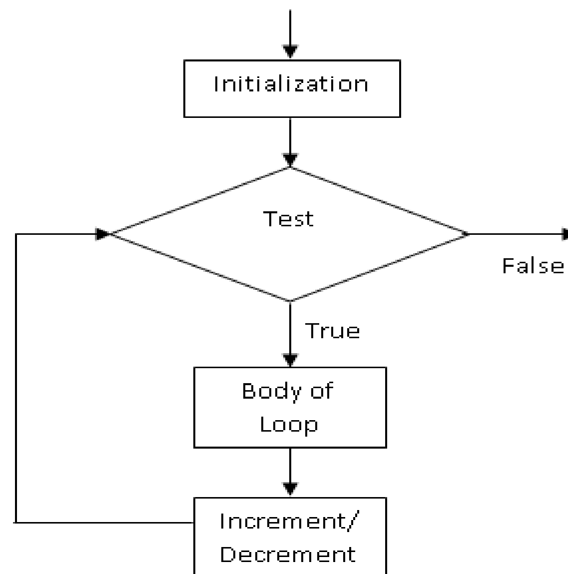


Fig. 5 *for* Loop flow chart

How *for* loop differs from other *while* loop:

- We have all initialization, condition and increment/decrement statement are only in one line.
- We can use multiple initialization or multiple increment/decrement statements by using the comma operator. For example,

```
for ( i=1, j=10 ; i<=10 ; i++, j--)
{
        ...
}
```

- We can omit any section of all section. (That means we can omit initialization, condition, or increment/decrement section) For example,

```
for( ; i< =10 ; )  // Initialization and incr/decrement section is omitted
{
        ....
}
for( ; ; ) // All sections are omitted
{
        ...
}
```

Here the second loop has omitted all the sections means this loop is infinite loop since there is no any condition to check so it is always true and body of loop is executed continuously.

DELAY (NULL) LOOP

When we put the semicolon after *for* loop or *while* loop statement then body of loop is never executed that mean we are using useless loop. This loop is used to generate the delay and so it is known as delay loop or null loop.

```
for( i =1 ; i<=10000 ; i++) ;
```

Here we have put semicolon after the loop means this loop is executed but without body of loop.

Navtej Bhatt

NESTED *for* LOOP

When we put loop inside the loop it is known as the nested loop. The nesting is allowed up to 15 levels in ANSI C but many compilers allow more. Consider the example of nested loop. In which the we are using two loops one is outer loop and another is inner loop which is used to read marks and calculate the total of 4 student

```
for( i = 1 ; i<=4 ; i++)
{
        t= 0;
        for(j=1; j< =6 ;j++)
        {
                printf("Enter the marks for %d subject : ",i);
                scanf("%d", & m);
                t= t + m;
        }
}
```

Here the outer loop is having the loop counter ‘i’ and inner loop has loop counter ‘j’. First outer loop is counter ‘i’ is initialized by 1 and if test condition (i< = 4) is true then next inner loop is executed in which the loop counter ‘j’ is initialized by 1 and if test condition (j <=6) is true then the marks of students are read and total is calculated, then the loop counter ‘j’ is incremented. This is continuing till the condition for the inner loop is true. When condition for inner loop is false the loop counter ‘i’ for outer loop is incremented and again same process is done

Jumps from loop

As we know that loop is executed repeatedly till test condition is true. But sometimes it is necessary to skip the portion of loop or exit from the loop during its execution although test condition is true. For example, we are searching one desired number from list of 100 numbers. We are executing the loop for 100 times but as soon as we are getting our desired number we have to terminate the loop.

break statement

break statement is used to terminate the loop. When computer execute *break* statement the control automatically goes to next immediate statement after loop. We cannot transfer the control at any other place. When you put *break* in nested loop then the only one loop in which *break* statement is encounter is terminated.

```
while(test condition)
{
        ...
        if(condition1)
        {
                ...
 ---------------break;
|       }
|       ...
|}
 -->...
```

Naval Bhatt

If the 'condition1' is true then computer encounter *break* statement and terminate the loop.

```
void main( )
{
        int n, a[100], i;

        // Read 100 array elements
        ...
        printf("Enter your number to be search : ");
        scanf("%d",&n);
        for( i = 0 ; i< 100 ; i++ )
        {
                if(a[i] = = n)
                {
                        printf("Your number is on position %d\n",i + 1);
                        break;
                }
        }
        getch( ) ;
}
```

Here when computer encounter your desired number then computer print the position of number and terminate the loop and go to next immediate statement *getch( )*.

