# Functions

# Introduction

- During the 1970s and into the 80s, the primary software engineering methodology was *structured programming.*
- Structured programming makes
  - Programs more comprehensible.
  - Programming errors less frequent.
- A *function is a self-contained block of program statements* that performs a particular task.

# Why are Functions Needed?

- It breaks up a program into easily manageable chunks and makes programs significantly easier to understand.
- Well written functions may be reused in multiple programs. eg. The C standard library functions.
- Functions can be used to protect data.
- Different programmers working on one large project can divide the workload by writing different functions.

# Why are Functions Needed?

- All C programs contain at least one function, called main() where execution starts.
- When a function is called, the code contained in that function is executed.
- When the function has finished executing, control returns to the point at which that function was called.

# Function declaration an definition

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 Prototype Declaration

- The general form of function declaration statement is as follows:

return_data_type function_name (data_type variable1,...);

Or

return_data_type function_name (data_type_list);

# Function Prototype Declaration

return_data_type  function_name (data_type variable1,...);

- function_name :
  - This is the name given to the function
  - it follows the same naming rules as that for any valid variable in C.
- return_data_type:
  - This specifies the type of data given back to the calling construct by the function after it executes its specific task.
- data_type_list(parameters):
  - This list specifies the data type of each of the variables.

# Function Example

```
/* function returning the max between two
   numbers */
 int max(int num1, int num2)
{ /* local variable declaration */
 int result;
if (num1 > num2)
result = num1;
else result = num2;
return result;
}
```

```
#include <stdio.h>
/* function declaration */
int max(int num1, int num2);
int main ()
{ /* local variable definition */
int a = 100;
int b = 200;
int ret;
/* calling a function to get max value */
ret = max(a, b);
printf( "Max value is : %d\n", ret );
return 0; }

/* function body*/
 int max(int num1, int num2)
{ /* local variable declaration */
int result;
if (num1 > num2)
      result = num1;
else
     result = num2;
return result; }
```

Main

Function Body

# Actual and Formal parameters

Actual **Parameters** or Arguments:

When a function is called, the values (expressions) that are passed in the call are called the arguments or actual **parameters**.

At the time of the call each actual **parameter** is assigned to the corresponding **formal parameter** in the function definition.

Parameter Written In Function Call is Called "Actual Parameter"

Parameter Written In Function Definition is Called "Formal Parameter"

- Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

```
int main ()
{
..
ret = max(a, b);

printf( "Max value
is : %d\n", ret );
}
```

```
int max(int num1,
int num2)
{ ....

return result; }
```

# Function Prototype Declaration

- The name of a function is global.
- No function can be defined in another function body.
- Number of arguments must agree with the number of parameters specified in the prototype.
- The function return type cannot be an array or a function type.

# Rules for Parameters

- The number of parameters in the actual and formal parameter lists must be consistent.
- Parameter association in C is *positional*.
- Actual parameters and formal parameters must be of compatible data types.
- Actual (input) parameters may be a variable, constant, or any expression matching the type of the corresponding formal parameter.

# Calling a function

While calling a function, there are two ways in which arguments can be passed to a function –

| S.N. | Call Type & Description |
|---|---|
| 1 | **Call by value** 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. |
| 2 | **Call by reference** This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument. |

# Call by Value Mechanism

- In call by value, a copy of the data is made and the copy is sent to the function.
- The copies of the value held by the arguments are passed by the function call.
- As only copies of the values held in the arguments are sent to the formal parameters, the function cannot directly modify the arguments passed.

# An Example of Call by value Mechanism:

```
#include <stdio.h>
int mul_by_10(int num); /* function prototype */
int main(void)
{
  int result,num = 3;
  printf("\n num = %d before function call.", num);
  result = mul_by_10(num);
  printf("\n  result  =  %d  after  return  from
        function", result);
  printf("\n num = %d", num);
  return 0;
}
/* function definition follows */
int mul_by_10(int num)
{
  num *= 10;
  return num;
}
```

**Output**

```
num = 3, before function call.
result = 30, after return from function.
num = 3
```

# Function Example : Factorial

```
#include <stdio.h>
long factorial(int);
int main()
{ int number;
long fact = 1;
printf("Enter a number to
   calculate it's factorial\n");
   scanf("%d", &number);

printf("%d! = %ld\n", number,
   factorial(number));
return 0;
 }
```

n! = n*(n-1)*(n-2)*(n-3)...3.2.1 and zero factorial is defined as one i.e. 0! = 1.

```
long factorial(int n)
 {
int c;
long result = 1;
for (c = 1; c <= n; c++)
result = result * c;
return result;
}
```

WAP to swap the values of two variables by using a suitable user defined function (say SWAP) for it.

