# Functions in C

- In C, we can divide a large program into the basic building blocks known as function.
- The function contains the set of programming statements enclosed by {}.
- A function can be called multiple times to provide reusability and modularity to the C program.
- In other words, we can say that the collection of functions creates a program.
- The function is also known as *procedure* or *subroutine* in other programming languages.

# Functions in C

## Advantage of functions in C

- By using functions, we can avoid rewriting same logic/code again and again in a program.
- We can call C functions any number of times in a program and from any place in a program.
- We can track a large C program easily when it is divided into multiple functions.
- Reusability is the main achievement of C functions.
- However, Function calling is always a overhead in a C program.

# Functions in C

## Function Aspects

- **Function declaration** A function must be declared globally in a c program to tell the compiler about the function name, function parameters, and return type.
- **Function call** Function can be called from anywhere in the program. The parameter list must not differ in function calling and function declaration. We must pass the same number of functions as it is declared in the function declaration.
- **Function definition** It contains the actual statements which are to be executed. It is the most important aspect to which the control comes when the function is called. Here, we must notice that only one value can be returned from the function.

# Functions in C

| S.No. | Function Aspects | Syntax |
|---|---|---|
| 1 | Function declaration | return_type function_name (argument list); |
| 2 | Function call | function_name (argument_list) |
| 3 | Function definition | return_type function_name (argument list)<br>{<br>function body;<br>} |

# Functions in C

The **syntax** of creating function in c language is given below:

```
return_type function_name(data_type parameter...)
{
//code to be executed
}
```

## Types of Functions

There are two types of functions in C programming:

- **Library Functions**: are the functions which are declared in the C header files such as scanf(), printf(), gets(), puts(), ceil(), floor() etc.
- **User-defined functions**: are the functions which are created by the C programmer, so that he/she can use it many times. It reduces the complexity of a big program and optimizes the code.

# Functions in C

## Return Value

- A C function may or may not return a value from the function. If you don't have to return any value from the function, use void for the return type.

**Example without return value:**

```
void hello()
{
   printf("hello c");
}
```

- If you want to return any value from the function, you need to use any data type such as int, long, char, etc. The return type depends on the value to be returned from the function.

# Functions in C

**Example with return value:**

```
int get()
{
    return 10;    // function that returns int value from the function
}
```

In the above example, we have to return 10 as a value, so the return type is int. If you want to return floating-point value (e.g., 10.2, 3.1, 54.5, etc), you need to use float as the return type of the method.

```
float get()
{
    return 10.2;
}
```

Now, you need to call the function, to get the value of the function.

# Functions in C

## Different aspects of function calling

- A function may or may not accept any argument. It may or may not return any value. Based on these facts, there are four different aspects of function calls.
    - ✓ function without arguments and without return value
    - ✓ function without arguments and with return value
    - ✓ function with arguments and without return value
    - ✓ function with arguments and with return value

# Example for Function without argument and return value

```
#include<stdio.h>
void printName();  // Function declaration
void main ()
{
        printf("Hello ");
        printName();  // Function call
}
void printName()  // Function definition
{
        printf("Javatpoint");
}
```

# Example for Function without argument and return value

```
#include<stdio.h>
void sum();   // Function declaration
void main()
{
        printf("\nGoing to calculate the sum of two numbers:");
        sum();  // Function call
}
void sum()   // Function definition
{
        int a,b;
        printf("\nEnter two numbers");
        scanf("%d %d",&a,&b);
        printf("The sum is %d",a+b);
}
```

# Example for Function without argument and with return value

```
#include<stdio.h>
int sum();   // Function declaration
void main()
{
    int result;
    printf("\nGoing to calculate the sum of two numbers:");
    result = sum();   // Function call
    printf("%d",result);
}
int sum()   // Function definition
{
    int a,b;
    printf("\nEnter two numbers");
    scanf("%d %d",&a,&b);
    return a+b;
}
```

## Example for Function without argument and with return value

```
#include<stdio.h>
int sum();   // Function declaration
void main()
{
        printf("Going to calculate the area of the square\n");
        float area = square();   // Function call
        printf("The area of the square: %f\n",area);
}
int square()   // Function definition
{
        float side;
        printf("Enter the length of the side in meters: ");
        scanf("%f",&side);
        return side * side;
}
```