***continue* statement**

During loop operations, it may be necessary to skip the part of the body of the loop during certain condition. We are using *continue* statement to skip the portion of loop. For example we are determining the square root of any number and print it. But our condition is that if number is negative then we don't want to determine square root. At that time we can use *continue* statement to skip the body of loop for negative number.

```
while(test condition)   <──────┐
{                              │
        ...                    │
        if(condition1)         │
        {                      │
                ...   ─────────┘
                continue;
        }
        ...
}
```

Let's see the example of skipping negative number printing using 'continue'. Here we are reading the number until number is not greater than 1000. If the number is negative then we have to not print it otherwise print it.

Natej Bhatt

```
void main( )
    {
        int n=1;
        while(n<=1000)
        {
            printf("Enter your number : ");
            scanf("%d", &n);
            if(n<0)
                continue;
            printf("Your number is %d\n", n);
        }
    }
```

Method of Structured Programming

Large program are difficult to understand. Hence the large programs are written using three logic structures:

1. Sequence
2. Selection
3. Iteration

A **sequence** construct is a set of instructions serially performed without any looping. The normal flow of control is from top to bottom and from left to right unless there is an iteration construct. Thus the first instruction in the program is executed first, then control passes to the next instruction that sequence.

A **selection** construct involves branching but the branches join before the exit point. The selection is required and the flow of control depends on the selection. For example, if statement, if…else statement and switch statement in C language.

An **iteration** construct contains loops in it but has only one exit point. This construct allows repeated execution of a set of codes without repeating codes in the program. It saves the trouble of writing the same codes again and again in the program. For example, while loop, do..while loop and for loop in C language.

Array

- *"Array is a group of the related data items that share a common name."*
- *"Individual elements of array are known as the elements of array."*
- *"The number specified within the square bracket during the declaration of variable is known as the array size."*
- *"The number given to each element is known as the index or subscript."*

Navtej Bhatt

For example, 20 is the size of the array since it is specified during the declaration of array.

```
int a[20];
```

In 'C' program the array index start from 0 and goes up to the (size of array –1). For example
e.g.

```
a[0]
a[1]
a[2]
…….
```

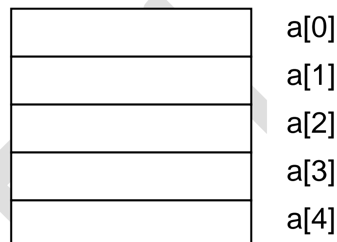
Here the number given from 0, 1, 2, … are known as the index

One Dimensional Array

"An array with only one subscript is known as the one-dimensional array of single-subscripted variable."

One-dimensional array store the list of values. For example Here we have the array with the size of 5 that means it can store maximum 5 integer elements that means we can store 5 different integer values inside it. The subscript (index) or array is start with the 0 and up to 4 (size –1). When you declare the array your computer assign continuous memory location to it. In below case 'a' has assigned 5 continuous memory locations.

int a[5];



You can access each array elements by using its subscript: Suppose you want to access the first element of the array them you can use subscript 0 (a[0]), for second element 1 (a[1]), so on. If you want to read of print array element then you can do by

```
scanf("%d",&a[0]);        // read first element
scanf("%d",&a[1]);        // read second element
…….
printf("%d",a[0]);  // print first element
printf("%d",a[1]);  // print second element
……..
```

You can read/print the five elements by using only one scanf/printf statement as under with loop.

```
for(i=0 ; i<5 ; i++)
{
        scanf("%d",&a[i]);// read 1 to 5 element as loop will increment
        printf("%d",a[i]);   // print 1 to 5 element as loop will increment
}
```

You can perform the calculation also by using the subscript. For example, below will determine the total of five elements:

total = a[0] + a[1] + a[2] + a[3] + a[4];

If suppose we assign the values 5, 10, 15, 20, and 25 to each array elements respectively.

```
a[0] = 5;
a[1] = 10;
a[2] = 15;
a[3] = 20;
a[4] = 25;
```

Then each value is successively stored into the array as under inside memory:

Value	Element
5	a[0]
10	a[1]
15	a[2]
20	a[3]
25	a[4]

Then each value is successively stored into the array as under inside memory. If you give the subscript out of the given size then it will not generate any error message but it will give the unexpected output during execution.

