

UNIT – III

SLO – 1 :

Initializing and Accessing 2D Array

Initialization of 2-d array:

- 2-D array can be initialized in a way similar to that of 1-D array.
- For Example:-

```
int mat[4][3]={11,12,13,14,15,16,17,18,19,20,21,22};
```
- These values are assigned to the elements row wise, so the values of elements after this initialization are

Initialization of 2-d array:

- Mat[0][0]=11, Mat[0][1]=12, Mat[0][2]=13
- Mat[1][0]=14, Mat[1][1]=15, Mat[1][2]=16
- Mat[2][0]=17 Mat[2][1]=18, Mat[2][2]=19
- Mat[3][0]=20 Mat[3][1]=21,, Mat[3][2]=22
- While initializing we can group the elements row wise using inner braces.

For example:-

```
int mat[4][3]={ {11,12,13},{14,15,16},{17,18,19},{20,21,22}};
```

Two dimensional(2D) arrays

- Two dimensional array is known as **Matrix**.
- The array declaration in both the array i.e.in single dimensional array single subscript is used and in two dimensional array **two subscripts** are is used.
- Its syntax is Data-type array **name[row][column]**; Or we can say 2-d array is a collection of 1-D array placed one below the other.

2D Syntax

- Its syntax is

Data-type array name[row][column];

- Or we can say 2-d array is a collection of 1-D array placed one below the other.
- Total no. of elements in 2-D array is calculated as $\text{row} * \text{column}$

Example

int a[2][3];

- Total no of elements=row*column is $2*3 = 6$
- It means the matrix consist of 2 rows and 3 columns

Accessing 2-d array /processing 2-d arrays

- For processing 2-d array, we use two nested for loops.
- The outer for loop corresponds to the row and the inner for loop corresponds to the column.

For example `int a[4][5];`

Example : for reading value:-

```
int a[4][5];
```

```
for(i=0;i<4;i++)
```

```
    i,j
```

```
        0,0 0,1 0,2 0,3 0,4
```

```
        1,0 1,1 1,2 1,3 1,4
```

```
        2,0 2,1 2,2 2,3 2,4
```

```
        3,0 3,1 3,2 3,3 3,4
```

```
{
    for(j=0;j<5;j++)
    {
        scanf("%d",&a[i][j]);
    }
}
```


Example : For displaying value:-

```
for(i=0;i<4;i++)  
{  
    for(j=0;j<5;j++)  
    {  
        printf("%d",a[i][j]);  
    }  
}
```

SLO – 2 :

Initializing Multidimensional Array

Declaration

- Higher dimensional arrays are also supported.
- Declaration of such an array could:
`int a[5][10];`
- Multidimensional arrays are considered as array of arrays.
- Such array are programming abstraction, storage allocation remains same.

Multidimensional Declarations & Initializations

```
#include <stdio.h>
int main() {
    int a[5][10];
    int i, j;

    for (i = 0; i < 5; i++)
        for (j = 0; j < 10; j++)
            a[i][j] = (i+1) * (j+1);
}
```

```
int main() {
    int a[50];
    int i, j;

    for (i = 0; i < 5; i++)
        for (j = 0; j < 10; j++)
            a[i*10+j] = (i+1) * (j+1);
}
```

Initializing a multidimensional array

// Different ways to initialize two-dimensional array

```
int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};  
int c[][3] = {{1, 3, 0}, {-1, 5, 9}};  
int c[2][3] = {1, 3, 0, -1, 5, 9};
```

//3-D Array Initialization

```
int test[2][3][4] = { {{3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2}},  
                      {{13, 4, 56, 3}, {5, 9, 3, 5}, {3, 1, 4, 9}}};
```

SLO – 1 :

Array Programs - 2D

Example 1: Two-dimensional array to store and print values

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

```
int main()
{
    int i , j, temperature [2][7];
    clrscr();
    // Using nested loop to store values in a 2d array
    for (i = 0; i < 2; ++i)
    {
        for (j = 0; j < 7; ++j)
        {
            printf("City %d, Day %d: ", i + 1, j + 1);
            scanf("%d", &temperature[i][j]); } }
    printf("\nDisplaying values: \n\n");
    // Using nested loop to display values of a 2d array
    for (i = 0; i < 2; ++i)
    {
        for (j = 0; j < 7; ++j)
        {
            printf("City %d, Day %d = %d\n", i + 1, j + 1, temperature[i][j]);
        }
    }
    return 0;
}
```

Output

Given Input

City 1, Day 1: 33
City 1, Day 2: 34
City 1, Day 3: 35
City 1, Day 4: 33
City 1, Day 5: 32
City 1, Day 6: 31
City 1, Day 7: 30
City 2, Day 1: 23
City 2, Day 2: 22
City 2, Day 3: 21
City 2, Day 4: 24
City 2, Day 5: 22
City 2, Day 6: 25
City 2, Day 7: 26

Displaying values:

City 1, Day 1 = 33
City 1, Day 2 = 34
City 1, Day 3 = 35
City 1, Day 4 = 33
City 1, Day 5 = 32
City 1, Day 6 = 31
City 1, Day 7 = 30
City 2, Day 1 = 23
City 2, Day 2 = 22
City 2, Day 3 = 21
City 2, Day 4 = 24
City 2, Day 5 = 22
City 2, Day 6 = 25
City 2, Day 7 = 26

Example 2: Sum of two matrices

```
#include <stdio.h>

int main() {
    float a[2][2], b[2][2], result[2][2];
    // Taking input using nested for loop
    printf("Enter elements of 1st matrix\n");
    for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 2; ++j) {
            printf("Enter a%d%d: ", i + 1, j + 1);
            scanf("%f", &a[i][j]); }
    printf("Enter elements of 2nd matrix\n");
    for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 2; ++j) {
            printf("Enter b%d%d: ", i + 1, j + 1);
            scanf("%f", &b[i][j]); }
```

Example Continue

```
// adding corresponding elements of two arrays
for (int i = 0; i < 2; ++i)
    for (int j = 0; j < 2; ++j) {
        result[i][j] = a[i][j] + b[i][j]; }
// Displaying the sum
printf("\nSum Of Matrix:");
for (int i = 0; i < 2; ++i)
    for (int j = 0; j < 2; ++j)
    {
        printf("%.1f\t", result[i][j]);
        if (j == 1)
            printf("\n");
    }
return 0; }
```

Matrix Addition Output

Enter elements of 1st matrix

Enter a11: 2;

Enter a12: 0.5;

Enter a21: -1.1;

Enter a22: 2;

Enter elements of 2nd matrix

Enter b11: 0.2;

Enter b12: 0;

Enter b21: 0.23;

Enter b22: 23;

Sum Of Matrix:

2.2 0.5

-0.9 25.0

SLO – 2 :

Array Contiguous Memory

Array Contiguous Memory

- When Big Block of memory is reserved or allocated then that memory block is called as Contiguous Memory Block.
- Alternate meaning of Contiguous Memory is continuous memory.
- Suppose inside memory we have reserved 1000-1200 memory addresses for special purposes then we can say that these 200 blocks are going to reserve contiguous memory.

How to allocate contiguous memory

- Using static array declaration.
- Using `alloc()` / `malloc()` function to allocate big chunk of memory dynamically.

Contiguous Memory Allocation

- Two registers are used while implementing the contiguous memory scheme. These registers are base register and limit register.

How to allocate contiguous memory...

- When OS is executing a process inside the main memory then content of each register are as –

Register	Content of Register
Base register	Starting address of the memory location where process execution is happening
Limit register	Total amount of memory in bytes consumed by process

How to allocate contiguous memory...

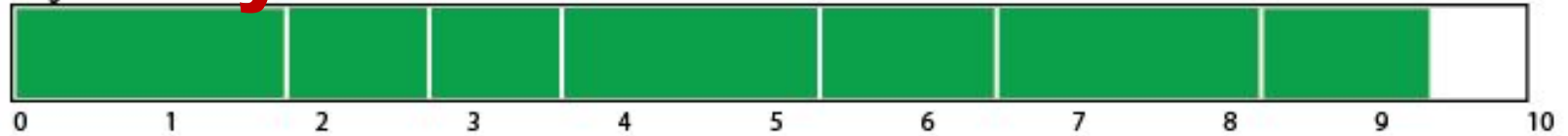
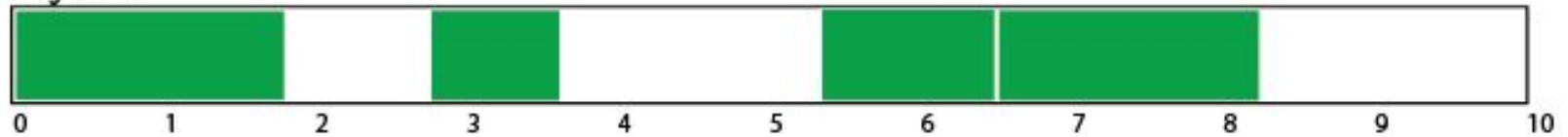


Diagram 2



- Here diagram 1 represents the **contiguous allocation** of memory and diagram 2 represents **non-contiguous allocation** of memory.
- ✓ When process try to refer a part of the memory then it will firstly refer the base address from base register and then it will refer relative address of memory location with respect to base address

SLO – 1 :
Array Advantages and
Limitations

Advantage and Limitations of Array

Advantages:

- It is better and convenient way of storing the data of **same datatype** with same size.
- It allows us to store **known number of elements** in it.
- It allocates memory in **contiguous memory** locations for its elements. It does not allocate any extra space/ memory for its elements. Hence there is **no memory overflow or shortage** of memory in arrays.
- Iterating the arrays using their index is **faster** compared to any other methods like linked list etc.
- It allows to **store the elements in any dimensional** array - supports multidimensional array.

Limitations of Array:

A) *Static Data*

- Array is Static data Structure
- Memory Allocated during Compile time.
- Once Memory is allocated at Compile Time it Cannot be Changed during Run-time.

B) *Can hold data belonging to same Data types*

- Elements belonging to different data types cannot be stored in array because array data structure can hold data belonging to same data type.
- Example : Character and Integer values can be stored inside separate array but cannot be stored in single array

Limitations of Array:

C) *Inserting data in Array is Difficult*

- Inserting element is very difficult because before inserting element in an array have to create empty space by shifting other elements one position ahead.
- This operation is faster if the array size is smaller, but same operation will be more and more time consuming and non- efficient in case of array with large size.

