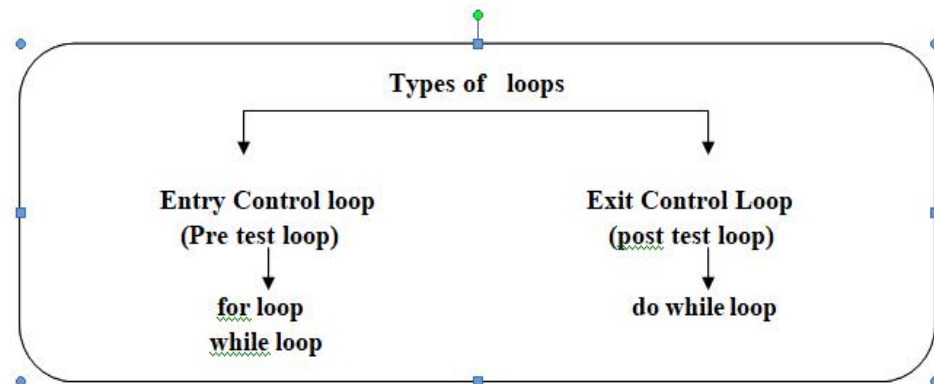
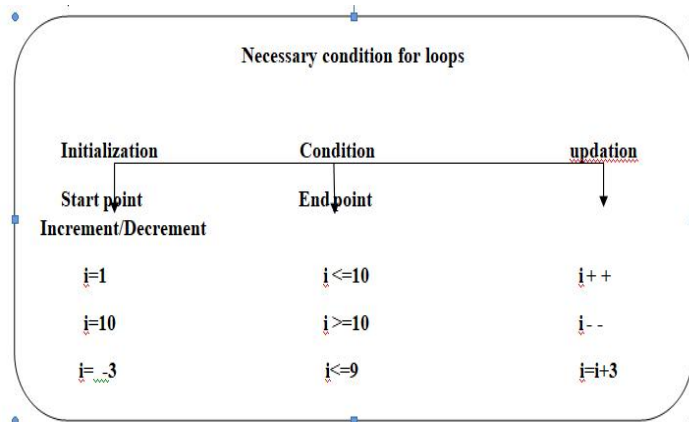


Unit 3

Q1 Iterative Statements(loops) / Different Types of Loops / Control Statement

- Programming languages provide various control structures that allow for more complicated execution paths.
- A loop statement allows us to execute a statement or group of statements multiple times.
- Iterative statement (loops) is used to repeat the same block of code. If we want to perform any work in a series then we used iterative statements (loops). Loops are used to perform a work in a series.
- If we want to display the series of 1 to 10 without loops then we have to take 10 variables and after initialize the 10 values display all the variables .but if we used loop then we have to take only single variable and display the value of that variable inside the body of loop. The loop iterates 10 times and displays 10 values.
- A loop consists of two parts one is the body of loop and other is control statement. A control statement perform a logical test (condition) whose result is either true or false .if the result of the condition is true than the statement contained in the body of the loop are executed otherwise the loop is terminated.
- The condition is checked either before or after body of the loop. If the control statement is placed before the body of the loop is called entry control loop or pretest loop like for and while loop.
- And if the Control statement (condition) is placed after the body of the loop is called exit control loop or posttest loop like do while loop. Loops are also called repetitive statements.



for loop

WAP to display the series of 1 to 10

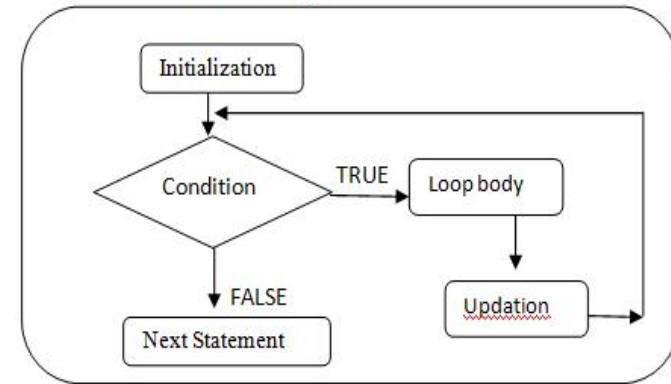
```
void main()
{
int i;
for(i=1;i<=10;i++)
{
printf("%d", i);
}
}
```

Output: 1 2 3 4 5 6 7 8 9 10

Syntax

```
for (initialize ; condition ; updation )
{
Set of Statements; loop body
}
Next statement;
```

