By
Trupthi.M
Asst.Prof, IT Department,
CBIT, Gandipet, Hyderabad

What is Function in C Language?

A function in C language is a block of code that performs a specific task. It has a name and it is reusable i.e. it can be executed from as many different parts in a C Program as required. It also optionally returns a value to the calling program

So function in a C program has some properties discussed below.

- Every function has a unique name. This name is used to call function from "main()" function. A function can be called from within another function.
- A function is independent and it can perform its task without intervention from or interfering with other parts of the program.
- A function performs a specific task. A task is a distinct job that your program must perform as a part of its overall operation, such as adding two or more integer, sorting an array into numerical order, or calculating a cube root etc.
- A function returns a value to the calling program. This is optional and depends upon the task your function is going to accomplish. Suppose you want to just show few lines through function then it is not necessary to return a value. But if you are calculating area of rectangle and wanted to use result somewhere in program then you have to send back (return) value to the calling function.

C language is collection of various inbuilt functions. If you have written a program in C then it is evident that you have used C's inbuilt functions. Printf, scanf, clrscr etc. all are C's inbuilt functions. You cannot imagine a C program without function.

Structure of a Function

A general form of a C function looks like this:

```
<return type> FunctionName (Argument1, Argument2, Argument3……)
{
Statement1;
Statement2;
Statement3;
}
```

An example of function.

```
int sum (int x, int y)
{
int result;
result = x + y;
return (result);
}
```

Advantages of using functions:

There are many advantages in using functions in a program they are:

1. It makes possible top down modular programming. In this style of programming, the high level logic of the overall problem is solved first while the details of each lower level functions is addressed later.
2. The length of the source program can be reduced by using functions at appropriate places.
3. It becomes uncomplicated to locate and separate a faulty function for further study.
4. A function may be used later by many other programs this means that a c programmer can use function written by others, instead of starting over from scratch.
5. A function can be used to keep away from rewriting the same block of codes which we are going use two or more locations in a program. This is especially useful if the code involved is long or complicated.

Types of functions:

A function may belong to any one of the following categories:

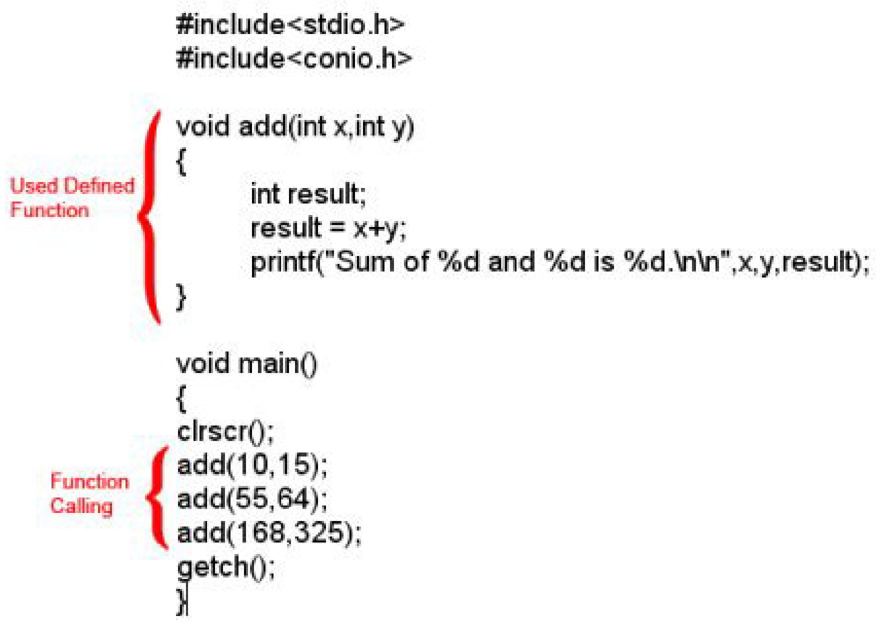
1. Functions with no arguments and no return values.
2. Functions with arguments and no return values.
3. Functions with arguments and return values.
4. Functions that return multiple values.
5. Functions with no arguments and return values.

1.

Example of a simple function to add two integers.

view plain

```
#include<stdio.h>
#include<conio.h>
void add(int x,int y)
{
int result;
result = x+y;
printf("Sum of %d and %d is %d.\n\n",x,y,result);
}
void main()
{
clrscr();
add(10,15);
add(55,64);
add(168,325);
getch();
}
```

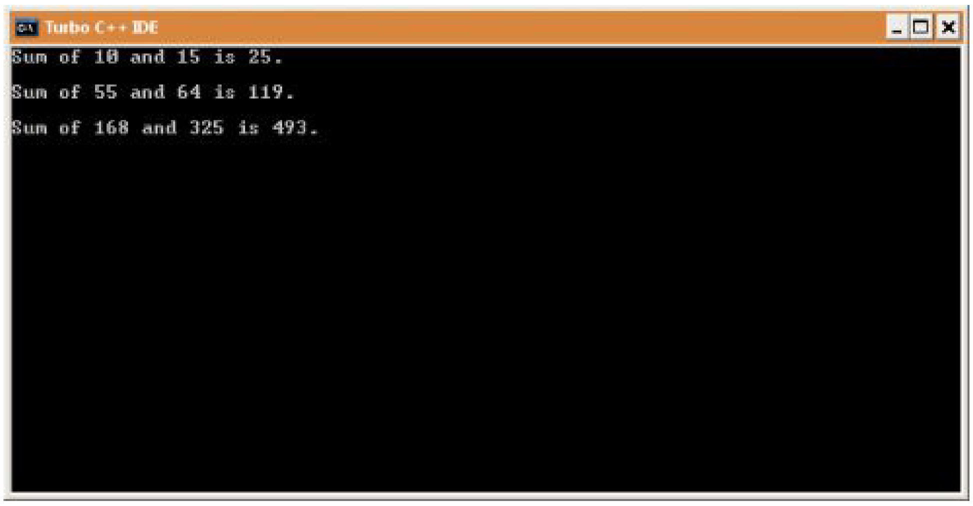


```
#include<stdio.h>
#include<conio.h>

void add(int x,int y)
{
        int result;
        result = x+y;
        printf("Sum of %d and %d is %d.\n\n",x,y,result);
}

void main()
{
clrscr();
add(10,15);
add(55,64);
add(168,325);
getch();
}
```

Program Output



```
Sum of 10 and 15 is 25.
Sum of 55 and 64 is 119.
Sum of 168 and 325 is 493.
```

Output of above program.