D) *Deletion Operation is difficult*

- Deletion is not easy because the elements are stored in contiguous memory location.
- Like insertion operation , we have to delete element from the array and after deletion empty space will be created and thus we need to fill the space by moving elements up in the array.

Limitations of Array:

E) Bound Checking

- If we specify the size of array as 'N' then we can access elements upto 'N-1' but in C if we try to access elements after 'N-1' i.e Nth element or N+1th element then we does not get any error message.
- Process of Checking the extreme limit of array is called *Bound checking*
- and C does not perform Bound Checking.
- If the array range exceeds then we will **get garbage value** as result.

F) Shortage of Memory

- Array is Static data structure. Memory can be **allocated at compile time** only. Thus if after executing program we need more space for storing additional information then we **cannot allocate additional space at run time**.
- Shortage of Memory , if we don't know the size of memory in advance

G) Wastage of Memory

- Wastage of Memory , if array of large size is defined.

SLO – 2 :
**Array Construction for Real-time Application,
Common Programming Errors**

Array Construction for Real-time Application

(i) Stores Elements of Same Data Type

- Array is used to store the number of elements belonging to same data type.

```
int arr[30];
```

Above array is used to store the integer numbers in an array.

```
arr[0] = 10;
```

```
arr[1] = 20;
```

```
arr[2] = 30;
```

```
arr[3] = 40;
```

```
arr[4] = 50;
```

```
-
```

```
-
```

- Similarly if we declare the **character array** then it can hold only character. So in short character array can store character variables while floating array stores only floating numbers.

Array Construction for Real-time Application...

(ii) Array Used for Maintaining multiple variable names using single name

- Suppose we need to store 5 roll numbers of students then without declaration of array we need to declare following:

int roll1,roll2,roll3,roll4,roll5;

- Now in order to get roll number of first student we need to access roll1.
- Guess if we need to store roll numbers of 100 students then what will be the procedure.
- Maintaining all the variables and remembering all these things is very difficult.

Array Construction for Real-time Application...

(iii) Array Can be Used for Sorting Elements

- We can store elements to be sorted in an array and then by using different sorting technique we can sort the elements.

Different Sorting Techniques are :

- Bubble Sort
- Insertion Sort
- Selection Sort
- Bucket Sort

(iv) Array Can Perform Matrix Operation

- Matrix operations can be performed using the array. We can use 2-D array to store the matrix.

Array Construction for Real-time Application...

(v) Array Can be Used in CPU Scheduling

- CPU Scheduling is generally managed by Queue. Queue can be managed and implemented using the array. Array may be allocated dynamically i.e at run time.

(vi) Array Can be Used in Recursive Function

- When the function calls another function or the same function again then the current values are stores onto the stack and those values will be retrieve when control comes back. This is similar operation like stack.

Common Programming Errors

(i) Constant Expression Require

```
#include<stdio.h>
void main()
{
int i=10;

int a[i];
}
```

In this example we see what's that error?

Common Programming Errors...

- We are going to declare an array whose size is equal to the value of variable.
- If we changed the value of variable then array size is going to change.
- According to array concept, we are *allocating memory for array at compile time* so if the size of array is going to vary then how it is possible to allocate memory to an array.
- i is initialized to 10 and using a[i] does not mean a[10] because 'i' is Integer Variable whose value can be changed inside program.

Common Programming Errors...

Value of Const Variable Cannot be changed

- we know that value of const Variable cannot be changed once initialized so we can write above example as below –

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

```
void main()
```

```
{
```

```
const int i=10; int a[i];
```

```
}
```

or

```
int a[10];
```

Common Programming Errors...

(ii) Empty Valued 1D Array

```
#include<stdio.h>
void main()
{
int arr[];
}
```

Instead of it Write it as –

```
#include<stdio.h>
void main()
{
int a[] = {1,1};
}
```

- Consider the above example, we can see the empty pair of square brackets means we haven't specified size of an 1D Array. In the example array 'arr' is undefined or empty.
- Size of 1D Array should be Specified as a Constant Value.

```
#include<stdio.h>
void main()
{
int a[] = {};
// This also Cause an Error
}
```

Common Programming Errors...

(iii) 1D Array with no
Bound checking

```
#include<stdio.h> void main()
{
int a[5];
printf("%d",a[7]);
}
```

- Here Array size specified is 5.
- So we have Access to Following Array Elements – a[0],a[1],a[2],a[3] and a[4]
- But accessing a[6] causes Garbage Value to be used because C Does not performs Array Bound Check.

If the maximum size of array is “MAX” then we can access following elements of an array – Elements accessible for Array Size "MAX" = arr[0]
= arr[MAX-1]

Common Programming Errors...

(iv) Case Sensitive

```
#include<stdio.h>
void main()
{
int a[5];
printf("%d",A[2]);
}
```

Array Variable is Case Sensitive so A[2] does not print anything it Displays

Error Message : “Undefined Symbol A”

SLO – 1 :

String Basics

String Basics

- A string in C is simply an array of characters. The following line declares an array that can hold a string of up to 99 characters.

char str[100];

- It holds characters as you would expect:
 - str[0] is the first character of the string,
 - str[1] is the second character, and so on.
- But why is a 100-element array unable to hold up to 100 characters?
- Because C uses null-terminated strings, which means that the end of any string is marked by the ASCII value 0 (the null character), which is also represented in C as '\0'.

What is NULL Char “\0”?

'\0' represents the end of the string. It is also referred as String terminator & Null Character.

String Basics

- Pascal can simply return the length byte, whereas C has to **count the characters until it finds '\0'**. This fact makes C much slower than Pascal in certain cases, but in others it makes it faster.
- Because C provides **no explicit support** for strings in the language itself, all of the string-handling functions are **implemented in libraries**.
- The string I/O operations (gets, puts, and so on) are implemented in **<stdio.h>**, and a set of fairly simple string manipulation functions are implemented in **<string.h>** (on some systems, **<strings.h>**)

String Basics

- The fact that strings are **not native to C** forces you to create some fairly round about code.
- For example, suppose you want to assign one string to another string; that is, you want to copy the contents of one string to another.
- **You cannot simply assign one array to another.** You have to **copy it element by element.**
- The string library (`<string.h>` or `<strings.h>`) contains a function called `strcpy` for this task.

SLO – 2 :

String Declaration and Initialization

String Declaration and Initialization

String Declaration

1) `char str1[]={ 'A', 'B', 'C', 'D', '\0'};`

2) `char str1[]="ABCD";`



'\0' would automatically
inserted at the end in this
type of declaration

Method 1:

`char address[]={ 'S', 'R', 'M', 'I', 'S', 'T', '\0'};`

Method 2: The above string can also be defined as –

`char address[]="SRMIST";`

In the above declaration NULL character (\0) will automatically be inserted at the end of the string.

SLO – 1 :
Strings functions :
gets()
puts()
getchar()
putchar()
scanf()
printf()

C – String functions

strlen - Finds out the length of a string

strlwr - It converts a string to lowercase

strupr - It converts a string to uppercase

strcat - It appends one string at the end of another

strncat - It appends first n characters of a string at the end of another.

strcpy - Use it for Copying a string into another

strncpy - It copies first n characters of one string into another

strcmp - It compares two strings

strncmp - It compares first n characters of two strings

strcmpi - It compares two strings without regard to case ("i" denotes that this function ignores case)

stricmp - It compares two strings without regard to case (identical to strcmpi)

strnicmp - It compares first n characters of two strings, Its not case sensitive

strdup - Used for Duplicating a string

strchr - Finds out first occurrence of a given character in a string

strrchr - Finds out last occurrence of a given character in a string

strstr - Finds first occurrence of a given string in another string

strset - It sets all characters of string to a given character

strnset - It sets first n characters of a string to a given character

strrev - It Reverses a string

String I/O :

gets()

puts()

String I/O

1) printf and scanf

2) puts and gets

Syntax of above functions - Assume string as str1

```
printf("%s", str1);
```

```
puts(str1); --%s not require here.
```

```
scanf("%s", &str1);
```

```
gets(str1); --%s not require
```

Read & Write Strings in C using *gets()* and *puts()* functions

```
#include <stdio.h>
#include <string.h>
int main()
{
    /* String Declaration*/
    char nickname[20];
    /* Console display using puts */
    puts("Enter your Nick name:");
    /*Input using gets*/
    gets(nickname);
    puts(nickname);
    return 0;
}
```

OUTPUT



```
Enter your Nick name:
SIVAS
SIVAS
```

Read & Write character in C using *getchar()* and *putchar()* functions

- When we need to read or write data character by character, functions *getchar* and *putchar* are very handy.
- To read a single character in variable *c*, we give:

c = getchar();
(equivalent to *scanf("%c", &c);*)

- To write the character in variable *c*, we give:

putchar(c);
(equivalent to *printf("%c", c);*)

Read & write Strings in C using *printf()* and *scanf()* functions

```
#include <stdio.h>
#include <string.h>
int main()
{
```

Note: %s format specifier is used for strings input/output

```
    /* String Declaration*/
```

```
    char nickname[20];
```

```
    clrscr();
```

```
    printf("Enter your Nick name:");
```

```
    /* I am reading the input string and storing it in nickname
```

```
    * Array name alone works as a base address of array so
```

```
    * we can use nickname instead of &nickname here
```

```
    */
```

```
    scanf("%s", nickname);
```

```
    /*Displaying String*/
```

```
    printf("%s",nickname);
```

```
    getch();
```

```
    return 0;
```

```
}
```

OUTPUT



```
Enter your Nick name:RAMKUMAR
RAMKUMAR_
```

SLO – 2 :
Strings functions :
atoi()
strlen()
strcat()
strcmp()

C String function – atoi

(Convert String to Integer)

- In the C Programming Language, the atoi function converts a string to an integer.
- The atoi function skips all white-space characters at the beginning of the string, converts the subsequent characters as part of the number, and then stops when it encounters the first character that isn't a number.
- The return value is 0 if the function cannot convert the input to a value of that type.

Syntax

The syntax for the atoi function in the C Language is:

```
int atoi(const char *nptr);
```

C String function – atoi

Parameters or Arguments

nptr

- A pointer to a string to convert to an integer.