Block Diagram



While loop

WAP to display the series of 1 to 10 and display its sum also

```
void main()
{
int i ,sum=0
i=1;
while(i<=10)
{
printf("%d", i);
sum=sum+i;
i++;
}
printf("\n sum=%d",sum);
}
```

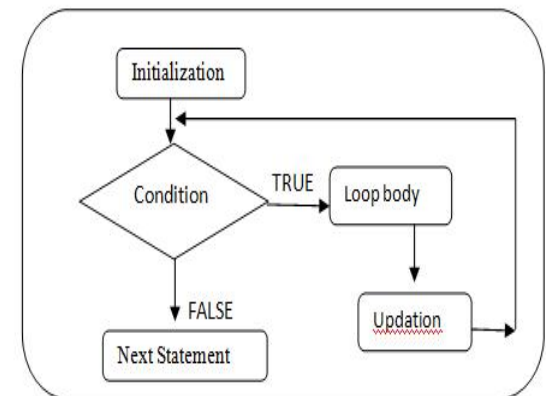
Output: 1 2 3 4 5 6 7 8 9 10

sum = 55

Syntax

```
initialize :
while (condition)
{
set of Statements ; loop body
updation:
}
Next statement;
```

Block Diagram



do while

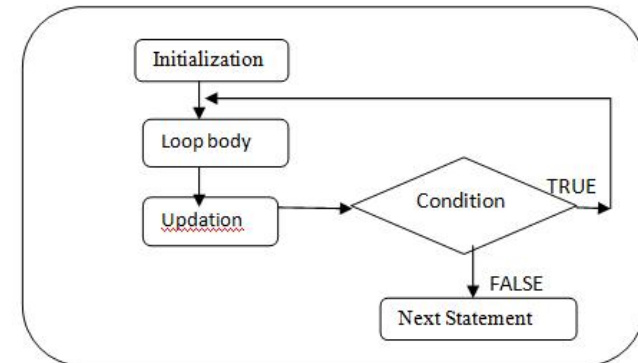
WAP the following series -9 -6 -3 0 3 6 9

```
void main()
{
int i ,
i= -9;
do
{
printf("%d", i);
i= i + 3 ;
}
while(i <=9);
}
```

Syntax

```
initialize :
do
{
set of Statements ; loop body
update:
}
while(condition);
Next statement;
```

Block Diagram



Difference between While & do while loop / (Enter control & Exit Control loop) (pre test and post test loop)

While

- while is a Entry control loop
- while is a pretest loop
- In while loop if the condition is false then there is no output
- In a while loop the condition is first tested and if it returns true then it goes in the loop body
- While loop test condition before statement execution

Do while

- do while is a Exit control loop
- do while is a post test loop
- in do while if the condition is false then there is at least one output
- In a do-while loop the condition is tested at the last.
- do while test condition after statement execution

Syntax:
while (condition)
{
Statements;
}

Example:
void main()
{
int n=10;
while(n<=5)
{
printf("%d",n);
n++;
}
}
Output: No output
because condition is false the
value of n is 10 and condition is
10<5 not true so there is no
output

Syntax:
do
{
Statements;
}while(condition);

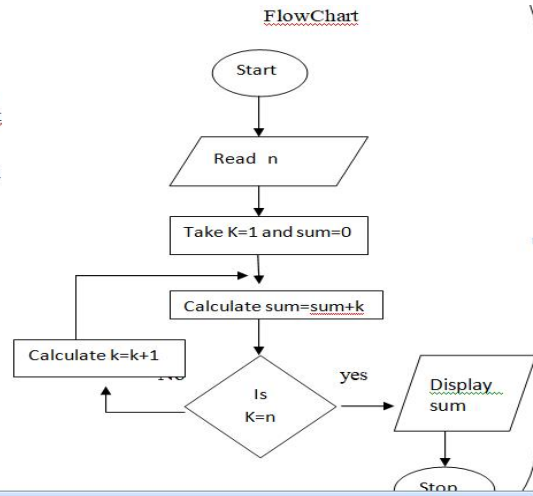
Example:
void main()
{
int n=10;
do
{
printf("%d",n);
n++;
}while(n<=5);
}
Output: 5
because condition is check at
the end of the loop .loop
executed at least once then
check the condition