# Example for Function with argument and without return value

```
#include<stdio.h>
void sum(int, int);   // Function declaration
void main()
{
        int a,b,result;
        printf("\nGoing to calculate the sum of two numbers:");
        printf("\nEnter two numbers:");
        scanf("%d %d",&a,&b);
        sum(a,b);   // Function call
}
void sum(int a, int b)   // Function definition
{
        printf("\nThe sum is %d",a+b);
}
```

# Example for Function with argument and without return value

```
#include<stdio.h>
void average(int, int, int, int, int);   // Function declaration
void main()
{
        int a,b,c,d,e;
        printf("\nGoing to calculate the average of five numbers:");
        printf("\nEnter five numbers:");
        scanf("%d %d %d %d %d",&a,&b,&c,&d,&e);
        average(a,b,c,d,e);   // Function call
}
void average(int a, int b, int c, int d, int e)   // Function definition
{
        float avg;
        avg = (a+b+c+d+e)/5;
        printf("The average of given five numbers : %f",avg);
}
```

# Example for Function with argument and with return value

```
#include<stdio.h>
int sum(int, int);   // Function declaration
void main()
{
        int a,b,result;
        printf("\nGoing to calculate the sum of two numbers:");
        printf("\nEnter two numbers:");
        scanf("%d %d",&a,&b);
        result = sum(a,b);   // Function call
        printf("\nThe sum is : %d",result);
}
int sum(int a, int b)   // Function definition
{
        return a+b;
}
```

# Example for Function with argument and with return value

```
#include<stdio.h>
int even_odd(int);   // Function declaration
void main()
{
        int n,flag=0;
        printf("\nGoing to check whether a number is even or odd");
        printf("\nEnter the number: ");
        scanf("%d",&n);
        flag = even_odd(n);   // Function call
        if(flag == 0)
        {
                printf("\nThe number is odd");
        }
        else
        {
                printf("\nThe number is even");
        }
}
```

# Example for Function with argument and with return value

```
int even_odd(int n)   // Function definition
{
        if(n%2 == 0)
        {
                return 1;
        }
        else
        {
                return 0;
        }
}
```

# Call by value and Call by reference in C

- There are two methods to pass the data into the function in C language, i.e., *call by value* and *call by reference*.

## Call by value in C

- In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
- In call by value method, we cannot modify the value of the actual parameter by the formal parameter.
- In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
- The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

# Call by value-Example 1

```
#include<stdio.h>
void change(int num)
{
      printf("Before adding value inside function num=%d \n",num);
      num=num+100;
      printf("After adding value inside function num=%d \n", num);
}
int main()
{
      int x=100;
      printf("Before function call x=%d \n", x);
      change(x); //passing value in function
      printf("After function call x=%d \n", x);
   return 0;
}
```

**Output**

Before function call x=100

Before adding value inside function num=100

After adding value inside function num=200

After function call x=100

# *Example 2: Swapping the values of the two variables*

```
#include <stdio.h>
void swap(int , int); //prototype of the function
 int main()
 {
    int a = 10;
    int b = 20;
    printf("Before swapping the values in main a = %d, b = %d\n",a,b);
    // printing the value of a and b in main
    swap(a,b);   // a and b is Actual Parameters
    printf("After swapping values in main a = %d, b = %d\n",a,b);
    //The value of actual parameters do not change by changing the for
    mal parameters in call by value, a = 10, b = 20
}
```

## *Example 2: Swapping the values of the two variables*

```
void swap (int a, int b)  // a and b is Formal Parameters
{
    int temp;
    temp = a;
    a=b;
    b=temp;
    printf("After swapping values in function a = %d, b = %d\n",a,b);
    // Formal parameters, a = 20, b = 10
}
```

**Output**

Before swapping the values in main a = 10, b = 20

After swapping values in function a = 20, b = 10

After swapping values in main a = 10, b = 20

# Call by reference in C

- In call by reference, the address of the variable is passed into the function call as the actual parameter.
- The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
- In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