Types of Function in C Programming Languages:

five types of functions and they are:

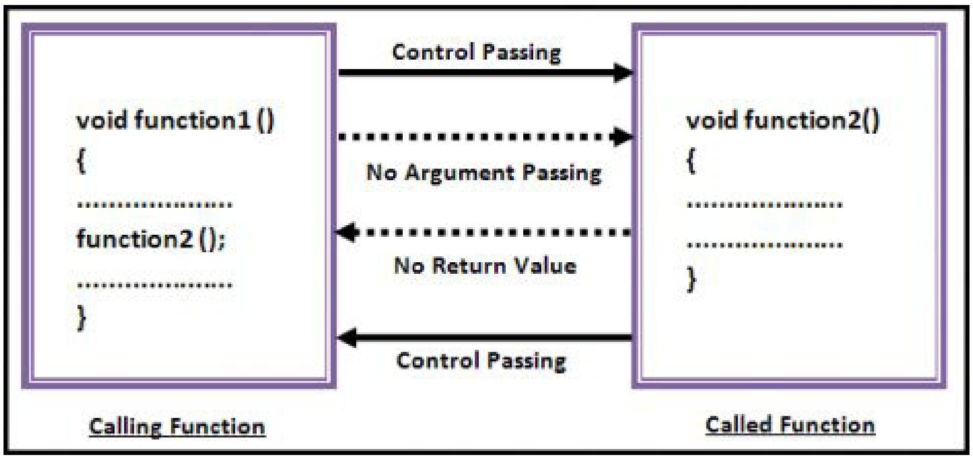
1. Functions with no arguments and no return values.
2. Functions with arguments and no return values.
3. Functions with arguments and return values.
4. Functions that return multiple values.
5. Functions with no arguments and return values.

1. Functions with no arguments and no return value.

A C function without any arguments means you cannot pass data (values like int, char etc) to the called function. Similarly, function with no return type does not pass back data to the calling function. It is one of the simplest types of function in C. This type of function which does not return any value cannot be used in an expression it can be used only as independent statement. Let’s have an example to illustrate this.

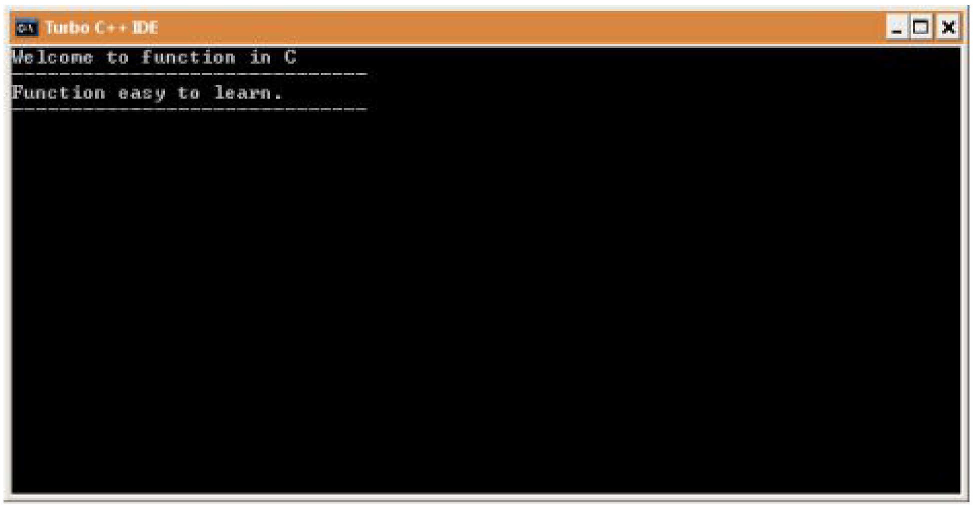
```
#include<stdio.h>
#include<conio.h>
void printline()
{
int i;
printf("\n");
for(i=0;i<30;i++)
{
printf("-");
}
printf("\n");
}
void main()
```

```
{
clrscr();
printf("Welcome to function in C");
printline();
printf("Function easy to learn.");
printline();
getch();
}
```



Logic of the functions with no arguments and no return value.

Output



```
Welcome to function in C
------------------------------
Function easy to learn.
------------------------------
```

Output of above program.

Source Code Explanation:

The above C program example illustrates that how to declare a function with no argument and no return type. I am going to explain only important lines only because this C program example is for those who are above the beginner level.

Line 3-12: This C code block is a user defined function (UDF) whose task is to print a horizontal line. This is a simple function and a basic programmer can understand this. As you can see in line no. 7 I have declared a “for loop” which loops 30 time and prints “-” symbol continuously.

Line 13-21: These line are “main()” function code block. Line no. 16 and 18 simply prints two different messages. And line no. 17 and 18 calls our user defined function “printline()”. You can see output this program below

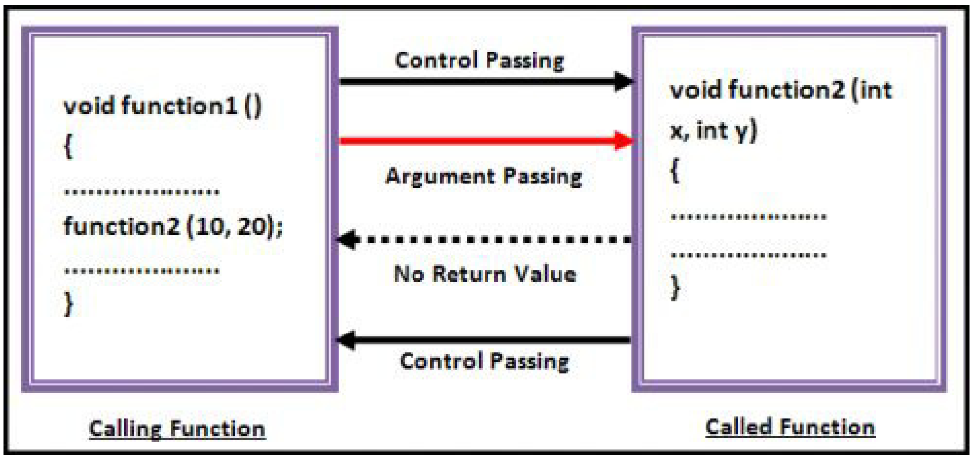
2. Functions with arguments and no return value.

In our previous example what we have noticed that “main()” function has no control over the UDF “printfline()”, it cannot control its output. Whenever “main()” calls “printline()”, it simply prints line every time. So the result remains the same.

A C function with arguments can perform much better than previous function type. This type of function can accept data from calling function. In other words, you send data to the called function from calling function but you cannot send result data back to the calling function. Rather, it displays the result on the terminal. But we can control the output of function by providing various values as arguments. Let’s have an example to get it better.