## WAP to swap the values of two variables by using a suitable user defined function (say SWAP) for it.

```
void swap(int, int);

int main()
{
    int a, b;
      printf("Enter values for a and
b\n");
    scanf("%d%d", &a, &b);
      printf("\n\nBefore swapping: a
= %d and b = %d\n", a, b);

    swap(a, b);

    return 0;
}
```

```
void swap(int x, int y)
{
    int temp;

    temp = x;
    x    = y;
    y    = temp;

    printf("\nAfter swapping: a = %d
and b = %d\n", x, y);
}
```

**Find out the value of nth term of the Fibonacci sequence**

**Find out the value of nth term of the Fibonacci sequence**

**PROGRAM CODE**

```
#include<stdio.h>
int fib(int);
int main()
{
int n;
printf("\nEnter term number :");
scanf("%d",&n);
printf("\nThe value of the term-%d of Fibonacci sequence is %d.”,
n, fib(n));
return 0;
}
/*User defined iterative function
 fib that returns the value of ith
term of Fibonacci sequence*/
int fib(int n)
{
     int i,f1=0,f2=1,f;
     if (n==0 || n==1)
     return (n-1);
     else
     {
          for(i=1; i<=n-2; i++)
          {
          f=f1+f2;
          f1=f2;
          f2=f;
          }
          return f;
     }
}
```

# INPUT/OUTPUT

- **RUN-1**
- Enter term number: 6
- The value of the term-6 of Fibonacci sequence is 8.
- **RUN-2**
- Enter term number: 15
- The value of the term-5 of Fibonacci sequence is 377.

# Factorial using function

```
#include <stdio.h>
long int fact(int); //Function Prototype
int main()
{
    int n;
    printf("\nEnter a number :");
    scanf("%d",&n);
    printf("\n%d!=%ld\n", n, fact(n));
    return 0;
}
```

```
/*Factorial function*/
long int fact(int n)
{
    long int f=1;;
    int i;
    for (i=1; i<=n; i++)
    f=f*i;
    return (f);
}
```

# SUM of DIGITS

```
#include <stdio.h>
int SUM-DIGIT(int); //Function Prototype
int main()
{
int n, s;
printf("\nEnter a number :");
scanf("%d",&n);
s=SUM-DIGIT(n);
printf("\nThe sum of digits of %d is %d." n, s);
return 0;
}
```

```
/*User defined function to find out sum of digits of number n*/
int SUM-DIGIT(int n)
{
int sum=0;
while(n!=0)
{
sum=sum+n%10;
n=n/10;
}
return sum;
}
```

# Passing Arrays to Functions

- When an array is passed to a function, the **address of the array is passed and not the copy of the complete array.**
- During its execution the function has the ability to **modify the contents** of the array that is specified as the function argument.
- The array is not passed to a function by value.
- This is an exception to the rule of passing the function arguments by value.

# calculate the sum of all the integers stored in the array.

```
#include<stdio.h>
int SUM-ARRAY(int a[],int);
//Function Prototype
int main()
{
int a[100], n, i;
printf("\nEnter how many numbers :");
scanf("%d",&n);
printf("\nEnter data for array: ");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
printf("\nThe sum of the elements of
   the array is %d", SUM-ARRAY(a, n);
return 0;
}
```

```
/*User Defined Function
SUM-ARRAY*/
int SUM-ARRAY(int a[], int
n)
{
int i, sum=0;
for(i=0; i<n; i++)
{
     sum=sum + a[i];
}
return sum;
}
```

# INPUT/OUTPUT

**RUN-1**

Enter how many numbers : 4
Enter data for array: 7 6 5 4
The sum of the elements of the array is 22

