UNIT 5

Functions

- Function is a self -contained block of code or statement that performs a particular task.
- Every program must contain one function named main() from where the program execution begin.
- Complex problem can be solved by breaking them into set of sub-problems, called modules or functions. This technique is called divide and conquer. Each module can be implemented independently and later can be combined into single unit.

Advantages of function /Importance of function

- Functions in Programming help break down a program into smaller, manageable modules. Each function can be developed, tested, and debugged independently, making the overall program more organized and easier to understand.
- Different programmers working on one large project divide the workload by writing different functions.
- A single function can be used multiple times which avoids rewriting of same code again and again.
- Program that use functions are easier to design debug and maintain.
- Functions in Programming allow programmers to abstract the details of a particular operation. A programmer can use a function with a clear interface, relying on its functionality without needing to understand the internal complexities.

Disadvantages

It increase the execution time because when function is called the control have to jump to function definition to perform the particular task, and after completion of task , control again come back to the function call.

Assignment:

- ✓ What do you mean by functions in C programming?
- ✓ Without using functions also, we can write a program. But we need functions in our program. What are the benefits of using them?

Types of functions

C functions are classified into two categories. They are:

1. **Library function**

These are the functions which are already written, compiled and placed in C Library and they are not required to be written by a programmer. The function name, its return type, their argument number and types have been already defined. We can use these functions as required by just calling them. For example: printf(),scanf(),sqrt(),getch() are examples of library functions .The library functions are also known as built-in functions and they are defined within a particular header file.

2. User defined function

These are the functions which are defined by user at the time of writing a program. The user has choice to choose its name, return type, arguments and their types. eg. int sum(int a,int b).The task of each user-defined functions is as defined by user. A complex C program can be divided into number of user-defined functions.

Differentiate between library functions and user-defined functions with examples.

Library functions	User defined functions
1. They are predefined functions/built in functions in C library.	1. They are the function which are created by the user as per his own requirements.
2. They are the part of header files (such as stdio.h, conio.h, math.h) which are called during runtime.	2. They are the part of program which is compiled at runtime.
3. The name of function ID is given by developers which can't be changed.	3. The name of function id is declared by user which can be changed.
4. The function's name, its return type, their arguments number and their types have been already defined .	4. The user has choice to choose function name, its return type, number of argument and their types.
5.Example. printf(),scanf(),sqrt(),pow(),clrscr() etc.	5.Example. int fact(int n), float sum(float x,float y)

Program to illustrate use of user defined function and library functions.

```
#include<stdio.h>
#include<math.h>
int square(int);
int main()
{
int num=5,result;
result=pow(num,2);
printf("Square of number using library function =%d\n",result);
result= square(num);
printf("Square of number using user defined function=%d",result);
}
int square(int x)
{
return (x * x);
}
```

Output:

```
Square of number using library function=25
Square of number using user defined function=25
```

Components associated with function

1) Function definition

The collection of program statements that describes the specific task to be done by the function is called function definition. It consists of:

Function header: which defines function's name, its return type and its argument list

Function body : which is a block of code enclosed in parenthesis.

Syntax:

```
return_type   function_name(data_type varibale1,data_type variable2,…..,data_type variable n)
{
statements ;
……..
……
return  value;
}
```

Note: If a function doesn't return any value , then its return type is void

Syntax:

```
void  function_name (data_type varibale1,data_type variable2,…..,data_type variable n)
{
statements ;
……..

……

}
```

2) function declaration /function prototype

The function declaration or prototype is a blueprint of a function. If a function is used before it is defined in a program, then function declaration is needed to provide the following information to the compiler.

- The name of function
- The type of the value returned by the function
- The number and type of arguments that must be supplied while calling the function.

The syntax for function declaration is :

return_type function_name(type1, type2 ,type3 …… type n);

Here, return_type specifies the data type of the value returned by the function. A function can return value of any data type.If there is no return value, the keyword void is used.

The function declaration and declarator or header in function definition must use the same function name, number of arguments, argument types and return types. Some examples of function declaration are:

```
int add(int a,int b);
```

float simple_interest();

int factorial(int n);

Note:

The terms' function declaration and function prototypes are often used interchangeably but they are different in the purpose and their meaning. Following are the major differences between the function declaration and function prototype in C:

Function declaration	**Function prototype**
Function Declaration is used to tell the existence of a function.	The function prototype tells the compiler about the existence and signature of the function.
A function declaration is valid even with only function name and return type.	A function prototype is a function declaration that provides the function's name, return type, and parameter list without including the function body.
Typically used in header files to declare functions.	Used to declare functions before their actual definitions.
Syntax: return_type function_name();	**Syntax:** return_type function_name(parameter_list);

3) Calling a function

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

A function can be called by specifying its name, followed by a list of arguments enclosed in parentheses and separated by commas. For example, the function add() with two arguments is called by add(a,b) to add two numbers.

During function call, function name must match with function declaration name, the type of return type, the number of arguments and order of arguments.

4) Return statement:

The job of return statement is to hand over some value given by function body to the point from where the call was made. The function defined with its return type must return a value of that type. For example: If a function has return type int, it returns an integer value from the called function to the calling function.

The main functions return statement are:

- It immediately transfers the control back to the calling program after execution of return statement (i.e. no statement within the function body is executed after the return statement.)
- It returns value to the calling function.

The syntax for return is:

return (expression);

Where expression must evaluate to a value of the type specified in the function header for the return value.

5) Function Arguments

Explain formal arguments and actual arguments with an example program.

Arguments enable data communication between the calling function and the called function.

They are classified into actual arguments and formal arguments:

Actual Arguments:

- ✓ These are the values or variables passed from the calling function to the called function.
- ✓ They are specified in the function call.