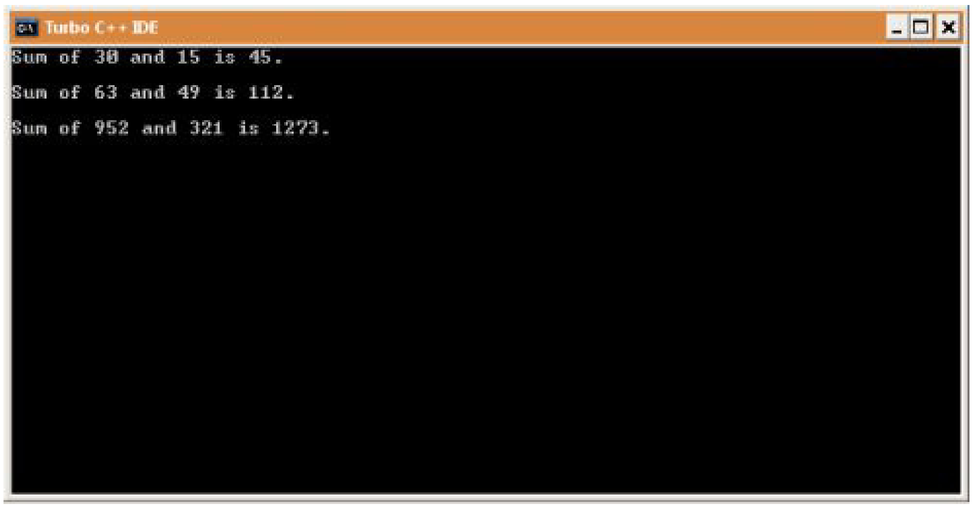
```
1.    #include<stdio.h>
2.    #include<conio.h>
3.    void add(int x, int y)
4.    {
```

```
5.    int result;
6.    result = x+y;
7.    printf("Sum of %d and %d is %d.\n\n",x,y,result);
8.    }
9.    void main()
10.   {
11.   clrscr();
12.   add(30,15);
13.   add(63,49);
14.   add(952,321);
15.   getch();
16.   }
```



Logic of the function with arguments and no return value.

Output



```
Sum of 30 and 15 is 45.

Sum of 63 and 49 is 112.

Sum of 952 and 321 is 1273.
```

Output of above program.

Source Code Explanation:

This program simply sends two integer arguments to the UDF “add()” which, further, calculates its sum and stores in another variable and then prints that value. So simple program to understand.

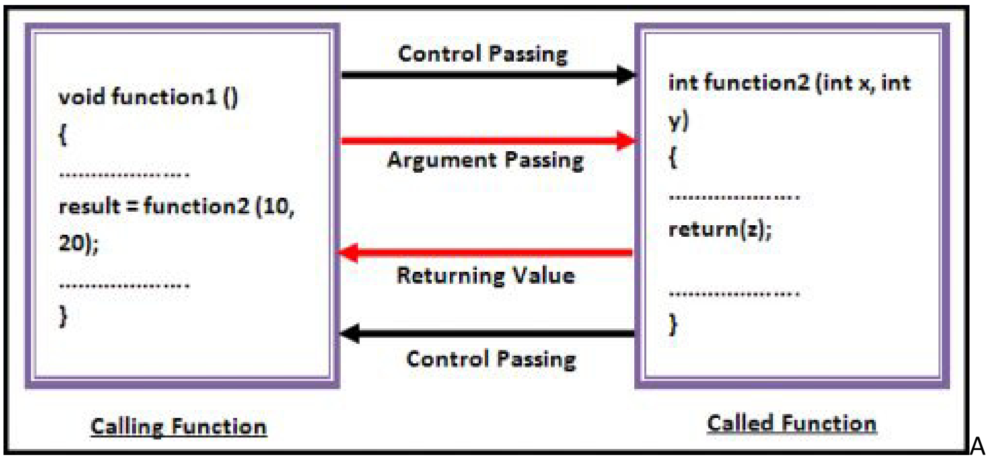
Line 3-8: This C code block is “add()” which accepts two integer type arguments. This UDF also has a integer variable “result” which stores the sum of values passed by calling function (in this example “main()”). And line no. 7 simply prints the result along with argument variable values.

Line 9-16: This code block is a “main()” function but only line no. 12, 13, 14 is important for us now. In these three lines we have called same function “add()” three times but with different values and each function call gives different output. So, you can see, we can control function’s output by providing different integer parameters which was not possible in function type 1. This is the difference

3. Functions with arguments and return value.

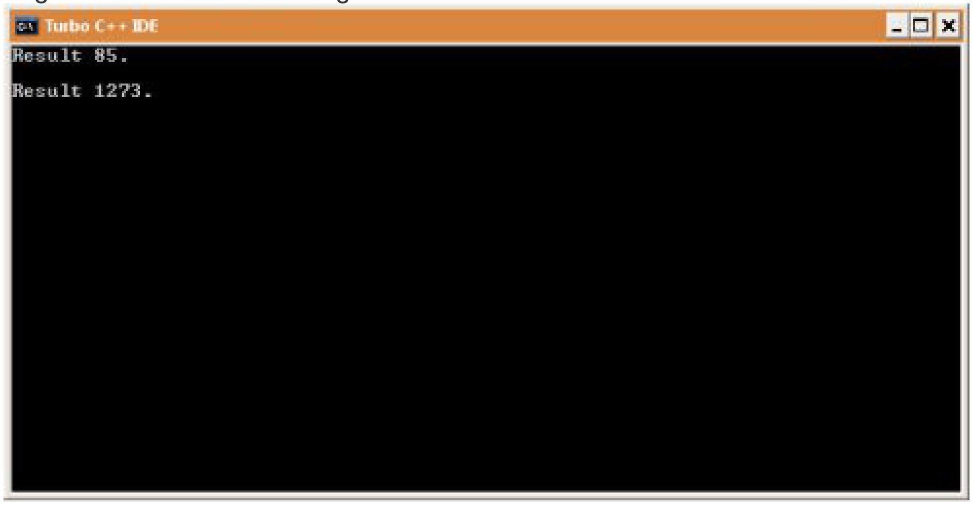
This type of function can send arguments (data) from the calling function to the called function and wait for the result to be returned back from the called function back to the calling function. And this type of function is mostly used in programming world because it can do two way communications; it can accept data as arguments as well as can send back data as return value. The data returned by the function can be used later in our program for further calculations.

```
1.     #include<stdio.h>
2.     #include<conio.h>
3.     int add(int x, int y)
4.     {
5.     int result;
6.     result = x+y;
7.     return(result);
8.     }
9.     void main()
10.    {
11.    int z;
12.    clrscr();
13.    z = add(952,321);
14.    printf("Result %d.\n\n",add(30,55));
15.    printf("Result %d.\n\n",z);
16.    getch();
17.    }
```



A

Logic of the function with arguments and return value.



```
Result 85.
Result 1273.
```

Output of the above program.