Returns

- The atoi function returns the integer representation of a string.
- The atoi function **skips all white-space characters** at the beginning of the string, converts the subsequent characters as part of the number, and then stops when it encounters the first character that isn't a number.

Required Header

In the C Language, the required header for the atoi function is:

`#include <stdlib.h>`

Example shows how to convert numbers that are stored as strings to numeric values.

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
    int i;
    char *s;
    clrscr();
    s = "-9885";
    i = atoi(s);    /* i = -9885 */
    getch();
    printf("i = %d\n",i);
}
```

OUTPUT

A black rectangular box with white text inside, representing a terminal window output. The text inside is "i = -9885".

i = -9885

C String function – strlen

Syntax:

size_t strlen(const char *str)

size_t represents unsigned short

- It returns the length of the string without including end character (terminating char '\0').

Example of strlen:

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

```
#include <string.h>
```

```
int main()
```

```
{
```

```
    char str1[20] = "SRMUNIVERSITY";
```

```
    clrscr();
```

```
    printf("Length of string str1: %d", strlen(str1));
```

```
    getch();
```

```
    return 0;
```

```
}
```

OUTPUT

```
Length of string str1: 13
```

strlen vs sizeof

- strlen returns you the **length of the string** stored in array, however sizeof returns the **total allocated size** assigned to the array.
- So if I consider the above example again then the following statements would return the below values.
- strlen(str1) returned value 13.
- sizeof(str1) would return value 20 as the array size is 20 (see the first statement in main function).

C String function – strlen

Syntax:

size_t strlen(const char *str, size_t maxlen)

size_t represents unsigned short

- It returns length of the string if it is less than the value specified for maxlen (maximum length) otherwise it returns maxlen value.

Example of strlen:

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

```
#include <string.h>
```

```
int main()
```

```
{
```

```
    char str1[20] = "BeginnersBook";
```

```
    clrscr();
```

```
    printf("Length of string str1 when maxlen is 30: %d", strlen(str1, 30));
```

```
    printf("Length of string str1 when maxlen is 10: %d", strlen(str1, 10));
```

```
    getch();
```

```
    return 0;
```

```
}
```

C String function – strcat

`char *strcat(char *str1, char *str2)`

- It concatenates two strings and returns the concatenated string.

Example of strcat:

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

```
#include <string.h>
```

```
int main()
```

```
{
```

```
    char s1[10] = "SRM";
```

```
    char s2[15] = "UNIVERSITY";
```

```
    clrscr();
```

```
    strcat(s1,s2);
```

```
    printf("Output string after concatenation: %s", s1);
```

```
    getch();
```

```
    return 0;
```

```
}
```

OUTPUT



```
Output string after concatenation: SRMUNIVERSITY
```

C String function – strncat

char *strncat(char *str1, char *str2, int n)

- It concatenates n characters of str2 to string str1. A terminator char ('\0') will always be appended at the end of the concatenated string.
- n: represents maximum number of character to be appended. size_t is an unsigned integral type.

Example of strncat:

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

```
#include <string.h>
```

```
int main()
```

```
{
```

```
    char s1[10] = "SRM";
```

```
    char s2[10] = "UNIVERSITY";
```

```
    clrscr();
```

```
    strncat(s1,s2, 3);
```

```
    printf("Concatenation using strncat: %s", s1);
```

```
    getch();    return 0; }
```

OUTPUT

```
Concatenation using strncat: SRMUNI
```

ASCII

- ASCII abbreviated from **American Standard Code for Information Interchange**, is a character encoding standard for electronic communication.
- ASCII codes represent text in computers, telecommunications equipment, and other devices.
- Most modern character-encoding schemes are based on ASCII

Example : ASCII

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

```
int main()  
{  
    char c;  
    clrscr();  
    printf("Enter a character : ");  
    scanf("%c" , &c);  
    printf("\n\nASCII value of %c = %d",c,c);  
    getch();  
    return 0;  
}
```


C String function – strcmp

int strcmp(const char *str1, const char *str2)

- It compares the two strings and returns an integer value.
- If both the strings are same (equal) then this function would return 0 otherwise it may return a negative or positive value based on the comparison.
- If string1 < string2 OR string1 is a substring of string2 then it would result in a negative value.
- If string1 > string2 then it would return positive value.
- If string1 == string2 then you would get 0(zero) when you use this function for compare strings.

C String function – strcmp...

- This function compares strings character by character using ASCII value of the characters.
- The comparison stops when either end of the string is reached or corresponding characters are not same.
- The non-zero value returned on mismatch is the difference of the ASCII values of the non-matching characters of two strings.

Example : strcmp

- Let's see how strcmp() function compare strings using an example.

```
strcmp("jkl", "jkq");
```

- Here we have two strings str1 = "jkl" and str2 = "jkq".
- Comparison starts off by comparing the first character from str1 and str2 i.e 'j' from "jkl" and 'j' from "jkq", as they are equal,
- the next two characters are compared i.e 'k' from "jkl" and 'k' from "jkq", as they are also equal,
- again the next two characters are compared i.e 'l' from "jkl" and 'q' from "jkq", as ASCII value of 'q' (113) is greater than that of 'l' (108),
- Therefore str2 is greater than str1 and strcmp() will return -5 (i.e $108 - 113 = -5$).

C String function – strcmp...

- It is important to note that not all systems return difference of the ASCII value of characters,
- On some systems if str1 is greater than str2 then 1 is returned.
- On the other hand, if str1 is smaller than str2 then -1 is returned.
- It is more likely that you will encounter this behaviour on your system.

Example : strcmp

```
#include<stdio.h>
#include<string.h>
void main()
{
clrscr();
printf("(a, a) result : %d\n",strcmp("a", "a")); // returns 0 as ASCII value of "a" and "a" are same
i.e 97

printf("(a, b) result :%d\n",strcmp("a", "b")); // returns -1 as ASCII value of "a" (97) is less than
"b" (98)

printf("(a, b) result :%d\n",strcmp("a", "c")); // returns -1 as ASCII value of "a" (97) is less than "c"
(99)

printf("(z, b) result :%d\n",strcmp("z", "d")); // returns 1 as ASCII value of "z" (122) is greater than
"d" (100)

printf("(abc, abe) result :%d\n",strcmp("abc", "abe")); // returns -1 as ASCII value of "c" (99) is
less than "e" (101)

printf("(apples, apple) result :%d\n",strcmp("apples", "apple")); // returns 1 as ASCII value of "s"
(115) is greater than "\0" (101)
getch();
}
```

C String function – strcmp

Example of strcmp:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s1[20] = "SRMUNIVERSITY";
    char s2[20] = "SRMUNIVERSITY.COM";
    clrscr();
    if (strcmp(s1, s2) == 0)
    {
        printf("string 1 and string 2 are equal");
    }
    else
    {
        printf("string 1 and 2 are different");
    }
    getch();
    return 0;
}
```

OUTPUT

string 1 and 2 are different_

C String function – strcmp

```
#include<stdio.h>
#include<string.h>
int main()
{
    char strg1[50], strg2[50];
    clrscr();
    printf("Enter first string: ");
    gets(strg1);
    printf("Enter second string: ");
    gets(strg2);
    /* if (strcmp(strg1,strg2) >=1)
        printf("+ve");
        else
            printf("-ve");
    */
    if(strcmp(strg1, strg2)==0)
    {
        printf("\nYou entered the same string two times");
    }

    else
    {
        printf("\nEntered strings are not same!");
    }
    getch(); return 0; }
```

SLO – 1 :
Strings functions :
sprintf
sscanf
strrev
strcpy
strstr
strtok

sprintf () Function

- The C library function sprintf () is used to store formatted data as a string.
- You can also say the sprintf () function is used to create strings as output using formatted data.

Syntax

```
int sprintf (char *string, const char *form, ... );
```

- The function sprintf () is included in the standard input output library **stdio.h**
- Here, the ***string** will stand for the name of the array that will store the output obtained by working on the formatted data.


sprintf () Function

- The ***form** parameter will show the **format of the output**. Outputs can be obtained in different formats, like in the list below:
- **%d**: You will obtain an integer output.
- **%f**: You will obtain a floating-point number in a fixed decimal form as your output.
- **%e**: You will obtain a floating-point number that follows the exponential (scientific) format.
 - For example: 5.050000e+01
- **%.1f**: You will obtain a floating-point number with just a single digit displayed after the decimal point.

Examples of the sprintf () Function

```
#include<stdio.h>
#include<string.h>
int main()
{
    int sal;
    char name[30], designation[30], info[60];
    clrscr();
    printf("Enter your name: ");
    gets(name);
    printf("Enter your designation: ");
    gets(designation);
    printf("Enter your salary: ");
    scanf("%d", &sal);
    sprintf(info, "Welcome %s !\nName: %s \nDesignation: %s\nSalary: %d",
        name, name, designation, sal);
    printf("\n%s", info);
    puts(info);
    getch();
    return 0; }
```

OUTPUT

A screenshot of a terminal window with a black background and white text. The text shows the execution of a C program. It starts with three prompts: 'Enter your name: MAIN', 'Enter your designation: DATA ENTRY OPERATOR', and 'Enter your salary: 10000'. Then, it displays the output of the sprintf function: 'Welcome MAIN !', 'Name: MAIN', 'Designation: DATA ENTRY OPERATOR', and 'Salary: 10000'. The output is formatted with line breaks and spaces, matching the format string in the code. A small horizontal line is visible at the bottom of the terminal window.

```
Enter your name: MAIN
Enter your designation: DATA ENTRY OPERATOR
Enter your salary: 10000

Welcome MAIN !
Name: MAIN
Designation: DATA ENTRY OPERATOR
Salary: 10000Welcome MAIN !
Name: MAIN
Designation: DATA ENTRY OPERATOR
Salary: 10000
_
```

sscanf() Function

- The sscanf() function allows us to read formatted data from a string rather than standard input or keyboard.

Syntax:

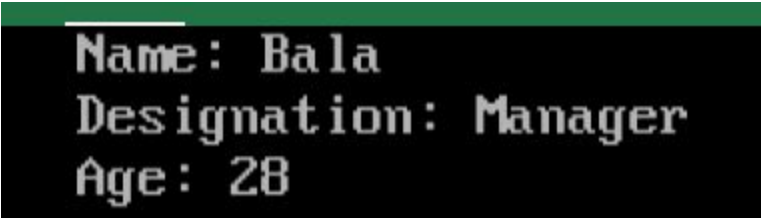
```
int sscanf(const char *str, const char * control_string [
arg_1, arg_2, ... ]);
```

- The first argument is a pointer to the string from where we want to read the data.
- The rest of the arguments of sscanf() is same as that of scanf().
- It returns the number of items read from the string and -1 if an error is encountered.