# Call by reference-Example 1

```
#include<stdio.h>
void change(int *num)
{
    printf("Before adding value inside function num=%d \n",*num);
    (*num) += 100;
    printf("After adding value inside function num=%d \n", *num);
}
int main()
{
    int x=100;
    printf("Before function call x=%d \n", x);
    change(&x);//passing reference in function
    printf("After function call x=%d \n", x);
    return 0;
}
```

**Output**

Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=200

# *Example 2: Swapping the values of the two variables*

```
#include <stdio.h>
void swap(int *, int *); //prototype of the function
int main()
{
   int a = 10;
   int b = 20;
   printf("Before swapping the values in main a = %d, b = %d\n",a,b);
   // printing the value of a and b in main
   swap(&a,&b);
   printf("After swapping values in main a = %d, b = %d\n",a,b);
   //The values of actual parameters do change in call by reference, a =
   10, b = 20
}
```

## *Example 2: Swapping the values of the two variables*

```
void swap (int *a, int *b)
{
   int temp;
   temp = *a;
   *a=*b;
   *b=temp;
   printf("After swapping values in function a = %d, b = %d\n",*a,*b);
   // Formal parameters, a = 20, b = 10
}
```

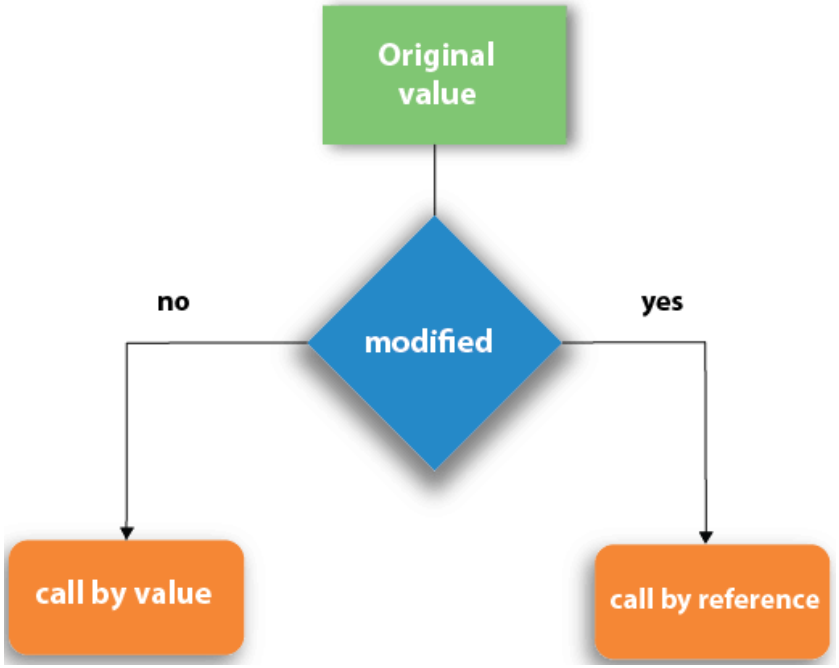
**Output**

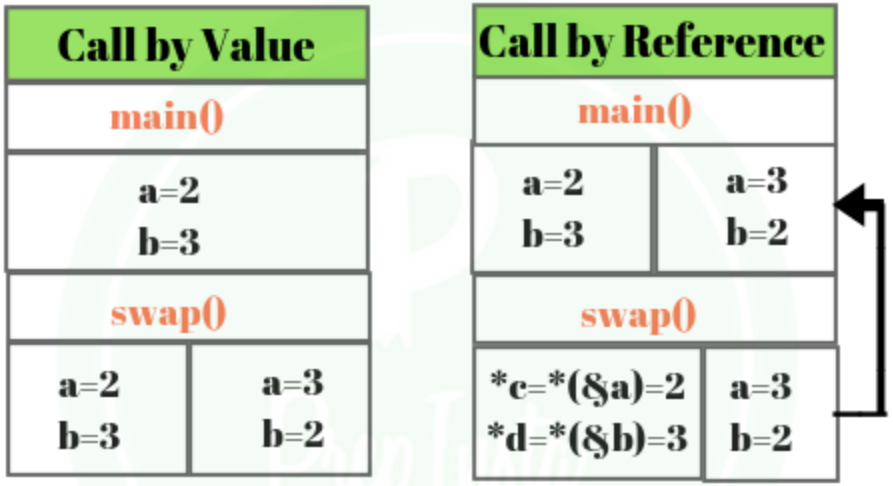
Before swapping the values in main a = 10, b = 20