Source Code Explanation:

This program sends two integer values (x and y) to the UDF “add()”, “add()” function adds these two values and sends back the result to the calling function (in this program to “main()” function). Later result is printed on the terminal.

Line No. 3-8: Look line no. 3 carefully, it starts with **int**. This int is the return type of the function, means it can only return integer type data to the calling function. If you want any function to return character values then you must change this to char type. On line no. 7 you can see return statement, return is a keyword and in bracket we can give values which we want to return. You can assign any integer value to experiment with this return which ultimately will change its output. Do experiment with all you program and don’t hesitate.

Line No. 9-17: In this code block only line no. 13, 14 and 15 is important. We have declared an integer “z” which we used in line no. 13. Why we are using integer variable “z” here? You know that our UDF “add()” returns an integer value on calling. To store that value we have declared an integer value. We have passed 952, 321 to the “add()” function, which finally return 1273 as result. This value will be stored in “z” integer variable. Now we can use “z” to print its value or to other function.

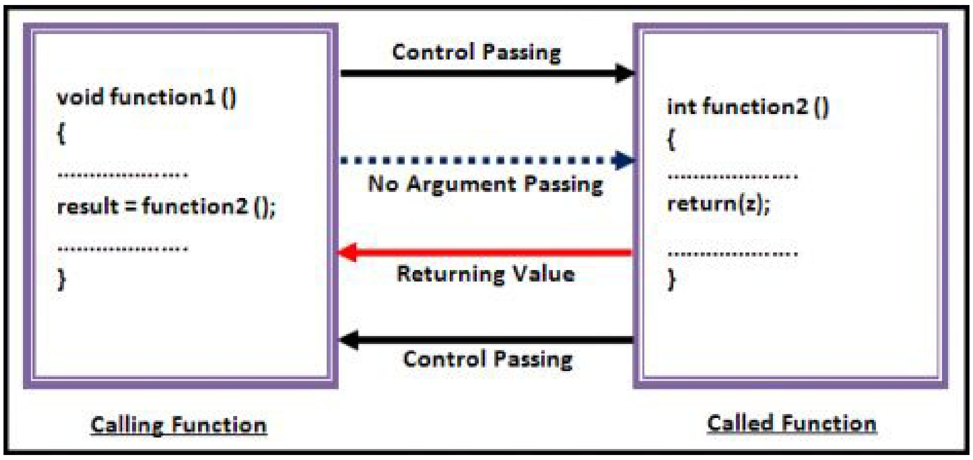
You will also notice some strange statement in line no. 14. Actually line no. 14 and 15 does the same job, in line no. 15 we have used an extra variable whereas on line no. 14 we directly printed the value without using any extra variable. This was simply to show you how we can use function in different ways.

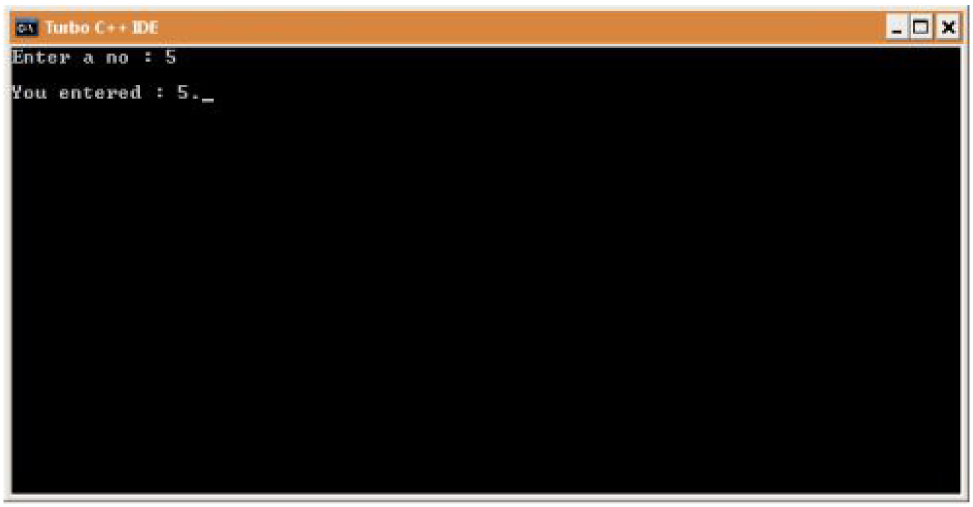
4. Functions with no arguments but returns value.

We may need a function which does not take any argument but only returns values to the calling function then this type of function is useful. The best example of this type of function is “getchar()” library function which is declared in the header file “stdio.h”. We can declare a similar library function of own. Take a look.

```
1.     #include<stdio.h>
2.     #include<conio.h>
3.     int send()
4.     {
5.     int no1;
6.     printf("Enter a no : ");
7.     scanf("%d",&no1);
8.     return(no1);
9.     }
10.    void main()
11.    {
12.    int z;
13.    clrscr();
14.    z = send();
15.    printf("\nYou entered : %d.", z);
16.    getch();
17.    }
```

Functions with no arguments and return values.





```
Enter a no : 5
You entered : 5._
```

Output of the above program.

Source Code Explanation:

In this program we have a UDF which takes one integer as input from keyboard and sends back to the calling function. This is a very easy code to understand if you have followed all above code explanation. So I am not going to explain this code. But if you find difficulty please post your problem and I will solve that.

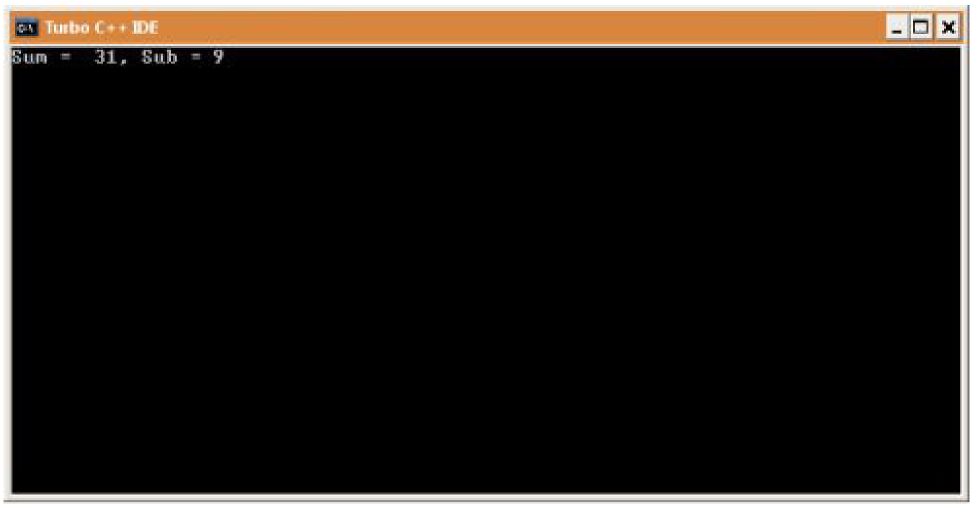
5. Functions that return multiple values.