Example of the sscanf() Function

```
#include<stdio.h>
#include<string.h>
int main()
{
    char *str = "Bala Manager 28";
    char name[10], designation[10];
    int age, ret;
    clrscr();
    ret = sscanf(str, "%s %s %d", name, designation, &age);
    printf("Name: %s\n", name);
    printf("Designation: %s\n", designation);
    printf("Age: %d\n", age);
    getch();
    return 0;
}
```

OUTPUT

A screenshot of a terminal window with a black background and green text. It shows the output of the C program: 'Name: Bala', 'Designation: Manager', and 'Age: 28' on three separate lines.

```
Name: Bala
Designation: Manager
Age: 28
```

How it works:

- In line 5, we have declared and initialized a variable str of type pointer to char.
- In line 6, we have declared two arrays of character name and designation of size 10 characters.
- In line 7, we have declared variable age of type int.
- In line 9, scanf() function is called to read the data from the string pointed to by str.
- Notice that the string literal "Bala Manager 28" contains three pieces of information name, designation and age separated by space.
- To read all the three items we need to supply three variables of appropriate type to the scanf() function.

How it works:

- Then variable ret is assigned the number of items read by sscanf() function. In this case, we are reading three items from the string str, so 3 will be assigned to ret.
- We are not obliged to read all the items in the string literal, if we want we can also read one or two **items** from it.

ret = sscanf(str, "%s %s", name, designation);

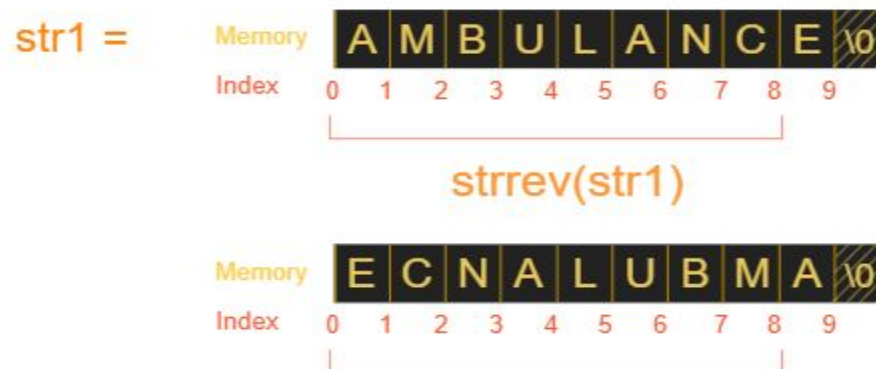
- Here we are only reading name and designation that's why **only two variables** are provided to sscanf().
- At last, the printf() function is used to display name, designation, age and ret.

strrev() Function

- **strrev()** is one of the inbuilt string function in c programming which is used to reverse a given string.

How strrev() Works

- The following diagram clearly illustrate the working principle of **strrev()** inbuilt string function in C.



- In the above diagram `strrev()` takes one parameter which is a string. `strrev()` will reverse the given string.

Syntax - strrev()

- **strrev()** accept single parameter.
- Parameter must be a string.
- To use **strrev()** inbuilt string function in C, we need to declare `#include<string.h>` header file.

Syntax

```
strrev(str1);
```

Example of the strrev() Function

```
#include <stdio.h>
#include <string.h>
int main()
{
    char str1[20] = "AMBULANCE";
    clrscr();
    printf(" %s ", strrev(str1));
    getch();
    return 0;
}
```

OUTPUT

The output of the program is the string "ECNALUBMA", which is the reverse of the input string "AMBULANCE". The text is displayed in a white, monospaced font on a black rectangular background.

Without strrev()

Let us reverse the string without using inbuilt string function strrev()

```
#include <stdio.h>
#include <string.h>
int main()
{
char str[20] = "AMBULANCE";
char str2[20];
int i, j, k;
clrscr();
for(i=0; str[i]!='\0'; i++){
//finding length of a string
}
k=i-1;
for(j=0;j<i;j++){
str2[j]=str[k];
k--;
}
for(i=0; str[i]!='\0'; i++){
printf("%c",str2[i]);
}
getch();
return 0;
}
```

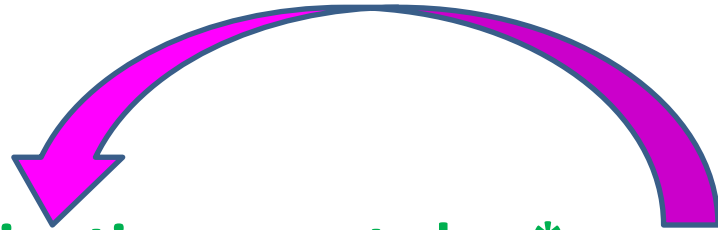
OUTPUT



strcpy() Function in C

- The strcpy() function is used to copy strings.

Syntax:



char* strcpy (char* destination, const char* source);

- The strcpy() function is used to copy strings. It copies string pointed to by source into the destination.
- This function accepts two arguments of type pointer to char or array of characters and returns a pointer to the first string i.e destination.
- Notice that source is preceded by the const modifier because strcpy() function is not allowed to change the source string.

Example of the strcpy() Function

```
#include<stdio.h>
#include<string.h>
int main()
{
    char ch_arr1[20];
    char ch_arr2[20];
    clrscr();
    printf("Enter first string (ch_arr_1): ");
    gets(ch_arr1);

    printf("Enter second string(ch_arr_1): ");
    gets(ch_arr2);
    printf("\nCopying first string into second... \n\n");
    strcpy(ch_arr2, ch_arr1); // copy the contents of ch_arr1 to ch_arr2
```

Example of the strcpy() Function

```
printf("First string (ch_arr_1) = %s\n", ch_arr1);  
printf("Second string (ch_arr_2) = %s\n", ch_arr2);
```

```
printf("\nCopying \"TAMILNADU\" string into ch_arr1 ... \n\n");  
strcpy(ch_arr1, "TAMILNADU"); // copy TAMILNADU to ch_arr1
```

```
printf("\nCopying \"KERALA\" string into ch_arr2 ... \n\n");  
strcpy(ch_arr2, "KERALA"); // copy KERALA to ch_arr2
```

```
printf("First string (ch_arr_1) = %s\n", ch_arr1);  
printf("Second string (ch_arr_2) = %s\n", ch_arr2);
```

OUTPUT

```
getch();  
return 0;  
}
```

```
Enter first string (ch_arr_1): SRM  
Enter second string(ch_arr_1): UNIVERSIY  
  
Copying first string into second...  
  
First string (ch_arr_1) = SRM  
Second string (ch_arr_2) = SRM  
  
Copying "TAMILNADU" string into ch_arr1 ...  
  
Copying "KERALA" string into ch_arr2 ...  
  
First string (ch_arr_1) = TAMILNADU  
Second string (ch_arr_2) = KERALA
```

Note : strcpy() function

- Another important point to note about strcpy() is that you should **never pass string literals** as a first argument. For example:

```
char ch_arr[] = "string array";
```

```
strcpy("destination string", c_arr); // wrong
```

- Here you are trying to copy the contents of ch_arr to "destination string" which is a string literal.
- Since modifying a string literal **causes undefined behaviour**, calling strcpy() in this way may cause the program to crash.

strstr() function

- strstr() is the built in standard string manipulation function that is defined under string handling library string.h.
- Therefore it is necessary to include string header file to use standard C string library function strstr().

```
#include<string.h>
```


Syntax : strstr() function

```
char *strstr( const char *str1, const char *str2 );
```

where,

str1 = string that needs to be scanned

str2 = small string which characters needs to be located in str1

strstr() function

- This function identifies all the characters of small string str2 in the large string str1.
- If all the characters from string str2 are located, a pointer to the string in str1 is returned, otherwise, a **NULL pointer** is returned.
- Use: This function is useful when you want to locate the **first occurrence of a small string in a huge text.**

Example of the strstr() Function

```
/*Use of C string library function strstr*/
#include<stdio.h>
#include<string.h>
int main()
{
    //initializing character pointer
    const char *str1 = "SRM Institute of Science and Technology";
    const char *str2 = "Sci";
    clrscr();
    //displaying both string
    printf("str1 = %s\n\n", str1);
    printf("str2 = %s\n\n", str2);
    printf("Remaining part of str1 after the first"
           "occurence of str2 = %s\n", strstr(str1, str2));
    getch();
    return 0;
} //end
```

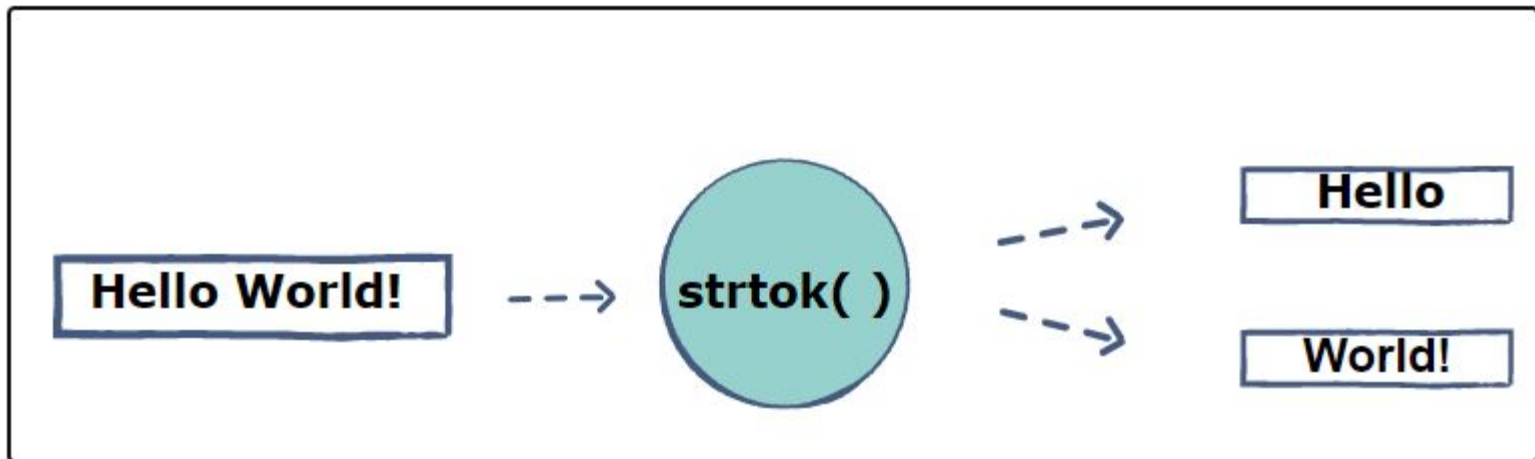
OUTPUT

```
str1 = SRM Institute of Science and Technology
str2 = Sci

Remaining part of str1 after the first occurrence of str2 = Science and Technolog
y
-
```

Splitting a string using strtok() in C

- In C, the strtok() function is used to split a string into a series of tokens based on a particular delimiter. A token is a substring extracted from the original string.