After swapping values in function a = 20, b = 10

After swapping values in main a = 20, b = 10

# Call by Value and Call by Reference



# Difference between call by value and call by reference in C

| S.No. | Call by value | Call by reference |
|---|---|---|
| 1 | A copy of the value is passed into the function | An address of value is passed into the function |
| 2 | Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters. | Changes made inside the function validate outside of the function also. The values of the actual parameters do change by changing the formal parameters. |
| 3 | Actual and formal arguments are created at the different memory location | Actual and formal arguments are created at the same memory location |

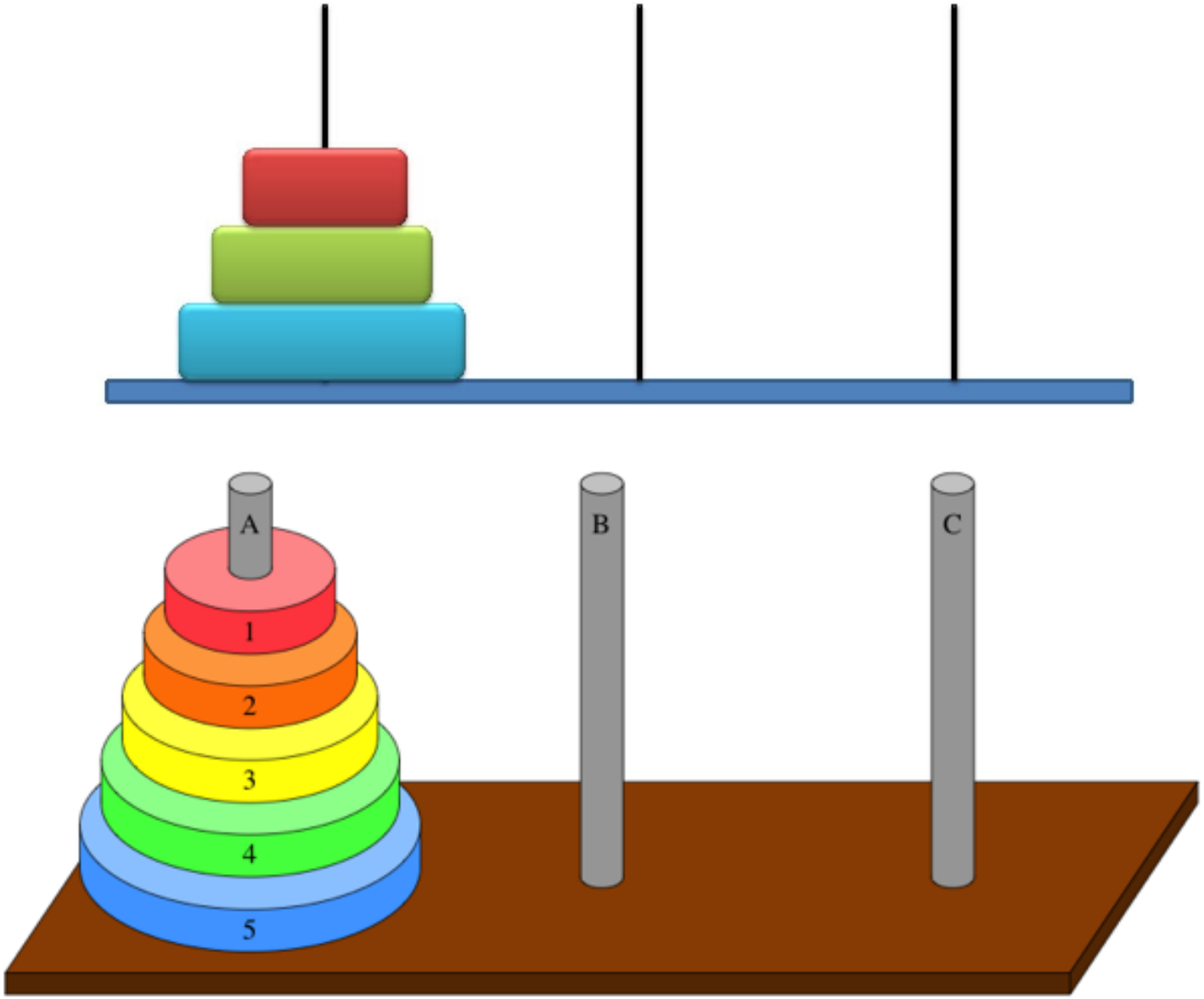
# Recursion in C

- A function that calls itself is known as a recursive function, and such function calls are called recursive calls. And, this technique is known as recursion. Recursion involves several numbers of recursive calls.
- For Example, recursion may be applied to sorting, searching, and traversal problems.
- Generally, iterative solutions are more efficient than recursion since function call is always overhead. Any problem that can be solved recursively, can also be solved iteratively. However, some problems are best suited to be solved by the recursion, for example, tower of Hanoi, Fibonacci series, factorial finding, etc.