So far, we have learned and seen that in a function, return statement was able to return only single value. That is because; a return statement can return only one value. But if we want to send back more than one value then how we could do this?

We have used arguments to send values to the called function, in the same way we can also use arguments to send back information to the calling function. The arguments that are used to send back data are called **Output Parameters**.

It is a bit difficult for novice because this type of function uses pointer. Let's see an example:

```
#include<stdio.h>
#include<conio.h>
void calc(int x, int y, int *add, int *sub)
{
*add = x+y;
*sub = x-y;
}
void main()
{
int a=20, b=11, p,q;
clrscr();
calc(a,b,&p,&q);
13.    printf("Sum =  %d, Sub = %d",p,q);
14.    getch();
15.    }
```



```
Sum =  31, Sub = 9
```

Output of the above program.

Source Code Explanation:

Logic of this program is that we call UDF “calc()” and sends argument then it adds and subtract that two values and store that values in their respective pointers. The “*” is known as indirection operator whereas “&” known as address operator. We can get memory address of any variable by simply placing “&” before variable name. In the same way we get value stored at specific memory location by using “*” just before memory address. These things are a bit confusing but when you will understand pointer then these thing will become clearer.

Line no. 3-7: This UDF function is different from all above UDF because it implements pointer. I know line no. 3 looks something strange, let’s have a clear idea of it. “Calc()” function has four arguments, first two arguments need no explanation. Last two arguments are integer pointer which works as output parameters (arguments). Pointer can only store address of the value rather than value but when we add * to pointer variable then we can store value at that address.

Line no. 8-15: When we call “calc()” function in the line no. 12 then following assignments occurs. Value of variable “a” is assigned to “x”, value of variable “b” is assigned to “y”, address of “p” and “q” to “add” and “sub” respectively. In line no. 5 and 6 we are adding and subtracting values and storing the result at their respective memory location. This is how the program works.

Storage Classes in C Language

STORAGE CLASSES:

The storage class determines the part of member storage is allocated for an object and how long the storage allocation continues to exit.

Storage class tells us:

1) Where the variable is stored.

2) Initial value of the variable.

3) Scope of the variable. Scope specifies the part of the program which a variable is accessed.

4) Life of the variable.

There are four types of storage classes:

1) Automatic storage class

2) Register storage class

3) Static storage class

4) External storage class

AUTOMATIC STORAGE CLASS:

In this automatic storage class,

Variable is stored in memory.

Default value is garbage value

Scope is local to the block

Life is, with in the block in which the variable is defined

Example 1:

```
main()

{

auto int i=10;

printf("%d",i);

}
```

Output:

```
10
```

Example 2:

```
main()

{

auto int i;

printf(“%d”,i);

}
```

Output:

```
1002
```

In example 1, i value is initialised to 10.So,output is 10.

In example 2, i value is not initialised.So,compiler reads i value is a garbage value.

REGISTER STORAGE CLASS:

Variable is stored in CPU registers.

Default value is garbage value.

Scope is local to the block.

Life is,with in the block in which the variable is defined.

We can not use register storage class for all types of variables.

For example:

```
register float f;

register double d;

register long l;
```

Example :

```
main()

{

register int i=10;

printf(“%d”,i);

}
```

Output:

```
10
```

STATIC STORAGE CLASS:

Variable is stored in memory.

Default value is zero.

Scope is local to the block.

Life is,value of the variable persists between different function calls.

Example :

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

Output:

```
10
11
```

EXTERNAL STORAGE CLASS:

Variable is stored in memory.

Default value is zero.

Scope is local to the block.

Life is,as long as the program execution doesn’t come to an end.

Example :

```
int i=10;

main()

{

int i=2;

printf(“%d”,i);

display();

}

display()

{

printf(“\n%d”,i);

}
```

Output:

```
2

10
```

In above example,i declared in two places.One is above the main(),second is within main().Within main() i is used in within main() only.Before main() i is used to outof the main() functions.i.e, display() uses the i value as 10.

NOTE:

Static and **auto** storage classes both are different in the case of life.Remaining all r same.

In the case of recursive function calls we use s**tatic** storage class.

Register storage classes are used in the case of frequently used variables.Because these are stored in CPU registers.

Extern storage class for only those variables are used almost all functions in the program.This would avoid unnecessary passing of the variables.

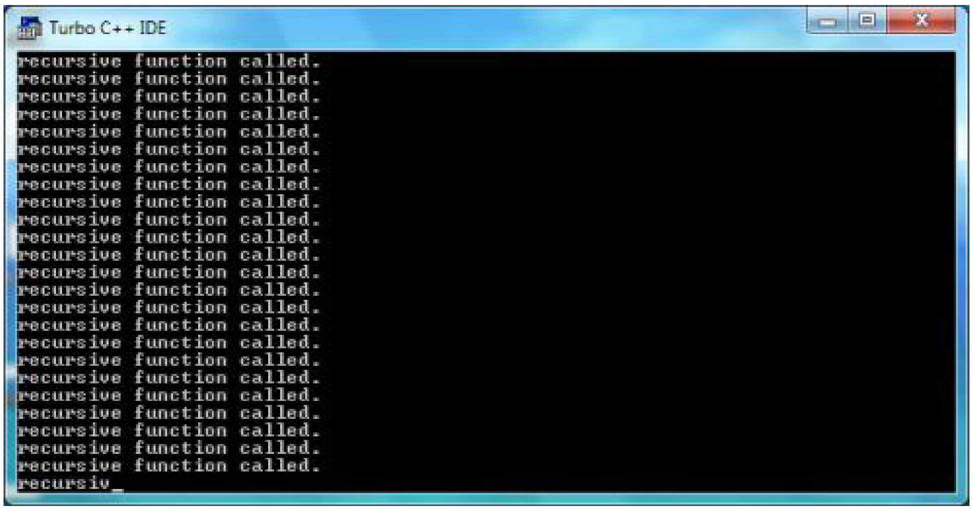
C Programming - Recursive Function

We have learnt different types function C language and now I am going to explain recursive function in C. A function is called “recursive” if a statement within body of that function calls the same function for example look at below code:

```
void main()
{
printf(“recursive function called.\n”);
main();
}
```

When you will run this program it will print message “recursive function called.” indefinitely. If you are using Turbo C/C++ compiler then you need to press Ctrl + Break key to break this in definite loop.

