Iteration: Use of Loops in C Programming Language

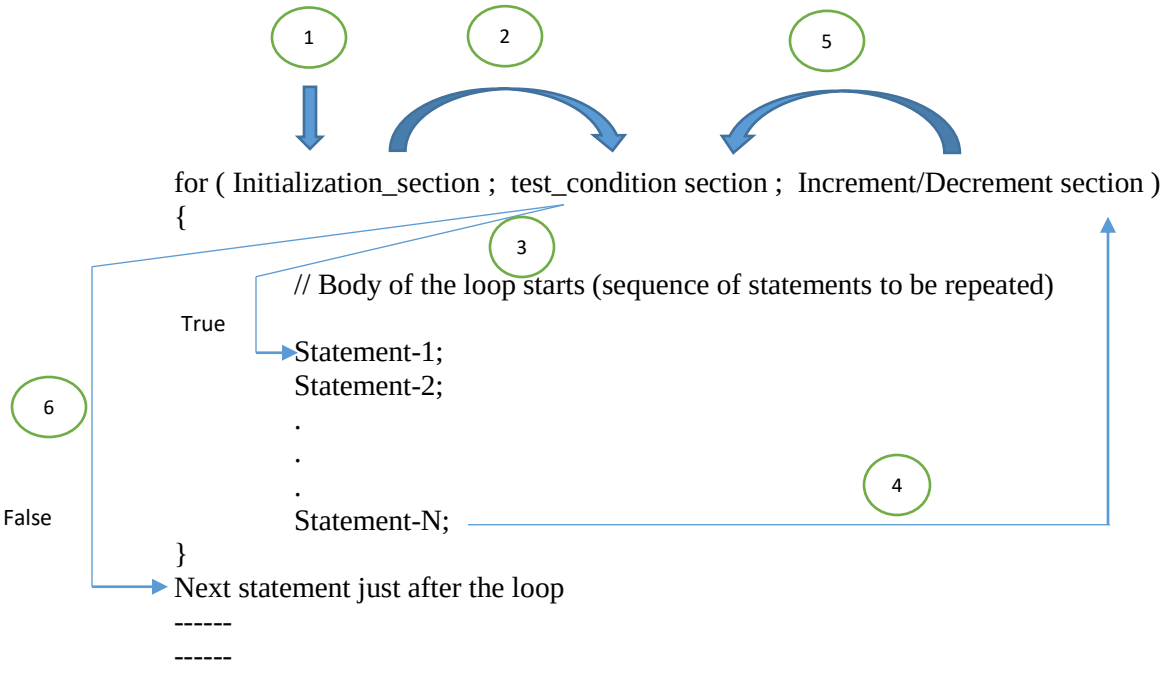
Loops are used in programming to repeat a sequence of statements again and again until a specific condition is met. There are two types of loops in C programming:

- **Entry Controlled Loop:** In this type of loop, the loop condition is checked before entering the loop. For example, for and while loops.
- **Exit Controlled Loop:** In this type of loop, the loop condition is checked at the end of the loop. For example, for and while loops. For example, do-while loop.

1. for Loop:

The for loop is commonly used when the number of iterations is known. That's why it is also called as deterministic loop.

Syntax of for loop is:



Working of for Loop:

Step 1: The initialization statement is executed only once.

Step 2: Then, the test expression is evaluated.

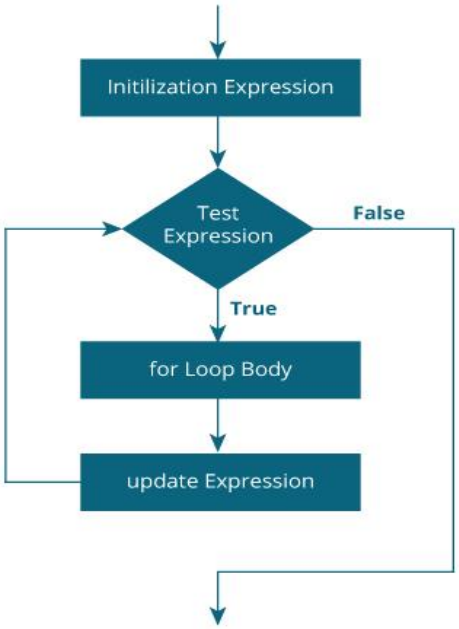
Step 3: If the test expression is true (i.e. 1), program control enters the very first statement inside the body of for loop and then executes it.