Find the Sum of n Natural numbers

Algorithm

1. Start
2. Read n
3. Take k=1 and sum=0
4. Calculate sum=sum+k
5. Is k=n goto end
6. Else k=k+1
7. And again go to step 4
8. End

FlowChart

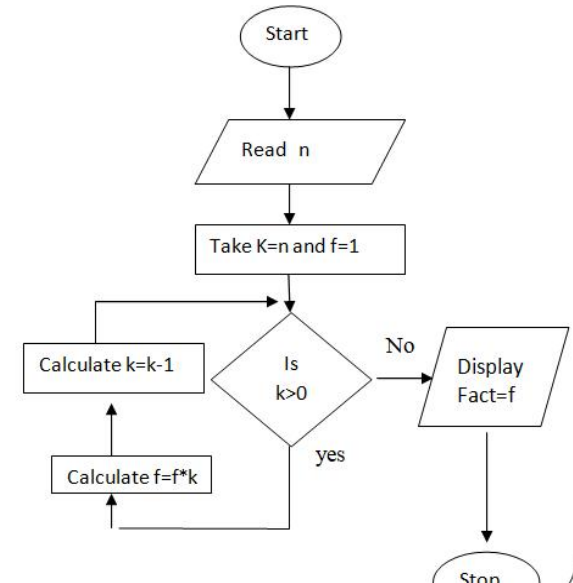


Find the Factorial of any entered number

Algorithm

1. Start
2. Read n
3. Take k=n and f=1
4. Is k>0 yes
5. Calculate f=f*k
6. k=k-1 goto step 4
7. Else
8. Display f
9. End

FlowChart



Program to find factorial of entered number using for loop

```
#include<stdio.h>
void main()
{
    int num,i,fact=1;
    clrscr();
    printf("enter the number to find factorial");
    scanf("%d",&num);
    for(i=num ; i>0 ; i--)
    {
        fact=fact*i;
    }
    printf("\n factorial of %d is %d",num,fact);
    getch();
}
```

A simple program to find out whether a given number is prime or not.

```
#include<stdio.h> # include<process.h>
void main()
{
    int a,i,n;
    printf("enter the number to be checked");
    scanf("%d",&n);
    for(i=2;i<n;i++)
    {
        If (n%i==0)
        {
            printf("No is Not Prime");
            exit(0); }
        If(i==n)
        printf("the given number is prime");
    }
}
```

**Program to display the Fibonacci series up to a +ve number .
The Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21,34,55,89**

```
#include <stdio.h>
Void main()
{
    int t1 = 0, t2 = 1, nextTerm = 0, n;
    printf("Enter a positive number: ");
    scanf("%d", &n);
    // displays the first two terms which is always 0 and 1
    printf("Fibonacci Series: %d, %d, ", t1, t2);
    nextTerm = t1 + t2;
    while (nextTerm <= n)
    {
        printf("%d, ", nextTerm);
        t1 = t2;
```

- 1 WA Program to find the Reverse of digits of a number entered by user.

```
•
• #include<stdio.h>
• #include<conio.h>
• void main()
• {
•     int num,d,s=0;
•     clrscr();
•     printf("Enter the number");
•     scanf("%d",&num);
•     while(num>0)
•     {
•         d=num%10;
•         s=s*10+d;
•         num=num/10;
•     }
•     printf("theReverseof digit is %d", s);
•     getch();
• }
• OUTPUT: Enter the number : 153
•           the rev of digit is 351
```

Sum of digits of any
number
s=s+d
Count no of digits
s++

```
t2 = nextTerm;
nextTerm = t1 + t2; } }
```

Program to find whether the entered number is Armstrong or not.

```
#include<stdio.h>
void main()
{
    int num,t,d,s=0;
    clrscr();
    printf("enter the number");
    scanf ("%d",&num);
    t=num;
    while(num >0)
    {
        d=num%10;
        s=s+(d*d*d);
        num = num/10;
    }
    if(t==s)
    printf("\n entered number is an Armstrong number\n ");
    else
    printf("\n this is not an Armstrong number\n");
    getch();
}
```

Output: enter the number 153
entered number is an Armstrong number

Number is Palindrome
s=s*10+d

Like 151 is palindrome
where no = its reverse

Write a C program to calculate the average of n numbers using do while loop

Write a C program using while to find the GCD of two numbers

Write a C program to Print the following sequences

2, 4, 8, 16, 32, 64 -4, -2, 0, 2, 4

$x + 2x + 3x + 4x + \dots + nx$

$x^1 - x^2 + x^3 - x^4 + x^5 - \dots$ Upto n terms

$1 + 1/2 + 1/3 + 1/4 + \dots + 1/n$

Write a C program to Print the following series 35 31 27 23 19 23 27 31 35

Write a C Program to display the fibonacci series up to n terms .