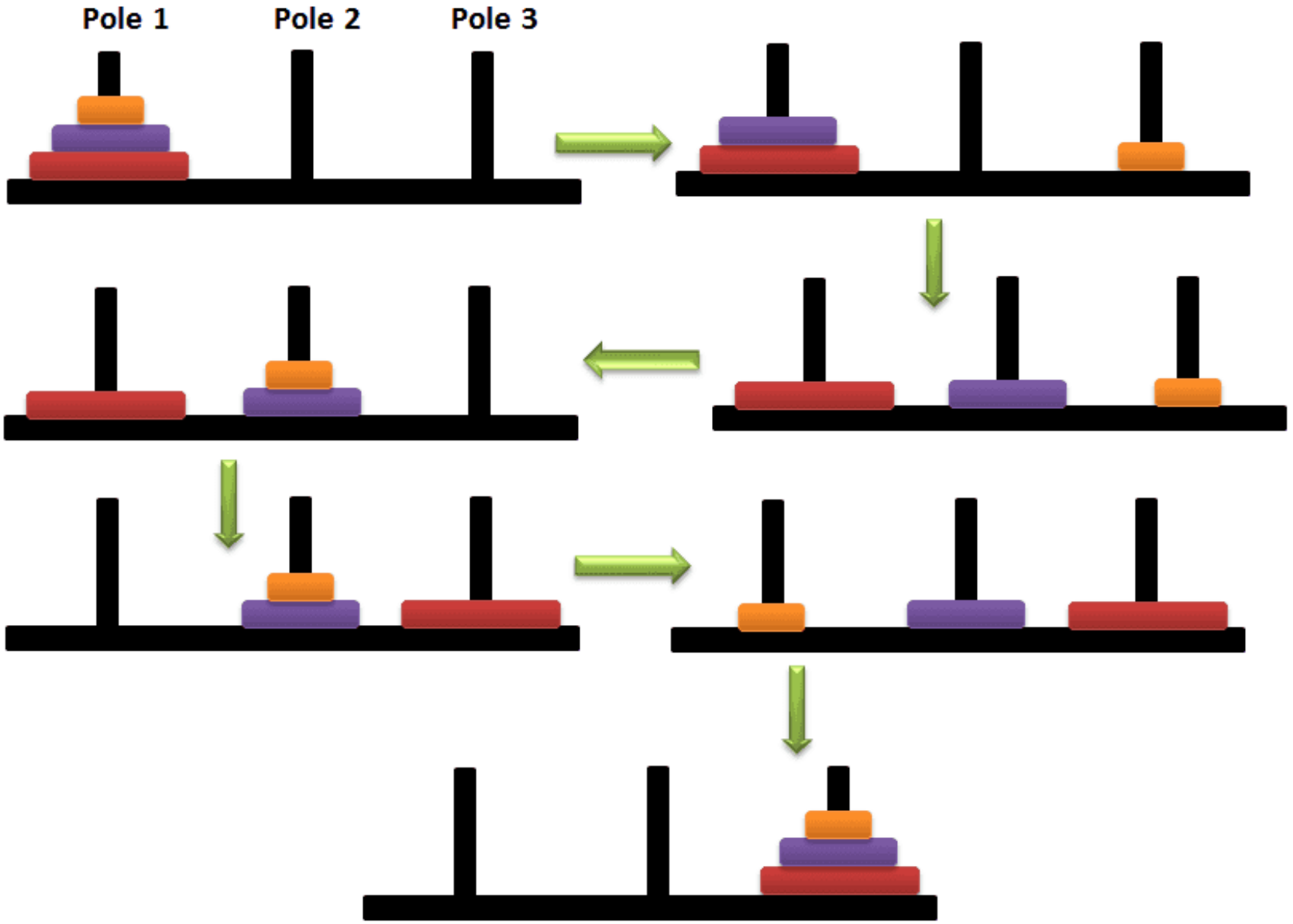
# Recursion in C

- A function that calls itself is known as a recursive function, and such function calls are called recursive calls. And, this technique is known as recursion. Recursion involves several numbers of recursive calls.
- For Example, recursion may be applied to sorting, searching, and traversal problems.
- Generally, iterative solutions are more efficient than recursion since function call is always overhead. Any problem that can be solved recursively, can