Step 4: After executing the last statement inside the for loop, program control goes back to the increment/decrement section as shown in above figure.

Step 5: After, performing increment or decrement, program control again jumps to the Testing section. This process is repeated till the test condition remains true.

Step 6: If the test expression is false (i.e. 0), for loop is terminated and program control jumps to the next statement just after the for loop.

Flowchart of 'for' loop



Example program of for loop

// Program to calculate the sum of first n natural numbers (i.e. 1,2,3...,n)

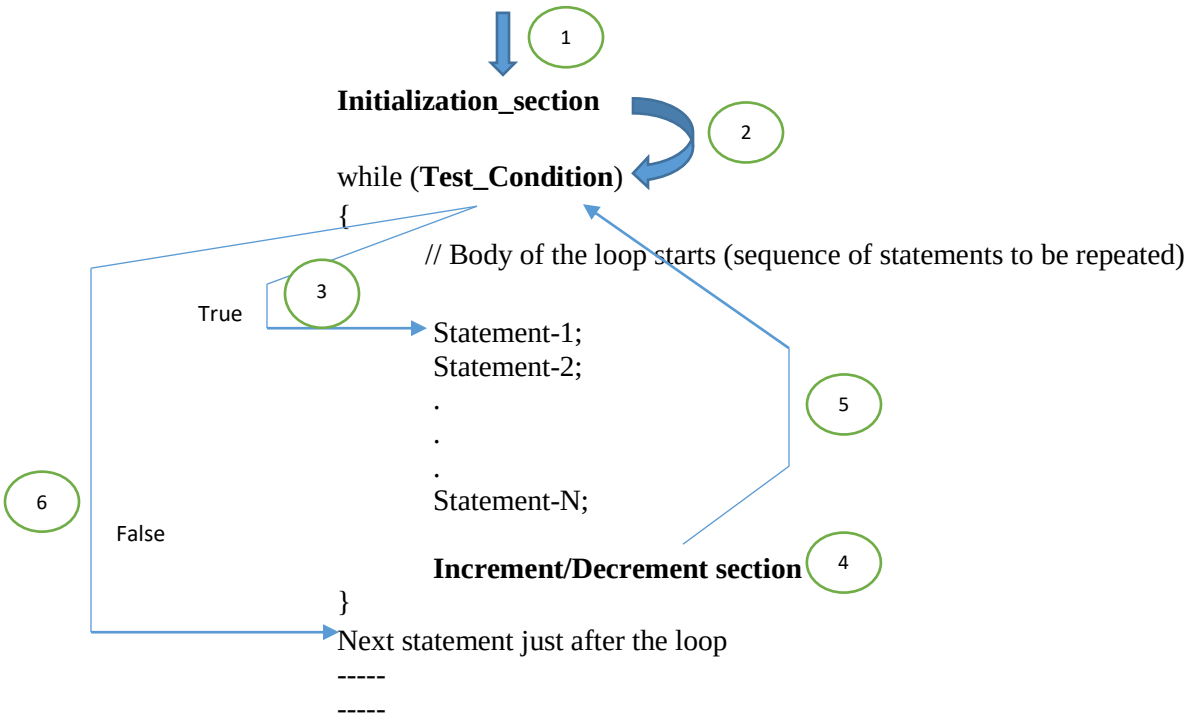
```
#include <stdio.h>
#include<conio.h>
void main()
{
   int n, count, sum = 0;
```

```
    printf("Enter a positive integer: ");
    scanf("%d", &n);

    // for loop terminates when the value of count becomes larger than n
    for(count = 1; count <= n; count++)
    {
       sum = sum + count;
    }
    printf("Sum of n natural numbers is = %d", sum);
    getch();
}
```

Output:
Enter a positive integer: 5
Sum of n natural numbers is = 15

2. <u>while loop</u>: Used usually where number of iterations are already not known. That's why it is also called as non-deterministic loops.