Declaration of Array

Like any other variables, arrays must be declared before they are used. The general format of declaration of array is:

```
datatype    arrayname[size];
```

The 'datatype' indicates whether your array is integer type, float type or any other type. 'arrayname' is a name given to array, and 'size' indicates the no. of elements in your array. For example, here array name is 'height' of float type and it contains different 10 elements. Here the valid subscript range is 0-9.

```
float height[10];
```

REPRESENTATION OF STRING USING ARRAY OF THE CHARACTER

In C program the array of character represents the string. In string the size indicates your string contains how many maximum characters? For example, here we have the array of the character 'name' which is known as string and here its size is 5 and you can store maximum 4 characters in your string because in string the last character is null character '\0' or NULL. Suppose we are storing string "Well" in array 'name' then it is stored as:

```
char name[5];
```

Navtej Bhatt

‘W’	name[0]
‘e’	name[1]
‘l’	name[2]
‘l’	name[3]
‘\0’	name[4]

As shown above all the characters of the string are stored in continuous memory location and the last character of your string is NULL character.

Initialization of one dimensional array

The general format of the initialization of array is:

```
datatype    arrayname[size]={list of values separated by comma};
```

To initialize your array you have to give list of the values separated by comma in curly bracket. For example,

```
int    x[5] = { 10, 20, 30, 40, 50};
```

The array variable of size 5 ‘x’ is initialized by five different values 10, 20, 30, 40 and 50 respectively that means

```
X[0]  = 10;
X[1]  = 20;
X[2]  = 30;
X[3]  = 40;
X[4]  = 50;
```

You can initialize the character array or string by two ways. For example,

```
char name[5] = { ‘W’,’e’,’l’,’l’,’\0’};
        or
char name[5] = “Well”;
```

You can initialize the array elements without specifying the size of array. For example,

```
int  x[ ] = { 10, 20, 30, 40, 50};
```

Here we have given the empty bracket and initialized by five variables that means by default the size is 5.

Initialization of array has two drawbacks or limitation:

1. There is no any way to initialize selected array elements.
2. There is no any shortcut method to initialize the large no. of array elements.

Navtej Bhatt

Two Dimensional Array

Two-dimensional array can store the table of values.

"An array with two subscript is known as the two-dimensional array of double-subscripted variable."

The two-dimensional array can be represented by the matrix in mathematics. The matrix elements can be represented by V_{ij} where 'i' value indicates the row number and 'j' value indicated the column number. We can declare the two-dimensional array by using following format:

datatype arrayname[row size][column size];

For example, in following declaration we have array 'x' contains 2 rows and 2 columns. The array elements can be represented by following way:

int x[2][2];

	Column: 1	Column: 2
Row 1	[0][0]	[0][1]
Row 2	[1][0]	[1][1]

These array elements can be accessed by using the two subscript one for row and another for the column. For example, suppose we have to read the value of the two-dimensional array and print it.

```
void main( )
{
	int a[5][5];		// Declaration of array with the size = 5
	int i, j;

	// read the array elements value
	for(i = 0 ; i < 5 ; i ++)
	{
		for(j = 0 ; j < 5 ; j++)
		{
			scanf("%d",&a[i][j]);
		}
	}
	// print the array elements value
	for(i = 0 ; i < 5 ; i ++)
	{
		for(j = 0 ; j < 5 ; j++)
		{
			printf("%d",a[i][j]);
		}
	}
```

Navtej Bhatt

INITIALIZATION OF TWO-DIMENSIONAL ARRAY

We can initialize two-dimensional array elements by following way:

```
datatype arrayname[row size][column size] = { {list of values of 1st row},
                                               {list of values of 2nd row},
                                            …… };
```

We can also initialize the array elements continuously as:

```
datatype arrayname[row size][column size] = { {list of values };
```

Here the elements are initialized continuously from first row, then second row, and so on. For example

```
int a[2][2] = {{0,0}, {1,1}};

        OR

int a[2][2] = {0,0,1,1};
```

Here the first row elements are initialized by zero and second rows elements are initialized by 1.

If you want to initialize the all the array elements with the zero then you can do by:

```
int a[2][2]={ 0 };
```

Navtej Bhatt