Recursive function example output



```
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursiv_
```

Before we move to another example lets have attributes of “recursive function”:-

1. A recursive function is a function which calls itself.
2. The speed of a recursive program is slower because of stack overheads. (This attribute is evident if you run above C program.)
3. A recursive function must have recursive conditions, terminating conditions, and recursive expressions.

Calculating factorial value using recursion

To understand how recursion works lets have another popular example of recursion. In this example we will calculate the factorial of n numbers. The factorial of n numbers is expressed as a series of repetitive multiplication as shown below:

Factorial of n = n(n-1)(n-2)……1.

Example :

Factiorial of 5 = 5x4x3x2x1

=120

view plain

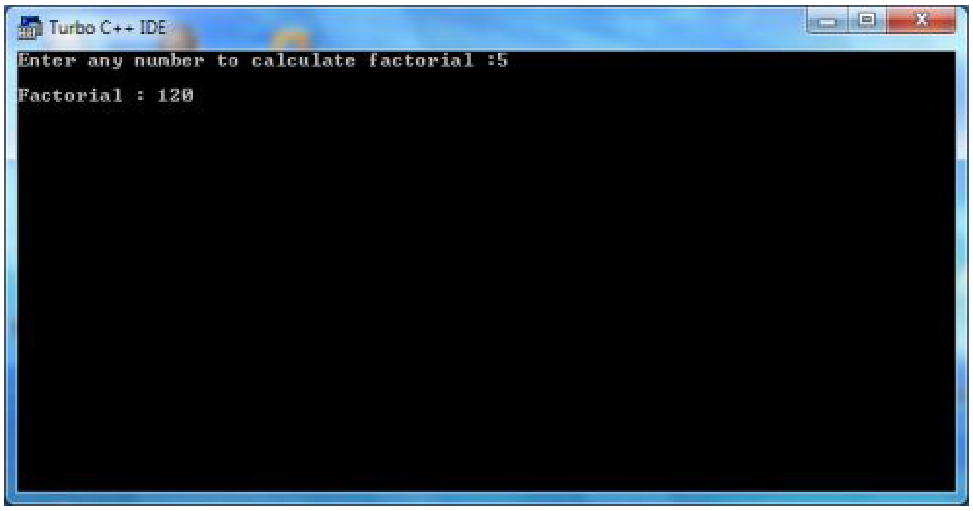
```
#include<stdio.h>
#include<conio.h>

int factorial(int);

int factorial (int i)
{
    int f;
    if(i==1)
    return 1;
    else
    f = i* factorial (i-1);
return f;
}

void main()
{
    int x;
    clrscr();
    printf("Enter any number to calculate factorial :");
    scanf("%d",&x);
    printf("\nFactorial : %d", factorial (x));
    getch();
}
```

Factorial value using recursive function output



```
Enter any number to calculate factorial :5
Factorial : 120
```

So from **line no. 6 – 14** is a user defined recursive function "**factorial**" that calculates factorial of any given number. This function accepts integer type argument/parameter and return integer value. If you have any problem to understand how function works then you can check my tutorials on C function (click here).

In **line no. 9** we are checking that whether value of i is equal to 1 or not; i is an integer variable which contains value passed from main function i.e. value of integer variable x. If user enters 1 then the factorial of 1 will be 1. If user enters any value greater than 1 like 5 then it will execute statement in **line no. 12** to calculate factorial of 5. This line is extremely important because in this line we implemented recursion logic.

Let's see how **line no. 12** exactly works. Suppose value of i=5, since i is not equal to 1, the statement:

f = i* factorial (i-1);

will be executed with i=5 i.e.

f = 5* factorial (5-1);

will be evaluated. As you can see this statement again calls factorial function with value i-1 which will return value:

4*factorial(4-1);

This recursive calling continues until value of i is equal to 1 and when i is equal to 1 it returns 1 and execution of this function stops. We can review the series of recursive call as follow:

f = 5* factorial (5-1);

f = 5*4* factorial (4-1);

f = 5*4*3* factorial (3-1);

f = 5*4*3*2* factorial (2-1);

f = 5*4*3*2*1;

f = 120;

I hope this will clear any confusion regarding recursive function.

Array in C programming – Programmer's view

What is an Array?

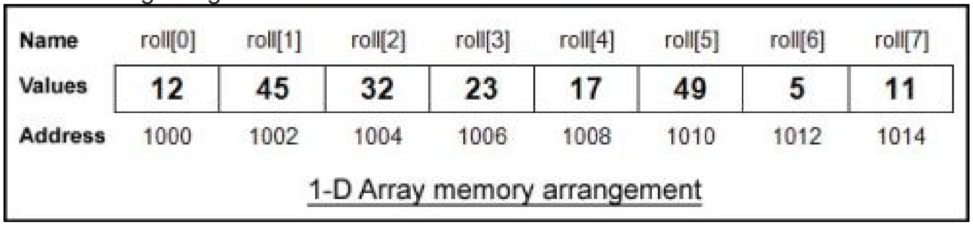
An array in C language is a collection of similar data-type, means an array can hold value of a particular data type for which it has been declared. Arrays can be created from any of the C data-types int, float, and char. So an integer array can only hold integer values and cannot hold values other than integer. When we declare array, it allocates contiguous memory location for storing values whereas 2 or 3 variables of same data-type can have random locations. So this is the most important difference between a variable and an array.

Types of Arrays:

1. One dimension array (Also known as 1-D array).
2. Two dimension array (Also known as 2-D array).
3. Multi-dimension array.

Decla

Syntax: data_type array_name[width];
Example: int roll[8];
In our example, int specifies the type if the variable, roll specifies the name of the variable and the value in bracket [8] is new for newbie. The bracket ([ ]) tells compiler that it is an array and number mention in the bracket specifies that how many elements (values in any array is called elements) it can store. This number is called dimension of array.
So, with respect to our example we have declared an array of integer type and named it “roll” which can store roll numbers of 8 students. You can see memory arrangement of above declared array in the following image:



Name	roll[0]	roll[1]	roll[2]	roll[3]	roll[4]	roll[5]	roll[6]	roll[7]
Values	12	45	32	23	17	49	5	11
Address	1000	1002	1004	1006	1008	1010	1012	1014

1-D Array memory arrangement

4.
5. One Dimensional Array Memory Arrangement.