**RUN-2**

Enter how many numbers : 5
Enter data for array: 1 2 3 4 5
The sum of the elements of the array is 15

# Searching an element within an array

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
 int a[30],n,i,key, FOUND=0;

 printf("\n How many numbers");

 scanf("%d",&n);
 if(n>30)
 {
  printf("\n Too many Numbers");

  exit(0);
 }
 printf("\n Enter the array elements \n");

 for(i=0 ; i<n; i++)

 scanf("%d", &a[i]);

 printf("\n Enter the key to be searched \n");

 scanf("%d",&key);

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

*For loop* ➡

```
{
if(a[i] == key)
{
  printf("\n Found at %d",i);
  FOUND=1;
}
}
if(FOUND = = 0)
printf("\n NOT FOUND...");
return 0;
}
```

# Search an element within array using function

```
#include<stdio.h>
 int linear_search(int A[], int, int);
 main()
{
   int array[100], search, c, n, position;
    printf("Enter the number of elements in
array\n");
   scanf("%d",&n);
    printf("Enter %d numbers\n", n);
    for ( c = 0 ; c < n ; c++ )
      scanf("%d",&array[c]);

   printf("Enter the number to search\n");
   scanf("%d",&search);

   position = linear_search(array, n, search);

   if ( position == -1 )
      printf("%d is not present in array.\n", search);
   else
      printf("%d is present at location %d.\n",
search, position+1);

   return 0;
}
```

```
int linear_search(int A[], int n,
int find)
{
   int c;

   for ( c = 0 ; c <= n ; c++ )
   {
      if ( A[c] == find )
         return c;
   }

   return -1;
}
```

sort the elements of an array in ascending order by calling a sort function

```
#include<stdio.h>
void bubbleSort(int a[],int);
int main()
{
int a[100], n, i;
printf("\nEnter how
many numbers :");
scanf("%d",&n);

printf("\nEnter data for array: ");
for(i=0;i<n;i++)
     scanf("%d",&a[i]);

bubbleSort(a,n); //Function Call

printf("\nThe Numbers in
ascending order are:");
for(i=0; i<n; i++)
     printf("%d ",a[i]);
return 0;
}
```

```
/*Bubble Sort Function*/
void bubbleSort(int a[], int n)
{
int i, j, temp;
for(i=1; i<=n-1; i++)
    {
        for(j=0; j<n-i; j++)
        {
            if(a[j]>a[j+1])
            {
                temp=a[j];
                a[j]=a[j+1];
                a[j+1]=temp;
                }
        }
    }
}
```

# INPUT/OUTPUT

- **RUN-1**
- Enter how many numbers :8
- Enter data for array: 7 6 5 4 5 2 4 8
- The Numbers in ascending order are: 2 4 4 5 5 6 7 8
- **RUN-2**
- Enter how many numbers :9
- Enter data for array: 1 4 3 8 6 5 2 9 7
- The Numbers in ascending order are: 1 2 3 4 5 6 7 8 9

## PROGRAM CODE

calculate GCD/HCF of two numbers by using a iterative function for GCD.

```
#include <stdio.h>
int gcd(int,int);
int main()
{
int a,b,g;
printf("\nEnter two numbers=>");
scanf("%d%d",&a, &b);

printf("\nThe GCD of %d and %d is
%d\n",a, b, gcd(a,b));
return 0;
}
```

```
/*GCD Iterative Function*/
int gcd(int a, int b)
{
int t;
while(b!=0)
    {   t=b;
         b=a%b;
        a=t;
    }
    return (a);
}
```

# INPUT/OUTPUT

- **RUN-1**
- Enter two numbers=>15 25
- The GCD of 15 and 25 is 5
- **RUN-2**
- Enter two numbers=>22 14
- The GCD of 22 and 14 is 2

# How to check if a given number is Fibonacci number?

A simple way is to generate Fibonacci numbers until the generated number is greater than or equal to ‘n’. Following is an interesting property about Fibonacci numbers that can also be used to check if a given number is Fibonacci or not.