Write a C program To print all integers which are not divisible either 2 or 3 and lie 1 to 100?

Write a C program to Count the no. of digits in any entered number

Write a C program to Print all Armstrong number between 101 to 1000

Write a C program to Print all prime number between 1 to 100

Write a C Program to Display Prime Numbers Between Two Intervals

Write C program to Display Factors of a Number

Write a C Program to Convert Binary Number to Decimal

Write a C Program to Convert Decimal to Binary Number.

Nested Loops

- Loop inside the loop
- These loops contain another looping statement in a single loop.
- These types of loops are used to create matrix.

Any loop can contain a number of loop statements in itself. If we are using loop within loop that is called nested loop. In this the outer loop is used for counting rows and the internal loop is used for counting columns. We can write any loop inside any loop in c i.e. we can write for loop inside the loop or while loop or do while loop etc.

When you "nest" two loops, the outer loop takes control of the number of complete repetitions of the inner loop. Means **the outer loop changes only after the inner loop is completely finished** .While all types of loops may be

| | |
|--|--|
| <pre> for (init; condition; increment) { for (init; condition; increment) { Statement for inner loop; } statement for outer loop . } </pre> | <pre> void main() { int num1 ,num2; for(num1 = 0; num1 <3; num1++) { for(num2 = 0; num2 <= 2; num2++) { printf (“ num1= %d , num2= %d \n ”,num1 , num2); } printf(“\n”); } } </pre> |
| <p>Program to print the following Pattern...</p> <pre> void main () { int i,j; clrscr(); for(i=1; i<=5; i++) { for(j=1; j<=i; j++) { printf(“%d”,j); } printf(“\n”); } getch(); } </pre> <pre> 1 2 2 3 3 3 4 4 4 4 5 5 5 5 5 </pre> | <p>Program to print the following Pattern...</p> <pre> void main () { int i,j,sp; clrscr(); for(i=1; i<=5; i++) { for(sp=4; sp>=1; sp--) { printf(“\n”); } for(j=1; j<=i; j++) { printf(“%d” , j); } printf(“\n”); } getch(); } </pre> <pre> 1 1 2 1 2 3 1 2 3 4 1 2 3 4 5 </pre> |

21) Print the following patterns:-

```

1
1 2
1 2 3
1 2 3 4
1 2 3 4 5

```

```

I
I N
I N D
I N D I
I N D I A

```

```

1
2 3
4 5 6
7 8 9 10
11 12 13 14 15

```

```

*
* *
* * *
* * * *
* * * * *

```

```

5
5 4
5 4 3
5 4 3 2
5 4 3 2 1

```

```

1
2 1
3 2 1
4 3 2 1
5 4 3 2 1

```

```

      *
     ***
    *****
   ********
  *****
 *****
*****
PYRAMID

```

```

      1
     2 1 2
    3 2 1 2 3
   4 3 2 1 2 3 4
  5 4 3 2 1 2 3 4 5

```

```

1
2 2
3 3 3
4 4 4 4
3 3 3
2 2
1

```

Jump Statement

Jump statements can be used to modify the behavior of conditional and iterative statements.

Jump statements allow you to exit a loop, start the next iteration of a loop, or explicitly transfer program control to a specified location in your program and immediately exit from the program

When executing a loop or conditional statement like switch it become desirable to skip a part of the loop or to leave the loop as soon as certain condition occur. C permits the jumps from one statement to another within a block (loop) as well as jump out of a block (loop).

break statement (block terminator)

continue statement (skip any iteration)

exit function (program terminator)

goto statement (shift one position to another)

BREAK

A **break** statement is used to terminate the execution of the rest of the block where it is present and takes the control out of the block to the next statement. It is mostly used in loops and switch-case to bypass the rest of the statement and take the control to the end of the loop.

Another point to be taken into consideration is that **break** statement when used in nested loops only terminates the inner loop where it is used and not any of the outer loops.