C Array Assignment and Initialization:

We can initialize and assign values to the arrays in the same way as we do with variable. We can assign value to an array at the time of declaration or during runtime. Let’s look at each approach.

Syntax: data_type array_name[size]={list of values};

Example:
int arr[5]={1,2,3,4,5};
int arr[]={1,2,3,4,5};

In our above array example we have declared an integer array and named it “arr” which can hold 5 elements, we are also initializing arrays in the same time.

Both statements in our example are valid method to declare and initialize single dimension array. In our first example we mention the size (5) of an array and assigned it values in curly brace, separating element’s value by comma (,). But in second example we left the size field blank but we provided its element’s value.

When we only give element values without providing size of an array then C compiler automatically assumes its size from given element values.

There is one more method to initialize array C programming; in this method we can assign values to individual element of an array. For this let’s look at example:

Array Initialization Example

view plain

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

void main()
{
int arr[5],i;
clrscr();
arr[0]=10;
arr[1]=20;
arr[2]=30;
arr[3]=40;
arr[4]=50;

printf("Value in array arr[0] : %d\n",arr[0]);
printf("Value in array arr[1] : %d\n",arr[1]);
printf("Value in array arr[2] : %d\n",arr[2]);
printf("Value in array arr[3] : %d\n",arr[3]);
printf("Value in array arr[4] : %d\n",arr[4]);
printf("\n");

for(i=0;i<5;i++)
{
    printf("Value in array arr[%d] : %d\n",i,arr[i]);
}
getch();
}
```

In the above c arrays example we have assigned the value of integer array individually like we do with an integer variable. We have called array element’s value individually and using for loop so that it would be clear for beginner and semi-beginner C programmers. So, from the above example it is evident that we can assign values to an array element individually and can call them individually whenever we need them.

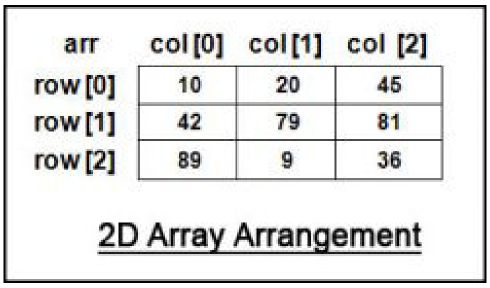
How to work with Two Dimensional Arrays in C

We know how to work with an array (1D array) having one dimension. In C language it is possible to have more than one dimension in an array. In this tutorial we are going to learn how we can use two dimensional arrays (2D arrays) to store values. Because it is a 2D array so its structure will be different from one dimension array. The 2D array is also known as Matrix or Table, it is an array of array. See the below image, here each row is an array.

Declaration of 2D array:

Syntax: data_type array_name[row_size][column_size];
Example: int arr[3][3];
So the above example declares a 2D array of integer type. This integer array has been named arr and it can hold up to 9 elements (3 rows x 3 columns).

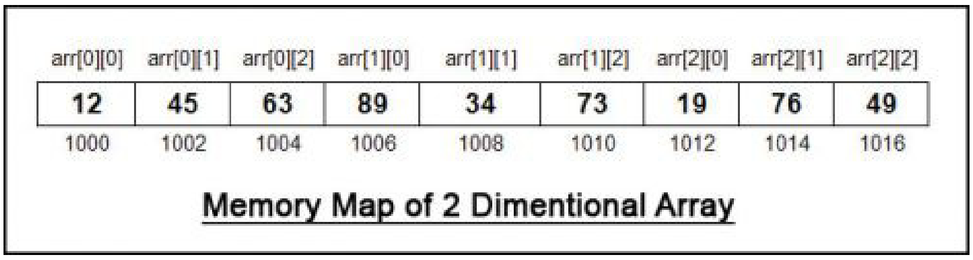
2D Array



arr	col [0]	col [1]	col [2]
row [0]	10	20	45
row [1]	42	79	81
row [2]	89	9	36

2D Array Arrangement

Memory Map of 2D Array



arr[0][0]	arr[0][1]	arr[0][2]	arr[1][0]	arr[1][1]	arr[1][2]	arr[2][0]	arr[2][1]	arr[2][2]
12	45	63	89	34	73	19	76	49
1000	1002	1004	1006	1008	1010	1012	1014	1016

Memory Map of 2 Dimentional Array

Code for assigning & displaying 2D Array

view plain

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

void main()
{
```

```c
int i, j;
int arr[3][3]={
        {12, 45, 63},
        {89, 34, 73},
        {19, 76, 49}
        };
clrscr();
printf(":::2D Array Elements:::\n\n");
for(i=0;i<3;i++)
{
    for(j=0;j<3;j++)
    {
    printf("%d\t",arr[i][j]);
    }
    printf("\n");
}
getch();
}
```

So in the above example we have declared a 2D array named arr which can hold 3x3 elements. We have also initialized that array with values, because we told the compiler that this array will contain 3 rows (0 to 2) so we divided elements accordingly. Elements for column have been differentiated by a comma (,). When compiler finds comma in array elements then it assumes comma as beginning of next element value. We can also define the same array in other ways, like.
int arr[3][3]={12, 45, 63, 89, 34, 73, 19, 76, 49}; or,
int arr[ ][3]={12, 45, 63, 89, 34, 73, 19, 76, 49};

But this kind of declaration is not acceptable in C language programming.

int arr[2][ ]={12, 45, 63, 89, 34, 73, 19, 76, 49}; or,
int arr[ ][ ]={12, 45, 63, 89, 34, 73, 19, 76, 49};

To display 2D array elements we have to just point out which element value we want to display. In our example we have a arr[3][3], so the array element reference will be from arr[0][0] to arr[2][2]. We can print display any element from this range. But in our example I have used for loop for my convenience, otherwise I had to write 9 printf statements to display all elements of array. So for loop i handles row of 2D array and for loop j handles column. I have formatted the output display of array so that we can see the elements in tabular form.

How to work with Multidimensional Array in C Programming

C allows array of two or more dimensions and maximum numbers of dimension a C program can have is depend on the compiler we are using. Generally, an array having one dimension is called 1D array, array having two dimensions called 2D array and so on. So in C programming an array can have two or three or four or even ten or more dimensions. More dimensions in an array means more data it can hold and of course more difficulties to manage and understand these arrays. A multidimensional array has following syntax:

Syntax:

type array_name[d1][d2][d3][d4]………[dn];
Where dn is the size of last dimension.