**A number is Fibonacci if and only if one or both of (5*n^2 + 4) or (5*n^2 – 4) is a perfect square .**

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

// A utility function that returns 1 if x is perfect square
int isPerfectSquare(int x)
{
    int s = sqrt(x);
    if(s*s == x)
        return 1;
    else
        return 0;
}

// Returns 1 if n is a Fibonacci Number, else 0
int isFibonacci(int n)
{
    // n is Fibonacci if one of 5*n*n + 4 or 5*n*n - 4 or both
    // is a perfect square

    if(isPerfectSquare(5*n*n + 4) || isPerfectSquare(5*n*n - 4))
        return 1;
    else
        return 0;
}

// A utility function to test above functions
int main(void)
{
    int i;
    for (i = 1; i <= 10; i++)
    {
        if(isFibonacci(i)
            printf("%d is a Fibonacci Number \n",i);
        else
            printf("%d is not a Fibonacci Number. \n",i);
    }
    return 0;
```

You can simply take i as input, no need for loop

```c
/*Calculate Sum of Series:
    Sum=1-x2/2! +x4/4!-x6/6!+x8/8!-x10/10! (x2 means : x power 2)
*/

#include <stdio.h>
#include <math.h>

main()
{
    int counter,f_coun;
    float sum=0,x,power,fact;

    printf("\tEQUATION SERIES : 1- X^2/2! + X^4/4! - X^6/6! + X^8/8! - X^10/10!");

    printf("\n\tENTER VALUE OF X : ");
    scanf("%f",&x);

    for(counter=0, power=0; power<=10; counter++,power=power+2)
    {
        fact=1;
        //Factorial of POWER value.
        for(f_coun=power; f_coun>=1; f_coun--)
            fact *= f_coun;
        //The main equation for sum of series is...
        sum=sum+(pow(-1,counter)*(pow(x,power)/fact));
    }

    printf("SUM : %f",sum);

}
```

# Scope Rules

- scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. There are three places where variables can be declared in C programming language –
- Inside a function or a block which is called **local** variables.
- Outside of all functions which is called **global** variables.
- In the definition of function parameters which are called **formal** parameters.

## Example of local variables

```
#include <stdio.h>
int main ()
 {
/* local variable declaration , local to main*/
 int a, b;
int c;
/* actual initialization */
a = 10;
b = 20;
c = a + b;

 printf ("value of a = %d, b = %d and c = %d\n", a, b, c);
return 0;
}
```

- Variables that are declared inside a function or block are called local variables.
- They can be used only by statements that are inside that function or block of code.

## Example of global variables

```
#include <stdio.h>
/* global variable declaration */
int g;
int main ()
 {
/* local variable declaration */
int a, b;
 /* actual initialization */
a = 10;
b = 20;
g = a + b;
printf ("value of a = %d, b = %d and g = %d\n", a, b, g);
 return 0;
 }
```

- Global variables are defined outside a function, usually on top of the program.
- Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

## Local and global variables Precedence

```
#include <stdio.h>
 /* global variable declaration */
int g = 20;
 int main ()
{
 /* local variable declaration */
int g = 10;
printf ("value of g = %d\n", g);
return 0;
}
```

**OUTPUT**
**value of g = 10**

- A program can have same name for local and global variables but the value of local variable inside a function will take preference.

## Formal Parameters

```
#include <stdio.h>
 /* global variable declaration */
int a = 20;
int main ()
{ /* local variable declaration in main function */
int a = 10;
int b = 20;
int c = 0;
printf ("value of a in main() = %d\n", a);
c = sum( a, b);
printf ("value of c in main() = %d\n", c);
return 0;
}
/* function to add two integers */
 int sum(int a, int b)
{ printf ("value of a in sum() = %d\n", a);
printf ("value of b in sum() = %d\n", b);
return a + b;
 }
```