Syntax

- The general syntax for the strtok() function is:
`char *strtok(char *str, const char *delim)`

Parameters

- Let's look at the parameters the strtok() function takes as input:
 - `str`: The string which is to be split
 - `delim`: The character on the basis of which the split will be done

Return value

- The function performs one split and returns a pointer to the token split up. A null pointer is returned if the string cannot be split.

Extracting a single token

- The following piece of code will just split up the Hello world string in two and will return the first token extracted from the function.

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

```
int main() {
    char string[50] = "Hello world";
    // Extract the first token
    char * token = strtok(string, " ");
    printf( " %s\n", token ); //printing the token
    return 0;
}
```

Extracting all possible tokens

- To find all possible splits of the string, based on a **given delimiter**, the function needs to be called in a loop. See the example below to see how this works.
- Let's see how we can split up a sentence on each occurrence of the white space character:

Extracting all possible tokens

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

int main() {
    char string[50] = "Hello! We are learning about strtok";
    // Extract the first token
    char * token = strtok(string, " ");
    // loop through the string to extract all other tokens
    while( token != NULL ) {
        printf( " %s\n", token ); //printing each token
        token = strtok(NULL, " ");
    }
    return 0;
}
```

SLO – 2 :
Arithmetic characters on Strings

Arithmetic characters on Strings

- .C Programming Allows you to Manipulate Strings
- . Whenever arithmetic operations are performed on characters it is automatically Converted into Integer Value called
 - ASCII value.
- . .All Characters can be Manipulated with that Integer
 - Value.(Addition,Subtraction)

Examples :

ASCII value of : 'a' is 97

ASCII value of : 'z' is 121

Possible Ways of Manipulation

Way 1:Displays ASCII value[Note that %d in Printf]

```
char x = 'a';  
printf("%d",x); // Display Result = 97
```

Way 2 :Displays Character value[Note that %c in Printf]

```
char x = 'a';  
printf("%c",x); // Display Result = a
```

Way 3 : Displays Next ASCII value[Note that %d in Printf
]

```
char x = 'a' + 1 ;  
printf("%d",x); //Display Result = 98 (ascii of 'b' )
```

Possible Ways of Manipulation

Way 4 : Displays Next Character value[Note that %c in Printf]

```
char x = 'a' + 1;  
printf("%c",x);          // Display Result = 'b'
```

Way 5 : Displays Difference between 2 ASCII in Integer[Note %d in Printf]

```
char x = 'z' - 'a';  
printf("%d",x);  
/*Display Result = 25 (difference between ASCII of z and a ) */
```

Way 6 : Displays Difference between 2 ASCII in Char [Note that %c in Printf]

```
char x = 'z' - 'a';  
printf("%c",x);  
/*Display Result =( difference between ASCII of z and a ) */
```

Example

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

```
int main(){  
    char s = 'm';  
    char t = 'z' - 'y';  
  
    printf("%d\n", s);  
    printf("%c\n", s);  
    printf("%d\n", (s+1));  
    printf("%c\n", (s+1));  
    printf("%d\n", (s-1));  
    printf("%c\n", (s-1));  
    printf("%d\n", t);  
  
    return 0;  
}
```

SLO – 1 :
**Functions declaration and
definition**

Lesson : Functions declaration and definition

Upon completion of this lesson, you will be able to:

- **Define Function**
- **Describe Modular Programming**
- **Function Types**
- **Types of Arguments /Parameters**
- **Types of function definition**

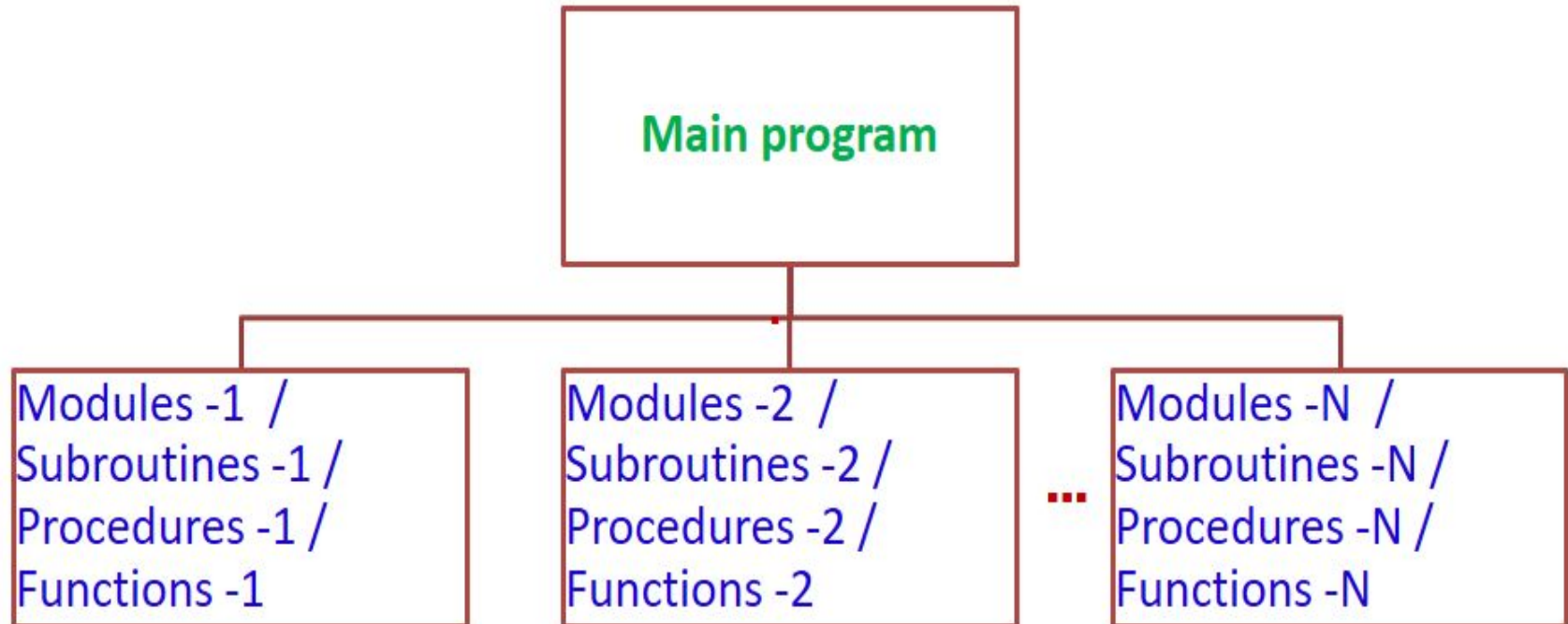
Defining a Function

Definition :

- A function is an independent part of the program, and can perform a specific tasks
(or)
- Function is basically set of statements that takes input, perform some computation and produces result.

Modular Programming

The process of subdividing main program into smaller separate functions is called modular programming



Example : Railway Reservation System □ Ticket Booking, Payment, Enquiry, Ticket Cancellation etc.,

Modular Programming

- Break a complex problem into smaller pieces
 - Smaller pieces sometimes called 'modules' or 'subroutines' or 'procedures' or functions
 - Why?
 - Helps manage complexity
 - Smaller blocks of code
 - Quality - improves the quality of a specific piece of code
 - Easier to read Encourages re-use of code
 - Within a particular program or across different programs
 - Allows independent development of code
 - Provides a layer of 'abstraction' **toupper()**

Function Types

- A function is a self-contained block of statements that perform coherent task of some kind.
- Every C program can be thought of as a collection of these functions.

Two types of functions,

1. Standard library functions
2. User-defined functions

Standard Library functions

- The functions which are in-built with the C compiler are called standard library functions.

Examples:

printf()

scanf()

sqrt()

getch()

clrscr()

User-defined Functions

- User can define functions to do a task relevant to their programs. Such functions are called user-defined functions.
- The `main()` function in which we write all our program is a user-defined function.
- Every program execution starts at the `main()` function.
Note : Any function (library or user-defined) has 3 things

1. **Function declaration**
2. **Function calling**
3. **Function definition**

- In case of library functions, the function declaration is in header files.
- But in case of user-defined functions all the 3 things are in your source program.

Function declaration

Syntax :