The syntax of a while loop is:



<u>Working of while Loop:</u>

Step 1: The initialization statement is executed only once.

Step 2: Then, the test expression is evaluated.

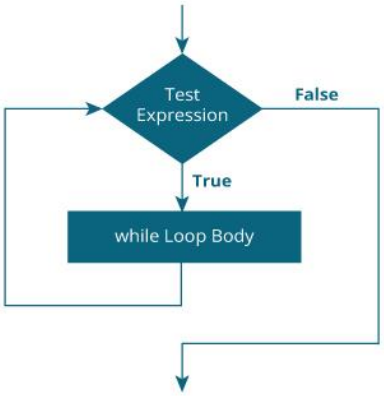
Step 3: If the test expression is true (i.e. 1), program control enters the very first statement inside the body of while loop and then executes it.

Step 4: After executing the last statement inside the while loop, program control goes back to the increment/decrement section as shown in above figure.

Step 5: After, performing increment or decrement, program control again jumps to the Testing section. This process is repeated till the test condition remains true.

Step 6: If the test expression is false (i.e. 0), while loop is terminated and program control jumps to the next statement just after the loop.

Flowchart of 'while' loop



Example program of while loop

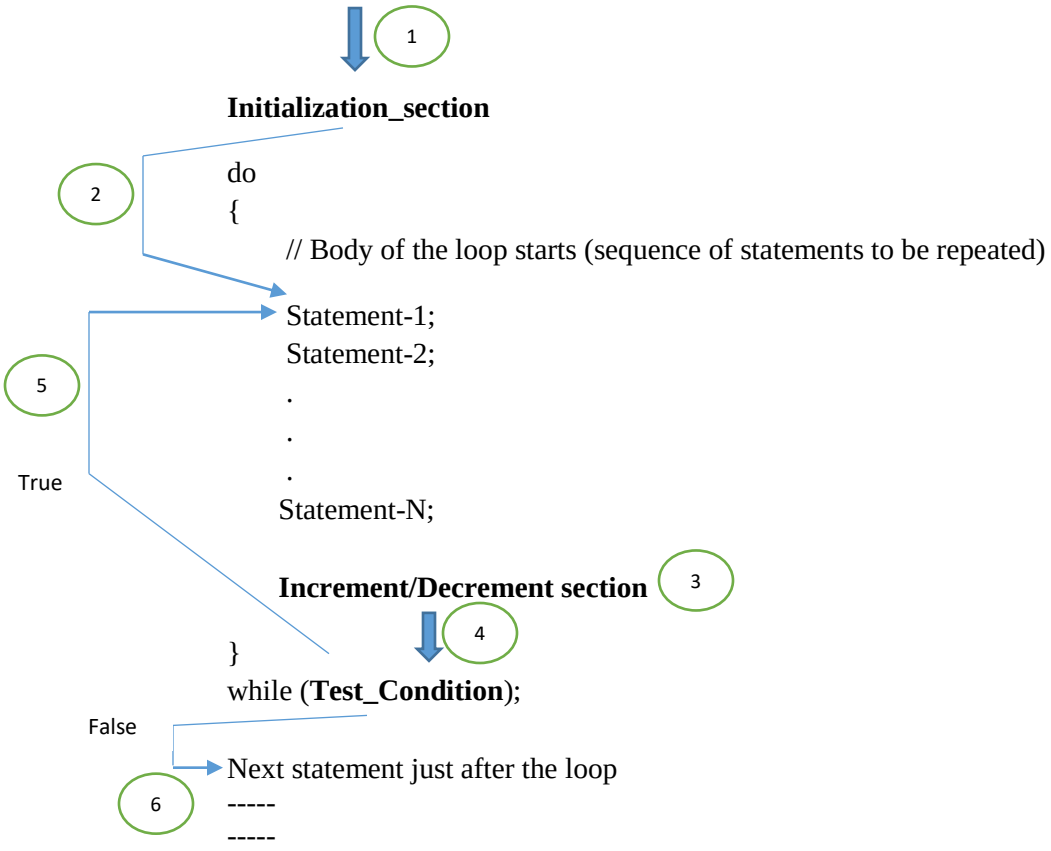
```
// Program to find factorial of a number
// For a positive integer n, factorial = 1*2*3...n

#include<stdio.h>
#include<conio.h>
void main()
{

   int n, f;

   printf("Enter a positive integer n: ");
   scanf("%d",&n);

   f = 1;
```

```
    // loop terminates when number is less than or equal to 0
    while (n > 0)
    {

        f = f *n;
        n = n-1;                // Decrementing value of n
    }

    printf("Factorial of input number is = %d", f);
    getch();
}
```