Example:

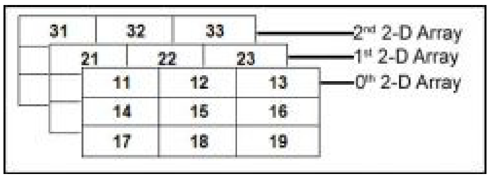
int table[5][5][20];
float arr[5][6][5][6][5];

In our example array “table” is a 3D (A 3D array is an array of arrays of arrays.) array which can hold 500 integer type elements. And array “arr” is a 5D array which can hold 4500 floating-point elements. Can see the power of array over variable? When it comes to hold multiple values in a C programming, we need to declare several variables (for example to store 150 integers) but in case of array, a single array can hold thousands of values (depending on compiler, array type etc).

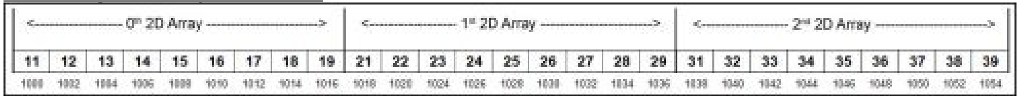
Note: To make this multidimensional array example simple I will discuss 3D array for the sake of simplicity. Once you grab the logic how 3D array works then you can handle 4D array or any multidimensional array easily.

How to Declaration and Initialization 3D Array

Before we move to serious programming let's have a look of 3D array. A 3D array can be assumed as an array of arrays of arrays, it is array (collection) of 2D arrays and as you know 2D array itself is array of 1D array. It sounds a bit confusing but don't worry as you will lead your learning on multidimensional array, you will grasp all logic and concept. A diagram can help you to understand this.



3D Array Conceptual View



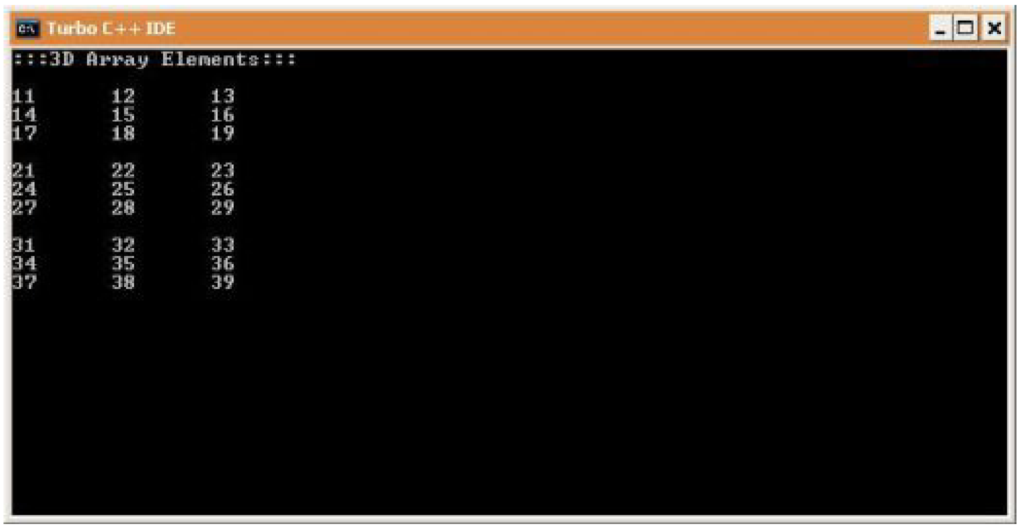
3D array memory map.

We can initialize a 3D array at the compile time as we initialize any other variable or array, by default an un-initialized 3D array contains garbage value. Let’s see a complete example on how we can work with a 3D array.

Example of Declaration and Initialization 3D Array

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

void main()
{
int i, j, k;
int arr[3][3][3]=
        {
            {
            {11, 12, 13},
            {14, 15, 16},
            {17, 18, 19}
            },
            {
            {21, 22, 23},
            {24, 25, 26},
            {27, 28, 29}
            },
            {
            {31, 32, 33},
            {34, 35, 36},
            {37, 38, 39}
            },
        };
clrscr();
printf(":::3D Array Elements:::\n\n");
for(i=0;i<3;i++)
{
    for(j=0;j<3;j++)
    {
        for(k=0;k<3;k++)
        {
        printf("%d\t",arr[i][j][k]);
        }
        printf("\n");
    }
    printf("\n");
}
getch();
}
```



```
:::3D Array Elements:::

11      12      13
14      15      16
17      18      19

21      22      23
24      25      26
27      28      29

31      32      33
34      35      36
37      38      39
```

So in the above example we have declared multidimensional array and named this integer array as "arr" which can hold 3x3x3 (27 integers) elements. We have also initialized multidimensional array with some integer values.
As I told you earlier that a 3D array is array of 2D array therefore I have divided elements accordingly so that you can get 3D array better and understand it easily. See the C code sample above, line no. 9-13, 14-18 and 19-23, each block is a 2D array and collectively from line no. 2-24 makes a 3D array. You can also assign values to this multidimensional array in other way like this.

int arr[3][3][3] = {11, 12, 13, 14, 15, 16, 17, 18, 19, 21, 22, 23, 24, 25, 26, 27, 28, 29, 31, 32, 33, 34, 35, 36, 37, 38, 39};

This kind of C multidimensional array (3D array) declaration is quite confusing for new C programmers; you cannot guess location of array element by just looking at the declaration. But look at the above multidimensional array example where you can get a clear idea about each element location. For example, consider 3D array as a collection of tables, to access or store any element in a 3D array you need to know first table number then row number and lastly column number. For instance you need to access value 25 from above 3D array. So, first check the table (among 3 tables which table has the value), once you find the table number now check which row of that table has the value again if you get the row no then check column number and you will get the value. So applying above logic, 25 located in table no. 1 row no. 1 and column no. 1, hence the address is arr[1][1][1]. Print this address and you will get the output.

So the conceptual syntax for 3D array stands like this.

data_type array_name[table][row][column];

If you want to store values in any 3D array then first point to table number, row number and lastly to column number.

arr[0][1][2] = 32;
arr[1][0][1] = 49;

Above code is for assigning values at particular location of an array but if you want to store value in continuous location of array then you should use loop. Here is an example using for loop.

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

void main()
{
int i, j, k, x=1;
int arr[3][3][3];
clrscr();
printf(":::3D Array Elements:::\n\n");

for(i=0;i<3;i++)
{
    for(j=0;j<3;j++)
    {
        for(k=0;k<3;k++)
        {
        arr[i][j][k] = x;
        printf("%d\t",arr[i][j][k]);
        x++;
        }
        printf("\n");
    }
    printf("\n");
}
getch();
}
```