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

```
Void main()
```

```
{
```

```
int i;
```

```
for(i=1;i<=15;i++)
```

```
{
```

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

```
if(i==10)
```

```
break;
```

```
}
```

```
Printf("loop finish");
```

```
}
```

Output:- 1 2 3 4 5 6 7 8 9 10 loop finish

Exit()

exit is a system call that terminates the current process. When we used exit () function in our program than exit immediately transfer the program at the end means terminate the program. The statements that are written after the exit function never executed.

exit () is used to exit the program as a whole. In other words it returns control to the [operating system](#)

After exit () all memory and temporary storage areas are all flushed out and control goes out of program.

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

```
Void main()
```

```
{
```

```
int i;
```

```
for(i=1;i<=15;i++)
```

```
{
```

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

```
if(i==10)
```

```
exit(0);
```

```
}
```

```
Printf("loop finish");
```

```
}
```

Output:- 1 2 3 4 5 6 7 8 9 10

Continue Statement

It is sometimes desirable to skip some statements inside the loop. In such cases, continue statements are used. The continue statement is used to repeat the same operations once again even if it check the error.

The continue statement is used for the next iteration of the loop to take place skipping any statement in between the loop, it's used to terminate the current iteration and continue with the next iteration of the loop.

Similar to a **break** statement, in case of nested loop, the **continue** passes the control to the next iteration of the inner loop where it is present and not to any of the outer loops..

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

```
Void main() {  
    int i;  
    for(i=1;i<=15;i++)  
    {  
        printf("%d\n",i);  
        if(i==10)  
            continue; }  
    Printf("loop finish");  
}
```

Output:- 1 2 3 4 5 6 7 8 9 11 12 13 14 15 loopfinish

Difference between break and Exit()

break is a keyword in C.

break causes an immediate exit from the switch or loop (for, while or do).

break is a reserved word in C; therefore it can't be used as a variable name.

No header files needs to be included in order to use break statement in a C program.

break transfers the control to the statement follows the switch or loop (for, while or do) in which break is executed.

Example of break

```
CODE  
// some code here before while Loop  
while(true)  
{  
    ...  
    if(condition)  
        break;  
}  
// some code here after while Loop
```

In the above code, break terminates the while loop and some code here after while loop will be executed after breaking the loop.

Conclusively, break is a program control statement which is used to alter the flow of control upon a specified conditions.

exit() is a standard library function.

exit() terminates program execution when it is called.

exit() can be used as a variable name.

stdlib.h needs to be included in order to use exit().

exit() returns the control to the operating system or another program that uses this one as a sub-process.

Example of exit()

```
CODE  
// some code here before while Loop  
while(true)  
{  
    ...  
    if(condition)  
        exit(-1);  
}  
// some code here after while Loop
```

In the above code, when if(condition) returns true, exit(-1) will be executed and the program will get terminated. Upon call of exit(-1); -1 will be returned to the calling program that is operating system most of the time. The some code here after while loop will never be executed in this case.

exit() is a library function, which causes immediate termination of the entire program, forcing a return to the operating system.

Modular programming (functional programming)

Modular programming (functional programming) is the process of subdividing a computer program into separate sub-programs. A module is a separate software component. It can often be used in a variety of applications and functions with other components of the system. Similar functions are grouped in the same unit of programming code and separate functions are developed as separate units of code so that the code can be reused by other applications.

If a program is lengthy it becomes very difficult for the programmer to handle it. Normally Larger programs are more errors and its difficult to correct there errors. Therefore such programmes should be broken down into a number of small units called module or functions .

The function or modules is a set of instructions which are to be accessed repeatedly from several different places within a program and call whenever necessary .This avoids rewriting of functions on every access. its also called reusability.

The advantages of using modular programming:

- fewer bug because each set of programming commands is shorter
- algorithm is more easily understood
- many programmers can be employed, one on each of the modules
- programmers can use their expertise on particular techniques
- testing can be more thorough on each of the modules
- allows library programs to be inserted
- all of which saves time and means the finished program can be completed more quickly

Functions

A function is a module or block of program code which deals with a particular task. Making functions is a way of isolating one block of code from other independent blocks of code.

Functions serve two purposes.

They allow a programmer to say: 'this piece of code does a specific job which stands by itself and should not be mixed up with anything else', Second they make a block of code reusable since a function can be reused in many different contexts without repeating parts of the program text.

A function can take a number of parameters, do required processing and then return a value. There may be a function which does not return any value.

You already have seen couple of built-in functions like printf(); Similar way you can define your own functions in C language. Any C program written in a C function main().

There are two types of function in C programming

- System Defined Function**
- User Defined Function**

The system defined function are also called library function or built in function .All system define function are predefined in the C library. And they can access directly from the library .