Output:
Enter a positive integer: 4
Factorial of input number is = 24

3. <u>do-while loop</u>: The do-while loop is similar to the while loop with one important difference. The body of do-while loop is executed once and after that the Test_Condition is checked. Hence, body of the do-while loop is executed at least once.

The syntax of do-while loop is:



Working of do-while Loop:

Step 1: The initialization statement is executed only once.

Step 2: Program control enters the very first statement inside the body of do-while loop and executes it.

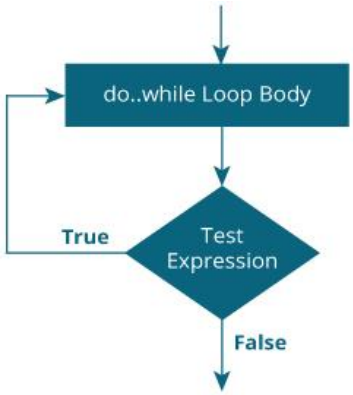
Step 3: After executing the last statement inside the do-while loop, program control goes to the increment/decrement section as shown in above figure.

Step 4: Then, the test condition is evaluated.

Step 5: If the test expression is true (i.e. 1), program control again jumps to the very first statement inside the body of the loop then executes it.

Step 6: If the test expression is false (i.e. 0), do-while loop is terminated and program control jumps to the next statement just after the for loop.

Flowchart of 'do-while' Loop



Example program of do-while loop

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int n, count, sum = 0;

   printf("Enter a positive integer n: ");
   scanf("%d",&n);
```

```
  // loop terminates when number is less than or equal to 0

   do
   {
      sum = sum + n;
      n = n-1;          // Decrementing value of n
   }while(n>0);       // Loop terminates when the value of n becomes less than 1