```
return data_type function_name(arguments list);
```

- **Function_name** -> valid user defined identifier
- **return data_type**-> data_type of the result returned to the caller
- **(argument list)**->list of parameters passed to the function

Function Definition

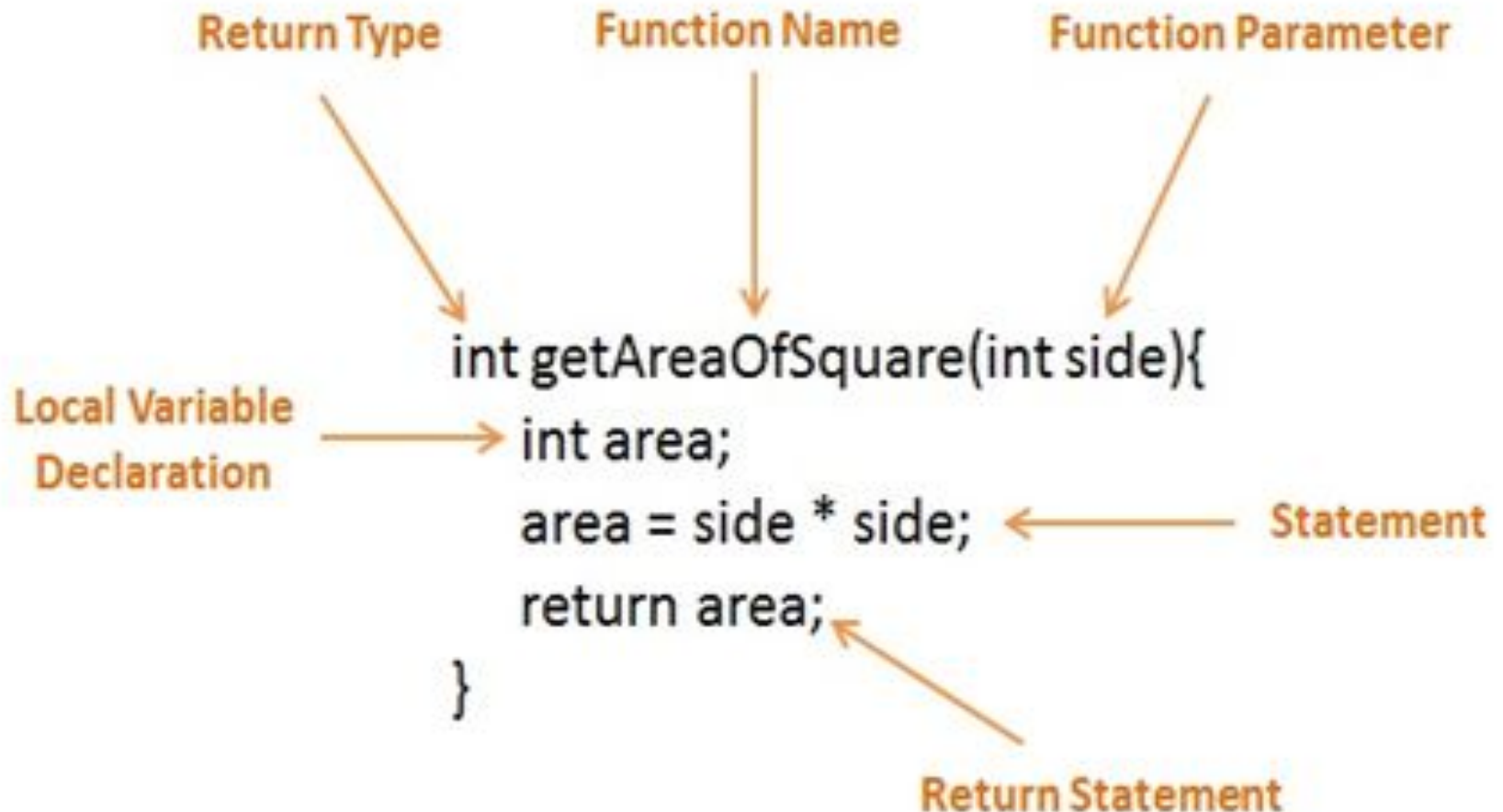
A function definition in C programming consists,

- function header and
- function body.

Syntax:

```
return data_type function_name(argument list)  
                                     // Function Header  
{  
    Body; // Function Body  
}
```


Function Definition



Function Calling

Syntax:

```
function_name(param_list);
```

General Form of a Function

```
return data type function_name(arglist)
    argument declaration
    {
        Local variable declaration;
        Executable statements;
        -----
        -----
        return(expression);
    }
```

Global variable & Local variable

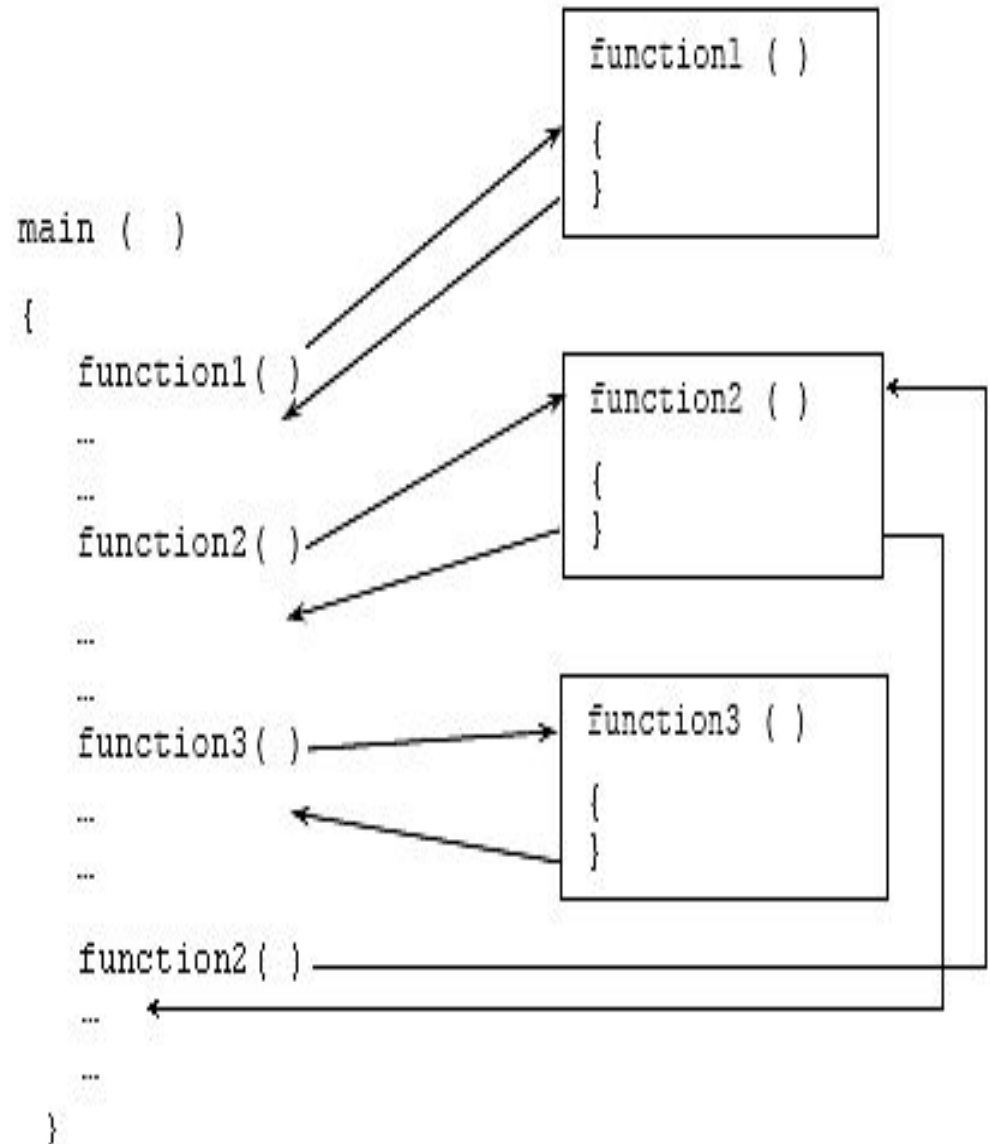
- **Global variable**

- Declared outside function block
- Accessible everywhere
- Global variable is destroyed only when a program is terminated.

- **Local variable**

- Declared inside function body
- Accessible only in the function
- Local variable is created when a function is called and is destroyed when a function returns.

- Functions are invoked by a function call.
- The calling function asks the called function to perform a task.
- When the task is done, the called function reports back to the calling function.



Example : Calling, Called Function

```
#include <stdio.h>
#include <conio.h>
void welcome();    /* Function Declaration */
void main()
{

    clrscr();
    welcome();    /* Function Calling */
    getch();
}

void welcome()    /* Function Definition, Called Function*/
{
    printf("\n WELCOME TO SRM INSTITUTE OF SCIENCE & TECHNOLOGY");
}
```

Note:

**In the above program main () ----calling function
welcome() ----called function**

Note :

- The return-value-type **void** indicates that a function **does not** return a value.
- By **default**, this return-value-type is **int**.
- If a function does not receive any parameters, the parameter list is **void**.

SLO - 2 :

Types: Call by Value, Call by Reference

Lesson : Types: Call by Value, Call by Reference

Upon completion of this lesson, you will be able to:

- **Different types Parameter Passing Techniques**
 - **Call by value**
 - **Call by reference**
- **Difference between Call by Value and Call by Reference**

Parameter Passing Techniques

Parameters : Inputs given by the user to the particular function

TWO TYPES

1. **Call by value**
2. **Call by reference**

Call by Value

- By default, C uses **call by value** to pass arguments.
- This method copies the actual value of an argument into the formal parameter of the function.
- In this case, changes made to the parameter **inside the function** have no effect on the argument.
- **Separate memory** location is allocated for actual and formal parameters.

Call by Value

```
int a=50, b=100;  
fun(a,b);
```

```
int fun(int x, int y)  
{  
    x= 40;  
    y=200; return y;  
}
```

Will not change

**Actual
Parameters**

Function call

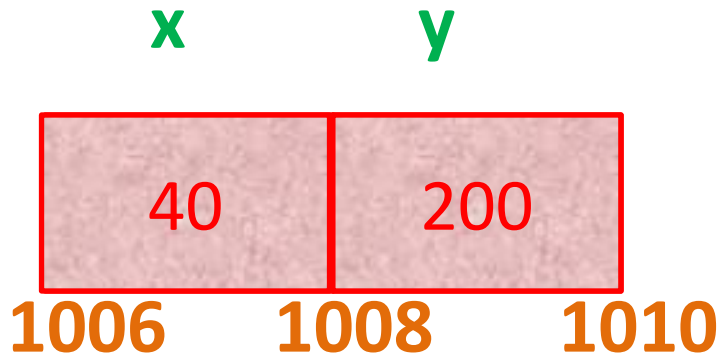
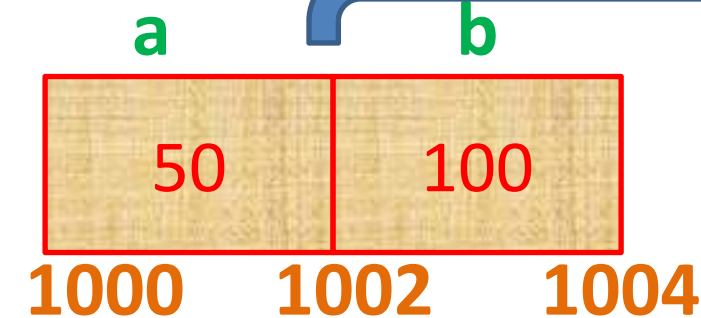
**Memory
location**

**Formal
Parameters**

**Function
Definition**

copied

Will change



TO EXCHANGING THE VALUES OF TWO VARIABLES USING CALL BY VALUE

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