also be solved iteratively. However, some problems are best suited to be solved by the recursion, for example, tower of Hanoi, Fibonacci series, factorial finding, etc.

# Tower of Hanoi



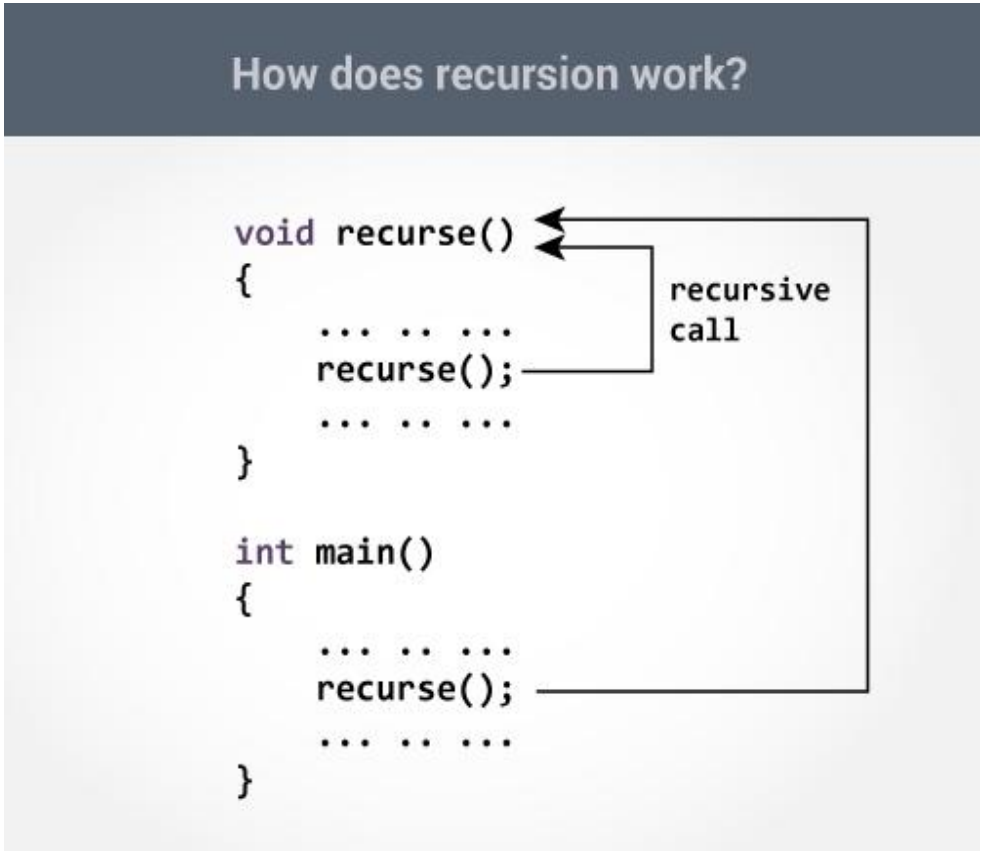
# Tower of Hanoi



# Recursive Function

**Pseudocode for writing any recursive function is given below.**

```
if (test_for_base)
{
   return some_value;
}
else if (test_for_another_base)
{
   return some_another_value;
}
else
{
   // Statements;
   recursive call;
}
```