Like sqrt(),printf() ,strcpy(),gets() etc. But most of the application programmer needs some functions that are not in the library .these function must be written by the programmer is called user define function.

C provides programmer to define their own function according to their requirement known as user defined functions. Suppose, a programmer wants to find factorial of a number and check whether it is prime or not in same program. Then, he/she can create two separate user-defined functions in that program: one for finding factorial and other for checking whether it is prime or not.

Advantages of user defined functions

The complexity of the entire program can be divided into simple subtask and then function subprograms can be written for each subtask.

The subprograms are easier to write, understand, and debug.

A function can be shared by other programs by compiling this function separately and loading and linking them together.

In C, a function can call itself again. It is called a recursive function. Many calculations can be done easily by using recursive functions such as calculation of factorial of a number, etc.

Reduction in size of program due to program code of a function can be used again and again, by calling it.

A library of a function can be designed and tested for use of every programmer.

Programmer working on large project can divide the workload by making different functions.

Prototype of a Function (How functions are declare) /Elements of user defined function

In C programming the function name and there type must be declared and defined before they are used in the program. Functions are basically special type of programs ,which perform a specific task ,but don't work independently as main() function. They get executed only when some other program calls them. if required they receive data from the program and may or may not return the data in to the program.

Three Necessary Condition for functions

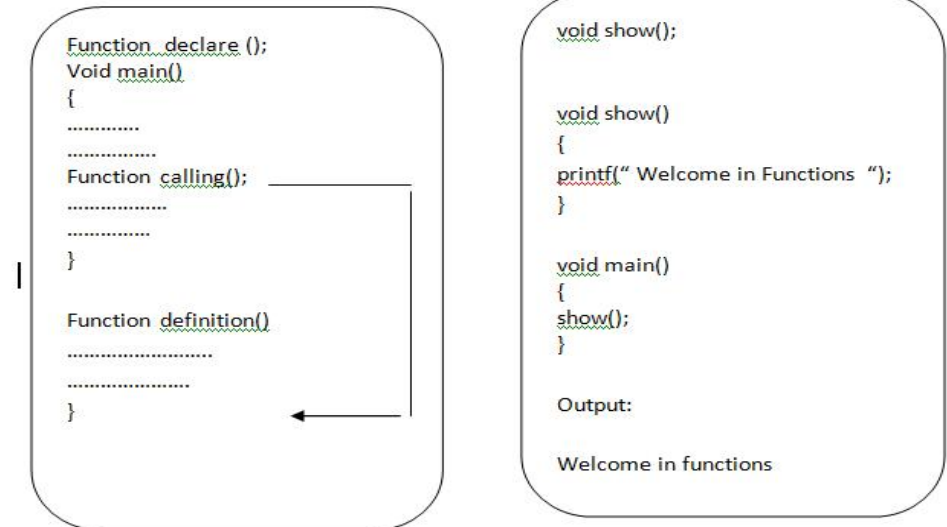
- 1 **Function Declare (Before main())**
- 2 **Function Calling (With in the main())**
- 3 **Function Definition (After the main())**

Prototype of a Function (How functions are declare)

The general form of a function definition in C programming language is as follows –

```
return_type function_name( parameter list )  
{  
    body of the function  
}
```

A function definition in C programming consists of a *function header* and a *function body*. Here are all the parts of a function –



A function **declaration** tells the compiler about a function's name, return type, and parameters.

A function **definition** provides the actual body of the function.

Function name should be declare before the main program.

The function definition means writing an actual code for a function which have a specific and identifiable task.

Suppose we have declare a function `sum()` which find the sum of entered number then we have to write a set of instruction to do sum in the definition. And In order to use this function we need to invoke it at a required place in the program .this is known as **function calling**.

Return Type – A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.

Function Name – This is the actual name of the function. The function name and the parameter list together constitute the function signature.

Parameters – A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

Function Body – The function body contains a collection of statements that define what the function does.

•

Here Return type works with respect to output and Parameters work with respect to Input.

For transfer input values from main program to function definition we use Parameters and For transfer output values from function definition to main program we use return types.

1 Return type always related to the output of our program . if the function is without return type means function definition does not return any value to the function calling. And if the the function is with return type means the function definition returns the computed value to the calling functions.

2 Arguments always related to the input of our program. if the function is declare without arguments means the function definition does not receive any data from the main program through function calling. The data is taken in the definition from the user .and if the function is declare with arguments the function definition receive the data from the main program and process on that data.