```
// CALL BY VALUE
```

```
void swapnumber( int a, int b );
```

```
int main( )
```

```
{
```

```
    int m1 = 100, m2 = 200 ;
```

```
    clrscr();
```

```
    printf("Before swapping:\n");
```

```
    printf("\nm1 value is %d", m1);
```

```
    printf("\nm2 value is %d\n", m2);
```

```
    swapnumber( m1, m2 ); /*calling swapnumber function*/
```

```
    getch();
```

```
    return 0;
```

```
}
```

```
void swapnumber( int x1, int x2 )
```

```
{
```

```
    int tempvar ;
```

```
        tempvar = x1 ;
```

```
        x1 = x2 ;
```

```
        x2 = tempvar;
```

```
        printf("\nAfter swapping:\n");
```

```
        printf("\nm1 value is %d\n", x1);
```

```
        printf("\m2 value is %d\n", x2);
```

```
}
```

OUTPUT

Before swapping:

m1 value is 100

m2 value is 200

After swapping:

m1 value is 200

m2 value is 100

Call by reference

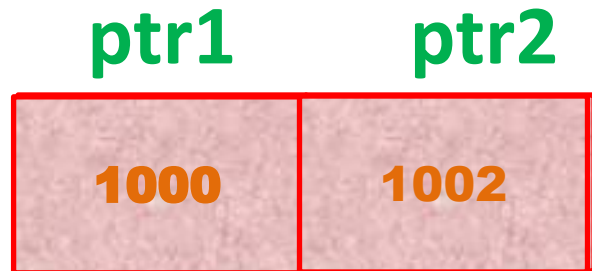
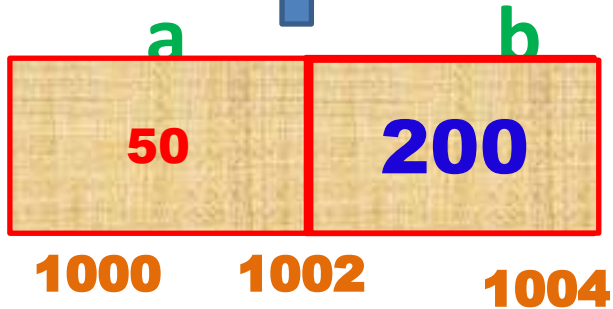
- Addresses of the actual arguments are copied and then assigned to the corresponding formal arguments, then this way of calling the function is known as call by reference.
- Here instead of passing values, we pass addresses
- Therefore, any changes made to the formal parameters will get reflected to actual parameters.
- Here both actual and formal parameters refers to same memory location.
- Declare the formal arguments of the function as pointer variables of an appropriate type.
- Because all the operations performed on the value stored in the address of actual parameters.

Call by reference

```
int a=50, b=100;  
fun(&a,&b);
```

```
int fun(int *ptr1, int *ptr2)  
{  
    *ptr1= 40;  
    *ptr2=200;  
}
```

get reflected



Actual Parameters

Function call

Memory location

Formal Parameters

passed the addresses

Function Definition

any changes

TO EXCHANGING THE VALUES OF TWO VARIABLES USING CALL BY REFERENCE

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

```
// CALL BY REFERENCE
```

```
void swapnumber( int *ptr1, int *ptr2 );
```

```
int main( )
```

```
{
```

```
    int m1 = 100, m2 = 200 ;
```

```
    clrscr();
```

```
    printf("Before swapping:\n");
```

```
    printf("\nm1 value is %d", m1);
```

```
    printf("\nm2 value is %d\n",m2);
```

```
    swapnumber( &m1, &m2 ); /*calling swapnumber function*/
```

```
    printf("\nAfter swapping:\n");
```

```
    printf("\nm1 value is %d\n", m1);
```

```
    printf("\nm2 value is %d\n", m2);
```

```
    getch();
```

```
    return 0;
```

```
}
```

```
void swapnumber( int *x1, int *x2 )
```

```
{
```

```
    int tempvar ;
```

```
        tempvar = *x1 ;
```

```
        *x1 = *x2 ;
```

```
        *x2 = tempvar;
```

```
}
```

Output

Before swapping:

m1 value is 100

m2 value is 200

After swapping:

m1 value is 200

m2 value is 100

& - To get the address

*** - To get the value**

M2

200

1004

1002

M1

100

1000

x1 □ **value(address(m1)) = 100**

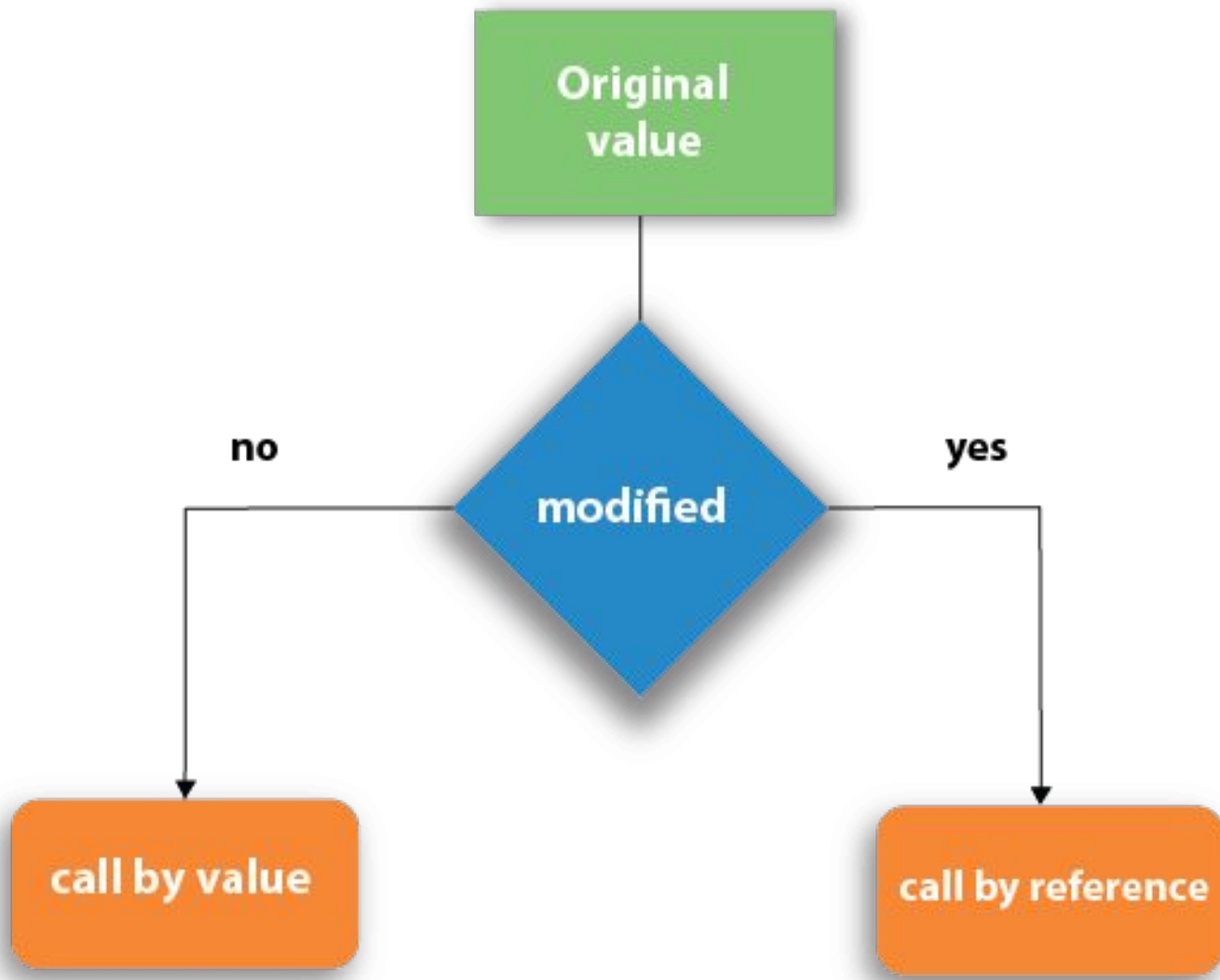
x2 □ **value(address(m2)) = 200**

tempvar

100

1006

1004



To returns the square and cube of a number using call by value and call by reference

```
#include<stdio.h>
void squ_cube( float n, float *ss, float *ccc );
// using call by value and call by reference
int main( void )
{
    float a = 5, s, c;
    clrscr();
    squ_cube( a, &s, &c );
    printf("The square of %f is %f and cube is %f\n", a, s, c);
    getch();
    return 0;
}
void squ_cube( float m, float *square, float *cube )
{
    *square = m * m;
    *cube = m * m * m;
}
```

OUTPUT

```
The square of 5.000000 is 25.000000 and cube is 125.000000
```

Difference between Call by Value and Call by Reference

S. No.	Call by Value	Call by Reference
1.	A copy of actual parameters is passed into formal parameters.	Reference of actual parameters is passed into formal parameters.
2.	Changes in formal parameters will not result in changes in actual parameters.	Changes in formal parameters will result in changes in actual parameters.
3.	Separate memory location is allocated for actual and formal parameters.	Same memory location is allocated for actual and formal parameters.
4.	The formal parameters are ordinary variables	The actual parameters are pointer variable
5.	At most only one values can be sent back to the calling function	Several results can be sent back to the calling function

SLO – 1 :

**Function with and without Arguments
and no return values**

SLO – 2 :

**Function with and without Arguments
and return values**

Types of Arguments /Parameters

- **Parameters** : Inputs given by the user to the particular function
- **Formal Arguments /Parameters**

The arguments which are given at the time of function declaration or function definition are called formal arguments.

- **Actual Arguments /Parameters**

The arguments which are given at the time of function calling are called actual arguments.

Types of function definition

- ✓ Functions **with** arguments **with** return value
- ✓ Functions **with** arguments **without** return value
- ✓ Functions **without** arguments **with** return value
- ✓ Functions **without** arguments **without** return value

To add internal and external Marks

Example : Formal Arguments, Actual Arguments & Functions with arguments with return value