```
value of a in main() = 10
value of a in sum() = 10
value of b in sum() = 20
value of c in main() = 30
```

- Formal parameters, are treated as local variables with-in a function and they take precedence over global variables.

# Initializing Local and Global Variables

When a **local variable** is defined, it is not initialized by the system, **you must initialize it yourself.**

**Global variables are initialized automatically** by the system when you define them as follows –

| Data Type | Initial Default Value |
|---|---|
| **int** | **0** |
| **char** | **'\0'** |
| **float** | **0** |
| **double** | **0** |

# Scope Rules

- The region of the program over which the declaration of an identifier is accessible is called the *scope of the identifier.*
- The scope relates to the accessibility, the period of existence, and the boundary of usage of variables declared in a program.
- Scopes can be of four types.
  - Block
  - File
  - Function
  - Function prototype

# Storage Classes

| Storage class specifier | Place of storage | Scope | Lifetime | Default value |
|---|---|---|---|---|
| auto | Primary memory | Within the block or function where it is declared. | Exists from the time of entry in the function or block to its return to the calling function or to the end of block. | garbage |
| register | Register of CPU | Within the block or function where it is declared. | Exists from the time of entry in the function or block to its return to the calling function or to the end of block. | garbage |
| static | Primary memory | *For local*<br>Within the block or function where it is declared.<br>*For global*<br>Accessible within the program file where it is declared | *For local*<br>Retains the value of the variable from one entry of the block or function to the next or next call.<br>*For global*<br>Preserves value in the program file | 0 |
| extern | Primary memory | | Exists as long as the program is in execution. | 0 |

# Storage Class Specifiers for Functions

- The only storage class specifiers that may be assigned with functions are extern and static.
- The extern signifies that the function can be referenced from other files.
- The static signifies that the function cannot be referenced from other files.
- If no storage class appears in a function definition, extern is presumed.

# Recursion

- *Recursion in programming is a technique for defining a problem in* terms of one or more smaller versions of the same problem.
- A function that calls **itself directly or indirectly to solve a smaller version of its task until a final call which does not require a self-call is a recursive function.**
- The following are necessary for implementing recursion:
  - Decomposition into smaller problems of same type.
  - Recursive calls must diminish problem size.
  - Necessity of base case.
  - Base case must be reached.

# code structure

```
 int main()
{
   take input
   call recursion();
}
void recursion()
 {
   if base class - return
  else - call recursion(); /* function calls itself with decreased
   parameter */
 }
```

But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.

**what is a base class ?**

An instance of a problem which requires no further recursive calls.

**The recursive algorithm**

If (this is a base class)

Solve it directly. Return directly

Else

- Redefine the problem using recursion.
- The size should be diminished.

# What is needed for implementing recursion?

- Decomposition into smaller problems of same type
- Recursive calls must diminish problem size
- Necessity of base case
- Base case must be reached
- It acts as a terminating condition. Without an explicitly defined base case, a recursive function would call itself indefinitely.
- It is the building block to the complete solution. In a sense, a recursive function determines its solution from the base case(s) it reaches.

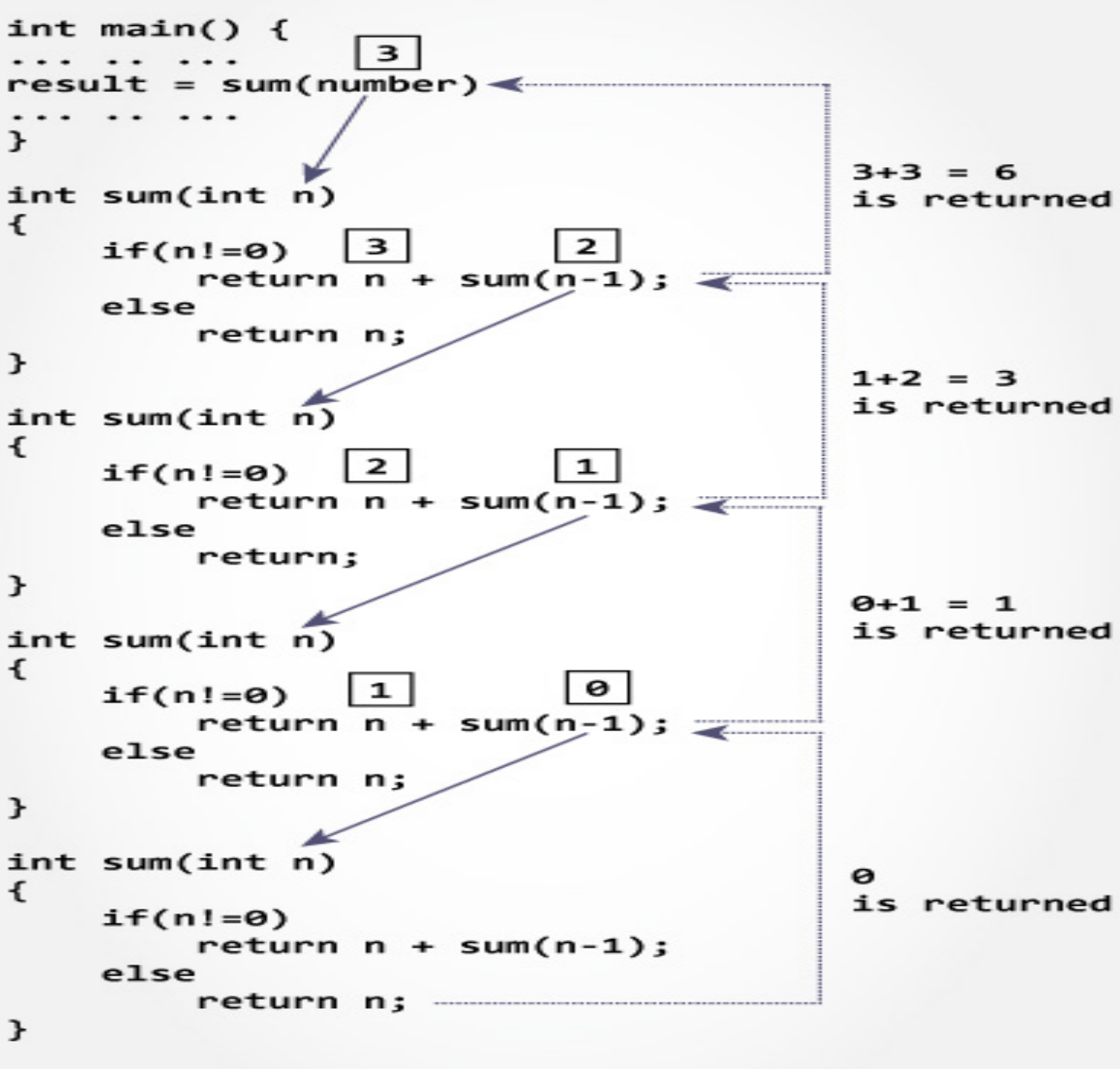
## Sum of Natural Numbers Using Recursion