Function without Return type without Parameters

```
void sum ();

void main()
{
    sum();
}

void sum()
{
    int a,b,c;
    printf("Enter the value of a,b");
    scanf("%d%d",&a,&b);
    c=a+b;
    printf("sum=%d" c);
}
```

Function without Return type with Parameters

```
void sum (int , int);

void main()
{
    int a,b ;
    printf("Enter the value of a,b");
    scanf("%d%d",&a,&b);
    sum(a,b);
}

void sum( int x, int y)
{
    int z =x+y;
    printf("sum=%d" z);
}
```

Function with Return type without Parameters

```
int sum ();

void main()
{
    int c;
    c=sum();
    printf("sum=%d" c);
}

void sum()
{
    int a,b,c;
    printf("Enter the value of a,b");
    scanf("%d%d",&a,&b);
    c=a+b;
    return(c);
}
```

Function with Return type with Parameters

```
int sum (int , int);

void main()
{
    int a,b ,c ;
    printf("Enter the value of a,b");
    scanf("%d%d",&a,&b);
    c=sum(a,b);
    printf("sum=%d" c);
}

void sum( int x, int y)
{
    int z =x+y;
    return(z);
}

Here a ,b are the arguments and x ,y are  
the parameters
```

Arguments and Parameters

An **argument** is referred to the values that are passed within a function when the function is called. These values are generally the source of the function that require the arguments during the process of execution. These values are assigned to the variables in the definition of the function that is called. The type of the values passed in the function is the same as that of the variables defined in the function definition. These are also called **Actual arguments** or **Formal Parameters**.

Parameters

The parameter is referred to as the variables that are defined during a function declaration or definition. These variables are used to receive the arguments that are passed during a function call. These parameters within the function prototype are used during the execution of the function for which it is defined. These are also called Formal arguments or Actual Parameters.

The following are the points to be remembered while using arguments and parameters

- 1 The number of arguments should be equal to the number of parameters.
- 2 There must be one to one mapping between arguments and parameters.
- 3 They should be in the same order and have the same data type
- 4 Same variables or different variable can be used as arguments and parameters

```
#include <stdio.h>
int sum(int a, int b)
{
    return a + b;
}

Void main()
{
    int num1 = 10, num2 = 20, res;
    res = sum(num1, num2);
    printf("The summation is %d", res);
}
```

Here num1 & num2 as ARGUMENTS. and a,b are the parameters

Different Parameter passing Techniques

In the function the arguments are passed to the function and their values are copied in the corresponding function definition . Means The values that are passes in the calling of function as an Arguments are directly copied in the definition of the calling function as a parameter . There are two techniques for passing the parameters:

- Call by Value
- Call By Reference

Call by Value : Actual Values are passed (Original values are copied)

In this Method the values of the Arguments are passed from a calling function to the definition ,the actual values are copied into the definition of that function .if any changes are made to the values in the definition (parameters) there will not be any change in the original values within the calling function

Call by Reference :The Address are passed (Original values are not copied, Address is copied)

In this method the actual values are not passed ,instead their addresses are passed to the calling function , Here no values are copied as the memory location themselves are referenced. If any changes are made to the values in the definition (parameters) then the original values will get changed with in the calling function.

Recursion (Recursive Function)

A Function calls itself again and again until a specific condition is called recursion. Recursion is a technique that defines a function in terms of itself.

That is a function which perform a particular task is repeatedly calling itself .There must be an exclusive stopping condition in a recursive function. otherwise the function will not be terminated its enter in the infinite loop.

if a program allows you to **call a function inside the same function**, then it is called a **recursive call** of the function.

There are some advantages of recursion

By using Recursion we call a function continuously until to a specific condition

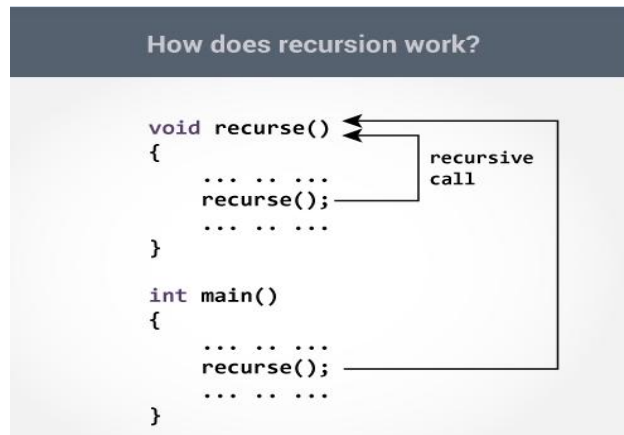
Through Recursion one can Solve problems in easy way while its iterative solution is very big and complex. Ex : tower of Hanoi

The recursion is very flexible in data structure like stacks, queues, linked list and quick sort.