```
#include <stdio.h>
#include <conio.h>
// FUNCTIONS WITH ARGUMENTS WITH RETURN VALUE
int add(int a, int b); /* Function Declaration with formal arguments */
void main()
{
    int c_prg_int, c_prg_ext, total;
    clrscr();
    printf("ENTER INTERNAL MARKS AND EXTERNAL MARKS\n");
    scanf("%d %d",&c_prg_int,&c_prg_ext);
    total = add(c_prg_int,c_prg_ext); /* Function Calling with actual arguments*/
    printf(" Total Marks =%d",total);
    getch();
}
int add(int cim,int cem) /* Function Definition, Called Function , formal arguments */
{
    int c_total_marks;
    c_total_marks = cim+cem;
    return c_total_marks;
}
```

To add internal and external Marks

Example : Functions with arguments without return value

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

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

```
// FUNCTIONS WITH ARGUMENTS WITHOUT RETURN VALUE
```

```
void add(int a, int b); /* Function Declaration with formal arguments */
```

```
void main()
```

```
{
```

```
int c_prg_int,c_prg_ext;
```

```
clrscr();
```

```
printf("ENTER INTERNAL MARKS AND EXTERNAL MARKS\n");
```

```
scanf("%d %d",&c_prg_int,&c_prg_ext);
```

```
add(c_prg_int,c_prg_ext); /* Function Calling with actual arguments */
```

```
getch();
```

```
}
```

```
void add(int cim,int cem) /* Function Definition, Called Function , formal arguments */
```

```
{
```

```
int c_total_marks;
```

```
c_total_marks = cim+cem;
```

```
printf(" Total Marks =%d",c_total_marks); /* No return statement */
```

```
}
```


To add internal and external Marks

Example : Function without arguments with return value

```
#include <stdio.h>
#include <conio.h>
// FUNCTION WITHOUT ARGUMENTS WITH RETURN VALUES
int add(); /* Function Declaration with no formal arguments */
void main()
{
    int total;
    clrscr();
    total = add(); /* Function Calling with no actual arguments */
    printf(" Total Marks =%d",total);
    getch();
}
int add() /* Function Definition, Called Function , no formal arguments */
{
    int c_prg_int, c_prg_ext,c_total_marks;
    printf("ENTER INTERNAL MARKS AND EXTERNAL MARKS\n");
    scanf("%d %d",&c_prg_int,&c_prg_ext);
    c_total_marks = c_prg_int+c_prg_ext;
    return c_total_marks;
}
```

To add internal and external Marks

Example : Function without arguments without return value

```
#include <stdio.h>
#include <conio.h>
// FUNCTION WITHOUT ARGUMENTS WITHOUT RETURN VALUES
void add(); /* Function Declaration with no formal arguments */
void main()
{
    clrscr();
    add(); /* Function Calling with no actual arguments */
    getch();
}
void add() /* Function Definition, Called Function , no formal arguments */
{
    int c_prg_int, c_prg_ext, c_total_marks;
    printf("ENTER INTERNAL MARKS AND EXTERNAL MARKS\n");
    scanf("%d %d", &c_prg_int, &c_prg_ext);
    c_total_marks = c_prg_int + c_prg_ext;
    printf(" Total Marks = %d", c_total_marks);
}
```

SLO – 1 :
**Passing array to functions with
return type**

PASSING ARRAY TO FUNCTION IN C

Array Definition:

- Array is collection of elements of similar data types

Types Passing array to function:

1. Passing array element by element
2. Passing entire array

Passing array element by element

- Individual elements are passed to function as argument
- Duplicate carbon copy of Original variable is passed to function
- So any changes made inside function does not affects the original value
- Function doesn't get complete access to the original array element
- Function passing method is "Pass by Value"

Passing array to function using call by value method

```
#include <stdio.h>
void disp( char ch)

{
    printf("%c ", ch);
}

int main()
{
    int x;
    char arr[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'};
    clrscr();
    for ( x=0; x<10; x++)
    {
        /* I'm passing each element one by one using subscript*/
        disp (arr[x]);
    }
    getch();
    return 0;
}
```

OUTPUT

a b c d e f g h i j

Passing array to function using call by reference

- When we pass the address of an array while calling a function then this is called function call by reference.
- When we pass an address as an argument, the function declaration should have a pointer as a parameter to receive the passed address.

Passing array to function using call by reference

```
#include <stdio.h>
void disp( int *num)
{
    printf("%d ", *num);
}

int main()
{
    int i,arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
    clrscr();
    for (i=0; i<10; i++)
    {
        /* Passing addresses of array elements*/
        disp (&arr[i]);
    }
    getch();
    return 0;
}
```

OUTPUT

1 2 3 4 5 6 7 8 9 0

Passing array to functions with return type (returnvalue.c)

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

```
float findAverage(int marks[]);
```

```
int main()
```

```
{
```

```
    float avg;
```

```
    int marks[] = {99, 90, 96, 93, 95};
```

```
    clrscr();
```

```
    avg = findAverage(marks);    // name of the array is passed as argument.
```

```
    printf("Average marks = %.1f", avg);
```

```
    return 0;
```

```
}
```

```
float findAverage(int marks[])
```

```
{
```

```
    int i, sum = 0;
```

```
    float avg;
```

```
    for (i = 0; i <= 4; i++) {
```

```
        sum += marks[i];
```

```
    }
```

```
    avg = (sum / 5);
```

```
    getch();
```

```
    return avg;
```

```
}
```

OUTPUT

```
Average marks = 94.0
```

SLO – 2 :

Recursion functions

Recursion function

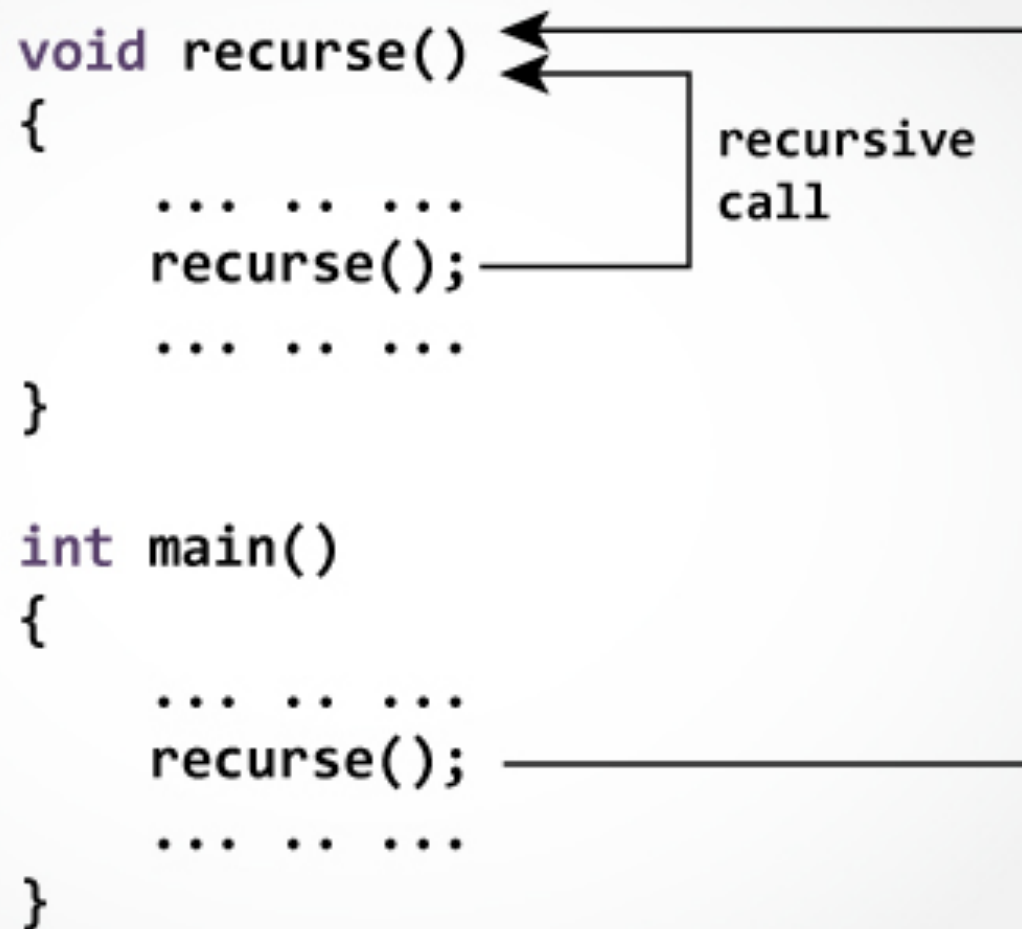
- A function that **calls itself** is known as a recursive function. And, this technique is known as recursion.
- The recursion continues **until some condition is met** to prevent it.
- To prevent infinite recursion, if...else statement (or similar approach) can be used where one branch makes the **recursive call**, and other doesn't.

How recursion works

```
void recurse()  
{  
    ... ..  
    recurse();  
    ... ..  
}
```

```
int main()  
{  
    ... ..  
    recurse();  
    ... ..  
}
```

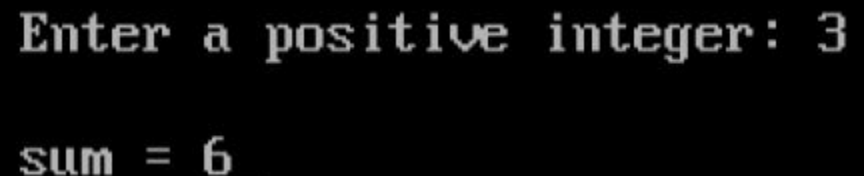
How does recursion work?



Example: Sum of Natural Numbers Using Recursion

```
#include <stdio.h>
int sum(int n);
int main() {
    int number, result;
    printf("Enter a positive integer: ");
    scanf("%d", &number);
    result = sum(number);
    printf("sum = %d", result);
    return 0;
}
int sum(int n) {
    if (n != 0)
        // sum() function calls itself
        return n + sum(n-1);
    else
        return n;
}
```

OUTPUT

A screenshot of a terminal window with a black background and white text. The first line shows the prompt "Enter a positive integer: 3" where '3' is the user input. The second line shows the output "sum = 6" followed by a cursor character (an underscore).

```
Enter a positive integer: 3
sum = 6_
```