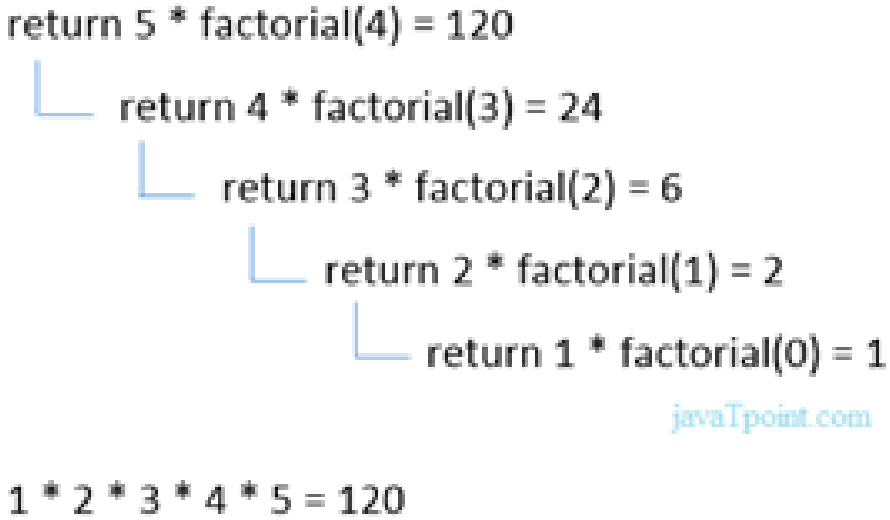
# Example 1- Calculate the factorial of a number

```
#include <stdio.h>
int fact (int);
int main()
{
        int n,f;
        printf("Enter the number to calculate factorial :");
        scanf("%d",&n);
        f = fact(n);
        printf("factorial = %d",f);
}
int fact(int n)
{
        if (n==0)
        {
                return 0;
        }
        else if ( n == 1)
        {
                return 1;
        }
        else
        {
                return n*fact(n-1);
        }
}
```

**Output**

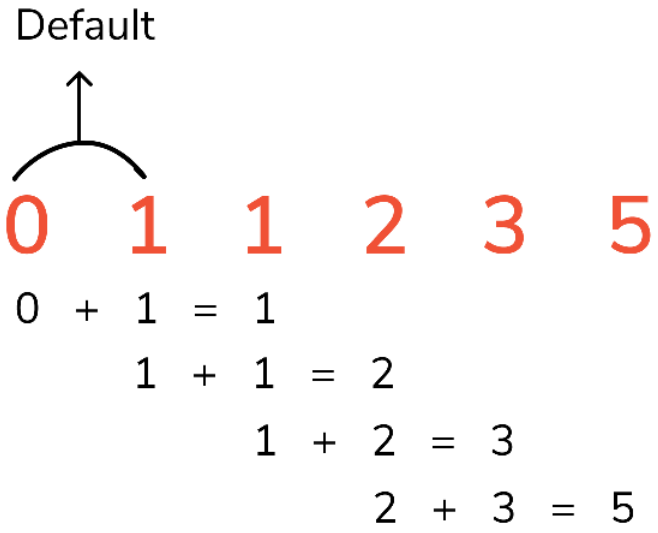
Enter the number to calculate factorial: 5

Factorial = 120



# Example 2-Find the nth term of the Fibonacci series

```
#include<stdio.h>
int fibonacci(int);
void main ()
{
      int n,f;
      printf("Enter the value of n?");
      scanf("%d",&n);
      f = fibonacci(n);
      printf("%d",f);
}
int fibonacci (int n)
{
      if (n==0)
      {
            return 0;
      }
      else if (n == 1)
      {
                  return 1;
      }
      else
      {
                  return fibonacci(n-1)+fibonacci(n-2);
      }
}
```