Using recursion, the **length of the program can be reduced**.

Using recursion we can avoid **unnecessary calling** of functions.

Recursive functions can be effectively used to solve problems where the solution is expressed in terms of applying the same solution.



| Iteration | Recursion |
|--|---|
| Iteration explicitly user a repetition structure. | Recursion achieves repetition through repeated function calls. |
| Iteration terminates when the loop continuation. | Recursion terminates when a base case is recognized. |
| Iteration keeps modifying the counter until the loop continuation condition fails. | Recursion keeps producing simple versions of the original problem until the base case is reached. |
| Iteration normally occurs within a loop so the extra memory assigned is omitted. | Recursion causes another copy of the function and hence a considerable memory space's occupied. |
| It reduces the processor's operating time. | It increases the processor's operating time. |

Tail Recursion

Tail recursion is a form of linear recursion. In tail recursion, **the recursive call is the last thing the function does**. Often, the value of the recursive call is returned. As such, tail recursive functions can often be easily implemented in an iterative manner; by taking out the recursive call and replacing it with a loop, the same effect can generally be achieved.

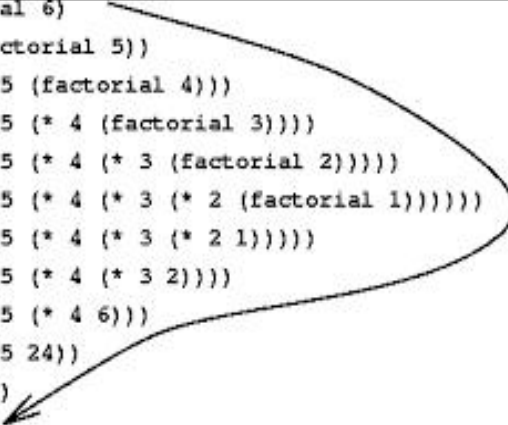
In fact, a good compiler can recognize tail recursion and convert it to iteration in order to optimize the performance of the code.

A good example of a tail recursive function is a function to compute the GCD,Factorial.

WAP find the Factorial of any Numbers Using Recursion

```
#include <stdio.h>
int fact (int);
int main()
{
    int n,f;
    printf("Enter the number whose factorial you want to calculate?");
    scanf("%d",&n);
    f = fact(n);
    printf("factorial = %d",f);
}
int fact(int n)
{
    if ( n == 1)
        return 1;
    else
        return n*fact(n-1); }
```

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1)))))
(* 6 (* 5 (* 4 (* 3 (* 2 1))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

**WAP find Sum of Natural Numbers Using Recursion**

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

find the nth term of the Fibonacci series Using Recursion

```
#include<stdio.h>
int f(int);
Void main()
{
    int n, m= 0, i;
    printf("Enter Total terms:n");
    scanf("%d", &n);
    printf("Fibonacci series terms are:n");
    for(i = 1; i <= n; i++)
    {
        printf("%d", fibonacci(m));
        m++;
    } }
int fibonacci(int n)
{
    if(n == 0 || n == 1)
        return n;
    else
```

| | |
|--|---|
| | return(fibonacci(n-1) + fibonacci(n-2)); } |
|--|---|

Enumeration (Enum)

- Enumeration is a user defined datatype in C language.
- It is used to assign names to the integral constants which makes a program easy to read and maintain.
- The keyword “enum” is used to declare an enumeration. ...
- The enum keyword is also used to define the variables of enum type.
- Here is the syntax of enum in C language,
- `enum enum_name{const1, const2, };`
- The enum keyword is also used to define the variables of enum type. There are two ways to define the variables of enum type as follows.
- `enum week{sunday, monday, tuesday, wednesday, thursday, friday, saturday};`
- `enum week day;`

Program for implement Enum

```
#include<stdio.h>
enum year
{
Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec
};
```

```
Void main()
{
int i;
for (i=Jan; i<=Dec; i++)
printf("%d ", i);
```

```
} Output:0 1 2 3 4 5 6 7 8 9 10 11
```