```
#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);
}
```

RUN

Enter a positive integer:
3
6

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


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

factorial

```
#include <stdio.h>
int factorial(unsigned int i)
{
if(i <= 1)
     {
     return 1;
      }
 return i * factorial(i - 1);
}


int main()
{
 int i = 15;
 printf("Factorial of %d is %d\n", i, factorial(i));
 return 0;
}
```

## PROGRAM CODE

calculate GCD/HCF of two numbers by using a iterative function for GCD.

```
#include <stdio.h>
int gcd(int,int);
int main()
{
int a,b,g;
printf("\nEnter two numbers=>");
scanf("%d%d",&a, &b);

printf("\nThe GCD of %d and %d is
%d\n",a, b, gcd(a,b));
return 0;
}
```

```
/*GCD Iterative Function*/
int gcd(int a, int b)
{
int t;
while(b!=0)
    {   t=b;
         b=a%b;
        a=t;
    }
    return (a);
}
```

## GCD of Two Numbers using Recursion

```
#include <stdio.h>

 int hcf(int n1, int n2);
int main()
 {
int n1, n2;
printf("Enter two positive integers: ");
scanf("%d %d", &n1, &n2);

printf("G.C.D of %d and %d is %d.", n1, n2,
hcf(n1,n2)); return 0;
}

 int hcf(int n1, int n2)
{
      if (n2 != 0)
            return hcf(n2, n1%n2);
      else
            return n1;
}
```

**RUN**

```
Enter two positive integers:
366
60

G.C.D of 366 and 60 is 6.
```

**Count digits of a number program using recursion.**

```
include <stdio.h>
 //function to count digits
int countDigits(int num)
{
   static int count=0;

   if(num>0)
   {
      count++;
      countDigits(num/10);
   }
   else
   {
      return count;
   }
}

int main()
{
   int number;
   int count=0;