**Output**

Enter the value of n? 12

144

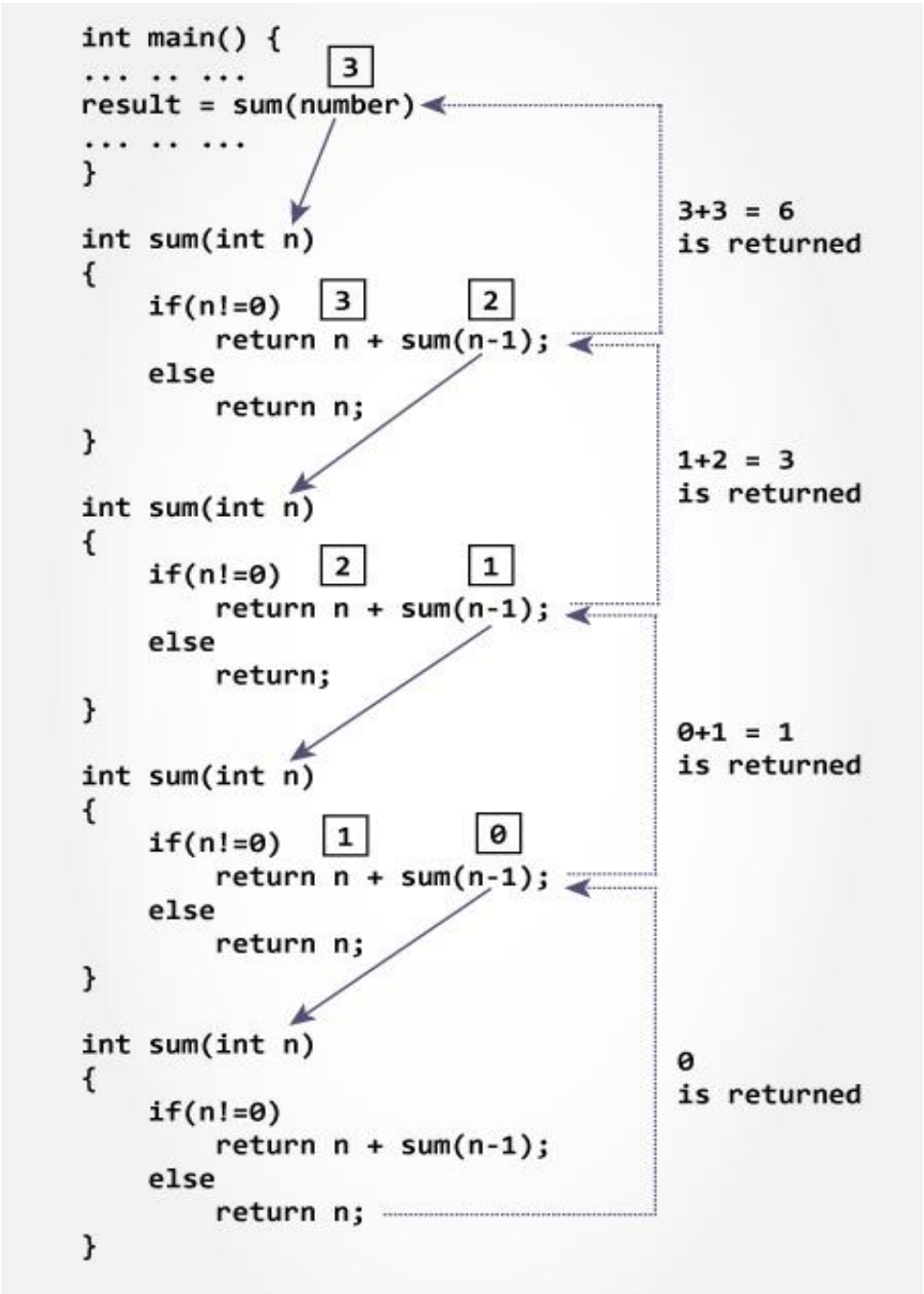
# Example 3- Sum of Natural Numbers Using Recursion

```
#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 num)
{
        if (num!=0)
        return num + sum(num-1); // sum() function calls itself
        else
        return num;
}
```

**Output**

Enter a positive integer:10

Sum= 55


int main() {
... .. ...
result = sum(number)
... .. ...
}
3
int sum(int n)
{
if(n!=0)
return n + sum(n-1);
else
return n;
}
3
2
3+3 = 6
is returned
int sum(int n)
{
if(n!=0)
return n + sum(n-1);
else
return;
}
2
1
1+2 = 3
is returned
int sum(int n)
{
if(n!=0)
return n + sum(n-1);
else
return n;
}
1
0
0+1 = 1
is returned
int sum(int n)
{
if(n!=0)
return n + sum(n-1);
else
return n;
}
0
is returned