   printf("Sum of n natural numbers is = %d", sum);
   getch();
}
```

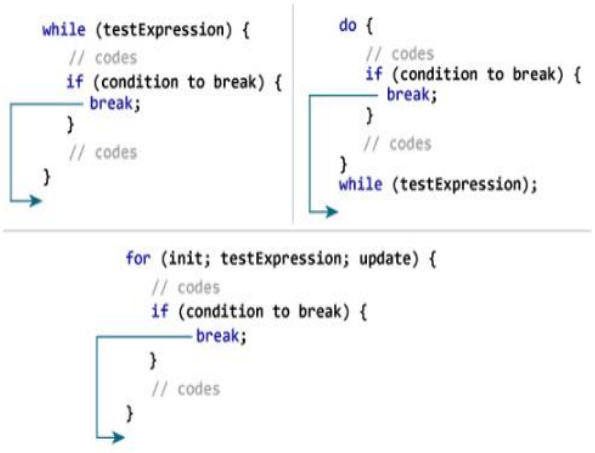
Output:
Enter a positive integer n: 6
Sum of n natural numbers is = 21

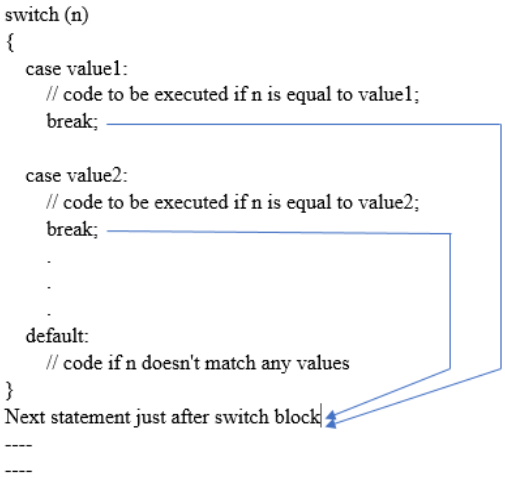
Use of Break and Continue Statements

1. ‘break’ statement: The ‘break’ statements in C language causes the termination of enclosing (within which ‘break’ is used) loop or switch-block immediately. As a break statement is encountered the program control exits from the loop (or switch-block) and jumps to the immediate next statement outside the block. In real, the break is used in:

- switch, where we want to exit after a particular case is executed
- loops, where it is required to exit the loop when certain condition occurs such as when an error is detected, etc.

Jump of program control while when using ‘break’ in loops and switch-case





Example Program: Following program calculates the sum of any 10 numbers entered by the user. If a negative number is entered, loop terminates and sum is displayed

```
# include <stdio.h>
# include <conio.h>
void main()
{
   int i;
   float number, sum = 0.0;

   for(i=1; i<= 10; i++)
   {
      printf("Enter a number: ");
      scanf("%f", &number);

      // If user enters negative number, loop is terminated
      if(number < 0.0)
      {
         break;
      }
       sum = sum + number;
   }
   printf("Sum of numbers is = %f", sum);

   getch();
}
```

Output:

```
Enter a number: 1.2
Enter a number:  2.5
Enter a number:  3.4
Enter a number: -3
Sum of numbers is = 7.10
```

2. ‘continue’ statement: The ‘continue’ statement causes the next iteration of enclosing for, while, or do-while loop to begin. The continue statement is used when we want to skip one or more statements in loop's body and to transfer the control to the next iteration. The continue statement in:

- while and do-while loops take the control to the loop's test-condition immediately
- for loop it takes the control to the increment step of the loop.

NOTE: The continue statement applies only to loops, not to switch.

Jump of program control while when using 'continue'

Example Program: Following program calculates the sum of first 10 positive integers excepting value 5.

```
# include <stdio.h>
# include <conio.h>
void main()
{
   int i;
   float sum = 0.0;

   for(i=1; i<= 10; i++)
   {
      // When value of i becomes 5, sum operation is skipped and next iteration of the loop starts
      if(i == 5)
      {
         continue;
      }
      sum = sum + i;
   }
   printf("Sum of 10 integers excepting value 5 is = %f", sum);

   getch();
}
```

Output:

Sum of 10 integers excepting value 5 is = 50.00

Difference Between break and continue

break	continue
'break' can appear in both; 'switch-case' and loops (for, while, do)	A continue can appear only in loop (for, while, do)
An occurrence of break causes the switch or loop to terminate	An occurrence of continue causes to skip the subsequent statements that appear after the continue and then loop goes to the next iteration.
When a break statement is encountered, it takes the control out of the switch or loop.	When a continue statement is encountered, it takes the control to the next iteration of the loop.

Functions in C-Language

A function is a group of statements that is written to perform a certain task. Every C program has at least one function, i.e. main(). Apart from this, a C-program may have many more functions as well. There are two classes of functions:

- ✓ **In-built functions:** The C standard library provides numerous built-in functions that your program can call. For example, printf() to print something to standard output, strcat() to concatenate two strings, etc. These prototypes for these functions are given in respective header files (files with .h extension) and their actual definition resides in library files (files with .lib extensions)

- ✓ **User Defined Functions:** A user defined function is written in the program to perform some task. These functions are written by the programmer as per the requirement of the program.

Advantages of using User Defined Functions

- It provides modular programming: The main problem is divided into smaller functions and each may be coded separately. Further all functions are combined to solve the main problem.
- Programming becomes easier to write and understand
- It becomes easy to isolate faulty function for further investigations
- Error tracking, debugging and maintenance of programs becomes easier
- It enables reusability of functions by other programs too

Defining and using User Defined Functions in C

Any function that is written and used in C programming must have following three entities:

1) **Function declaration/prototyping:** A function declaration tells the compiler about:
   - ✓ **Function's name:** It's a user defined name (identifier) given to a function.
   - ✓ **Return type:** It specifies the data type of the value returned by the function. If function does not return any value, then its return type would be given as 'void'
   - ✓ **Parameters/Arguments:** It specifies the number of arguments along with their data type. Each parameter in the list is separated by a comma.

   Syntax: ***return_type function_name(parameter_list);***

Example: Consider a function named 'sum' which takes two integer values as input and returns the addition of the input values (which would obviously be an integer). The prototype of the function will be given as following, where x and y are parameter names:

```
int sum(int x, int y);
or  int sum(int, int);
```

NOTE:

- ✓ Parameter names are optional in declaration.
- ✓ A declaration or prototyping must end with a semi-colon.

2) **Function Definition:** In function definition, the actual body of the function (sequence of statements) is provided.

Syntax:

```
return_type  function_name(parameter_list)
{
    // Body of the function
    return <value>;
}
```

Example: Consider the definition of 'sum' function:

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

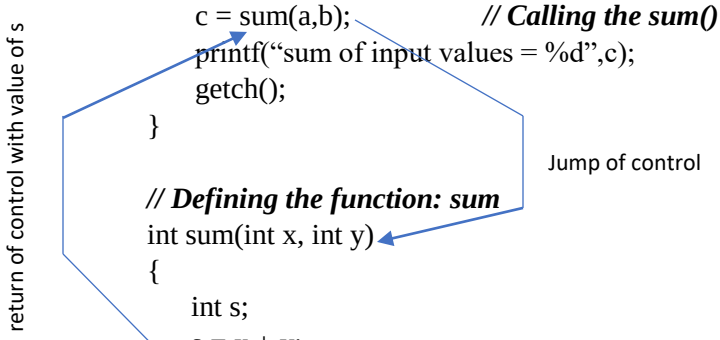
3) **Function Calling/ Invoking:** A function is called when that function is required to be used (by same or other function)

Syntax: *function_name(actual_parameter_list);*

Example: *sum(10,20);*

Example Program: Now, combining all above to make a program having sum() function. Here, main() is the *calling* function and sum() is the *called* function.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int sum(int,int);          // Function declaration (locally) of sum
    int a, b, c;
    printf("Enter value of a and b");
    scanf("%d%d",&a,&b);
    c = sum(a,b);              // Calling the sum() function and passing parameters to it
    printf("sum of input values = %d",c);
    getch();
}

// Defining the function: sum
int sum(int x, int y)
{
    int s;
    s = x + y;
    return s;
}
```



Parameter Passing: In the above program, the sum() function takes two input values x and y. The actual parameters 'a' and 'b' are passed to sum(), from the main() function during the function calling. The values of 'a' and 'b' are assigned to Formal parameters 'x' and 'y' respectively.

Types of functions based on return statement and parameter passing:

Based on return statement and parameter passing, the functions are classified into following four categories:

1. Pass – Return: In this category, parameters are passed to the function as well as function returns some value. Example prototype of sum() function under this type is as follows:

Example: *int sum(int, int);*

2. No Pass – Return: In this category, parameters are not passed to the function but function returns some value. Example prototype of sum() function under this type is as follows:

Example: *int sum(void);*

3. Pass – No Return: In this category, parameters are passed to the function but function does not return any value. Example prototype of sum() function under this type is as follows:

Example: *void sum(int, int);*

4. No Pass – No Return: In this category, parameters are not passed to the function as well as function does not return any value**.** Example prototype of sum() function under this type is as follows:

Example: *void sum(void);*

Example program using function sum(), for each category

Pass – Return	**No Pass – Return**	**Pass – No Return**	**No Pass – No Return**
#include<stdio.h> #include<conio.h> void main(){ int sum(int, int); int a, b, c; printf("Enter a and b"); scanf("%d%d",&a,&b); c = sum(a,b); printf("sum = %d",c); getch(); } **// Defining function: sum** int sum(int x, int y) { int s; s = x + y; return s; }	#include<stdio.h> #include<conio.h> void main(){ int sum(void); int c; c = sum(); printf("sum = %d",c); getch(); } **// Defining function: sum** int sum(void) { int a, b, s; printf("Enter a and b"); scanf("%d%d",&a,&b); s = x + y; return s; }	#include<stdio.h> #include<conio.h> void main(){ void sum(int, int); int a, b; printf("Enter a and b"); scanf("%d%d",&a,&b); sum(a,b); getch(); } **// Defining the function: sum** void sum(int x, int y) { int s; s = x + y; printf("sum = %d",s); }	#include<stdio.h> #include<conio.h> void main(){ void sum(void); sum(); getch(); } **// Defining function: sum** void sum(void) { int a,b,s; printf("Enter a and b"); scanf("%d%d",&a,&b); s = x + y; printf("sum = %d",s); }

Recursion

In programming languages, if a function calls itself inside its own body, then it is called a recursive call of the function and this process is called recursion.

```
void myfunc()
{
        myfunc();     // Function myfunc() calling itself, therefore, it is a recursive call
}
```

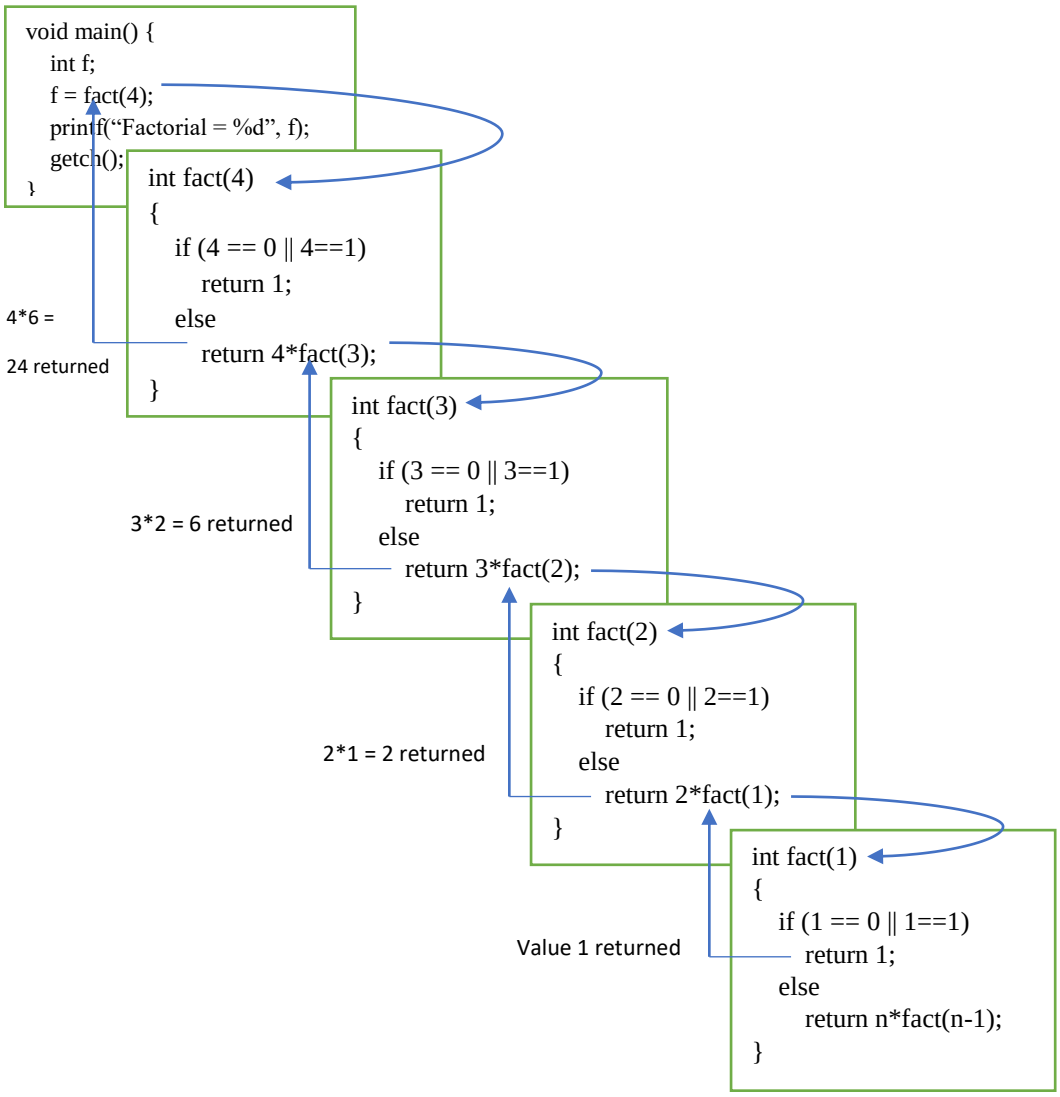
Rules/Laws of recursive algorithms:

- A recursive algorithm must have a base case.
  (**Base case:** is the condition where recursion finds concrete solution corresponding to the input and then terminates)
- A recursive algorithm must change its state and move toward the base case.
- A recursive algorithm must call itself, recursively.

Example Program: Following is the program having a recursive function fact(), that takes a non-negative integer as input and returns its factorial value.

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

// Recursive definition of function: fact()
int fact(int n)
{
   if (n == 0 || n==1)       // Base case: on input 0 or 1, function returns concrete solution 1
      return 1;
   else
      return n*fact(n-1);
}


// Start of main function
void main()
{
   int n, f;
   printf("Enter a non-negative integer value n:");
   scanf("%d",&n);
   f = fact(n);          // calling the function: fact()
   printf("Factorial of input integer = %d", f);
   getch();
}
```

Execution Sequence of recursive function fact() with value passed as 4, i.e. fact(4)



Advantages of recursion

- Recursion provides a clean and simple way to write code.
- Using recursion many problems can be solved in easier way as compared to iterative solution which is bigger and complex. For e.g. problems like:
    - Tree traversal
    - Tower of Hanoi
    - Merge sorting
    - Quick sorting, etc. can be easily and elegantly solved by using recursive algorithm.

Disadvantages of recursion

- Recursive programs consume more memory space than iterative programs as all functions will remain in memory stack until base case is reached.
- Recursive programs have greater run-time than iterative programs because of function calls and return overhead.

Linear, Binary and Multiple Recursion: Based on number of recursive calls inside a function, recursion can be of following three types:

- **Linear Recursion:** If a function contains one recursive call to itself, then it is Linear Recursion.
- **Binary Recursion:** If a function contains two recursive calls to itself, then it is Binary Recursion.
- **Multiple Recursion:** If a function contains more than two recursive calls to itself, then it is Multiple Recursion.

Example of each type of Recursion

Linear Recursion	// Recursive function definition to calculate factorial int fact(int n) { if (n == 0 || n==1) return 1; else return n*fact(n-1); // Only one recursive call to function fact() }
Binary Recursion	// Recursive function definition to calculate Fibonacci number int fib(int n) { if (n == 0 || n==1) return n; else return (fib(n-1) + fib(n-2)); // Two recursive calls to function fib() }
Multiple Recursion	// Recursive function definition of some function fun() int fun(int n) { if (n==0 || n==1 || n==2) return 1; else return (fun(n-1) + fun(n-2) * fun(n-3)); //Three recursive calls to function fun() }