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

   count=countDigits(number);

   printf("Total digits in number %d is: %d\n",number,count);

   return 0;
}
```

**RUN**

```
Enter a positive integer number:
123
Total digits in number 123 is: 3
```

# Print Sum of Even Numbers in Array using Recursion

```
#include<stdio.h>
void SumOfEven(int a[],int num,int sum);
main()
{
    int i,a[100],num,sum=0;
    printf("Enter number of Array Elements\n");
   scanf("%d",&num);
   printf("Enter Array Elements\n");
   for(i=0;i<num;i++)
   {
    scanf("%d",&a[i]);
    }

   SumOfEven(a,num-1,sum);
}
```

```
void SumOfEven(int a[],int num,int sum)
{

 if(num>=0)
 {
  if((a[num])%2==0)
  {
   sum+=(a[num]);
  }
  SumOfEven(a,num-1,sum);
 }
 else
 {
    printf("Sum=%d\n",sum);
  return;
 }
}
```

```
#include<stdio.h>
int fib(int);
int main()
{
int n;
printf("\nEnter term number :");
scanf("%d",&n);
printf("\nThe value of the term-%d of
Fibonacci sequence is %d.", n, fib(n));
return 0;
}
/*Recursive version
 of the
Fibonacci function to
compute the ith term*/

int fib(int i)
{
if(i==0 || i==1)
     return (i);
else
     return (fib(i-1)+ fib(i-2));
}
```

**Find nth term in Fibonacci series**
**0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …**

```
fib(4)

= fib(3) + fib(2)
= (fib(2) + fib(1)) + (fib(1) + fib(0))
= (fib(2) + 1 ) + ( 1 + 0 )
= (fib(2) + 2)
= ((fib(1) + fib(0)) + 2
= 1 + 0 + 2
= 3
```

**INPUT/OUTPUT**

**RUN-1**
**Enter term number: 6**
**The value of the term-6 of Fibonacci sequence is 8.**

**RUN-2**
**Enter term number: 15**
**The value of the term-5 of Fibonacci sequence is 377.**

# Searching

- Among the searching algorithms, only two of them will be discussed here;
  - Sequential search
  - Binary search.

# Sequential Search

- Here is an implementation of this simple algorithm:
  - int Lsearch(int  ArrayElement[], int key, int ArraySize)

```
{
    int i ;
    for (i = 0; i < ArraySize; i++)
        if (ArrayElement[i] == Key)
    return (i) ;
    return (-1);
}
```

# Binary Search

- The C code for binary search is given below.

```
#include <stdio.h>
int binarysearch(int a[], int n, int key)
{
     int beg,mid;
     beg=0; end=n-1;
     while(beg<=end)
     {
        mid=(beg+end)/2;
       if(key==a[mid])
              return mid;
        else if(key>a[mid])
                  beg=mid+1;
         else
                  end=mid-1;
     }
    return -1;
}
```

**Sort an array using a suitable function for sort operation.**

```
#include<stdio.h>
void bubbleSort(int a[],int);

int main()
{
int a[100], n, i;
printf("\nEnter how many numbers :");
scanf("%d",&n);

printf("\nEnter data for array: ");
for(i=0;i<n;i++)
scanf("%d",&a[i]);

bubbleSort(a,n); //Function Call
printf("\nThe Numbers in ascending order are:");

for(i=0; i<n; i++)
printf("%d ",a[i]);
return 0
}
```

```
/*Bubble Sort Function*/
void bubbleSort(int a[], int n)
{
int i, j, temp;
    for(i=1; i<=n-1; i++)
    {
        for(j=0; j<n-i; j++)
        {
            if(a[j]>a[j+1])
            {
                temp=a[j];
                a[j]=a[j+1];
                a[j+1]=temp;
            }
        }
    }
}
```

INPUT/OUTPUT

RUN-1

Enter how many numbers :8
Enter data for array: 7 6 5 4 5 2 4 8
The Numbers in ascending order are: 2 4 4 5 5 6 7 8

RUN-2

Enter how many numbers :9
Enter data for array: 1 4 3 8 6 5 2 9 7
The Numbers in ascending order are: 1 2 3 4 5 6 7 8