Formal Arguments:

- ✓ These are the variables defined in the function definition to receive data sent by the calling function.
- ✓ They can also be used to send data back to the calling function.

For example, in a function, a and b are actual arguments passed during the function call, while x and y are formal arguments used in the function definition to process the received values.

Program

```
#include<stdio.h>
int sum(int,int);                    //function declaration
int main()
{
    int a, b,result;
    printf("Enter the first number\n");
    scanf("%d",&a);
    printf("Enter the second number\n");
    scanf("%d",&b);

    /* Calling the function here, the function return type
     is integer so we need an integer variable to hold the
     returned value of this function.
     */

    result =sum(a,b);
    /*actual arguments a and b are passed in calling function*/
    printf ("Sum of two numbers= %d, result);
    return 0;
}
```

```
/*formal arguments x and y are passed in called function*/
int sum(int x, int y)      //function definition
{
    int r;
    r=x+y;

    /* Function return type is integer so we are returning
     an integer value, the sum of the passed numbers.
     */
    return r;
}
```

Passing argument to function

- ✓ Distinguish between call by value and call by reference with examples.
- ✓ How arguments can be passed by using call by value and call by reference? Explain with examples.
- ✓ Explain the call by value and call by pointer with suitable example.

The arguments in function can be passed in two ways, namely **call by value** and **call by reference.**

Function call by value (or pass by value)

- In this method the value of each actual arguments in the calling function is copied into corresponding formal arguments of the called function.
- With this method the changes made to the formal arguments in the called function have no effect on the values the actu[al argument in the calling function.

```
#include<stdio.h>
void swap(int x,int y);
int main()
{
          int a,b;
          a=10;
          b=20;
          printf("value before swapping a=%d and b=%d\n",a,b);
          swap(a,b);
          printf("value after swapping a=%d and b=%d\n",a,b);
         return 0;
}
void swap(int x,int y)
{
 int temp;
 temp=x;
 x=y;
 y=temp;
}
```

Output:

```
Value before swapping a=10 and b=20
Value after swapping a=10 and b=20
```

Explanation:
In this example, the values of a and b are passed in function swap() by value. The value of a is copied into formal argument x and value of b is copied into formal argument y. The copy of value of a and b to x and y means the original values of a and b remain same and these values are copied into the variables x and y also. Thus when x and y are changed within the function , the values of the variables x and y are changed but the original value of a and b remains same.

Function call by reference (Or pass by Reference)

- In this method the address of actual arguments in the calling function are copied into formal arguments of the called function. This means that using these addresses the actual arguments can be accessed.
- In call by reference since the address of the value is passed any changes made to the value reflects in the calling function.

```
#include<stdio.h>
void swap(int *x,int *y);

int main()
{
 int a,b;
 a=10;
 b=20;
 printf("Value before swapping  a=%d and b=%d\n",a,b);
 swap(&a,&b);
 printf("Value after swapping   a=%d and b=%d\n",a,b);
return 0;
}

void swap(int *x,int *y)
{
 int temp;
 temp=*x;
 *x=*y;
 *y=temp;

}
```

Output

```
Value before swapping a=10 and b=20
Value after swapping a=20 and b=10
```

Explanation: *In this example, the address of variables (ie.&a and &b) are passed in function .These addresses are received by corresponding formal arguments in function definition. To receive addresses, the formal arguments must be of type pointers which stores the address of variables. The address of a and b are copied to the pointer variable x and y in function definition. When the values in addresses pointed by these pointer variables are altered, the values of original variables are also changed as the content of their addresses have been changed. Thus, while arguments are passed in function by reference, the original values are changed if they are changed within function.*

Category of functions:

Function can be categorized in four types, on the basis of arguments and return value.

1.Function with no arguments and no return values

2.Function with arguments but no return values

3.Function with no arguments but with return values

4.Function with arguments and return values

1. Function with no arguments and no return values

When a function has no arguments, it does not receive any data from the calling function. Similarly, when it doesn't return a value, the calling function does not receive any data from the called function. Thus in such type of functions, there is no transfer between the calling function and the called function. This type of function is defined as.

```
void function_name( )
{
/*body of the function*/
}
```

The keyword **void** means the function does not return any value. There is no arguments within parenthesis which implies function has no argument and it does receive any data from the called function.

Program to illustrate the "function with no arguments and no return values"

```
#include<stdio.h>
void sum();
int main()
{
 sum();
return 0;
}
```

```
void sum()
{
 int x,y,r;
 printf("Enter the two numbers\n");
 scanf("%d%d",&x,&y);
 r=x+y;
 printf("The sum of numbers=%d",r);
}
```

2. **Function with arguments but no return value**

These type of function has arguments and receives the data from the calling function. The function completes the task and does not return any values to the calling function. Such type of functions are defined as

```
void function_name(argument list)
{
/*body of function*/
}
```

Program to illustrate the "function with arguments but no return values"

```
#include<stdio.h>
void sum(int x,int y);
int main()
{
 int a,b;
 printf("Enter the two numbers\n");
 scanf("%d%d",&a,&b);
 sum(a,b);
return 0;
}
void sum(int x,int y)
{
 int r;
 r=x+y;
 printf("The sum of numbers=%d\n",r);

}
```

3. **Function with no arguments but with return values**

These type of function does not receive any arguments but the function return values to the calling function. Such type of functions are defined as

```
return_type   function_name( )
{
/*body of function*/
}
```

Program to illustrate the "function with no arguments but with return values"

```
#include<stdio.h>
int sum();

int main()
{
 int result;
 result=sum();
 printf("The sum of numbers=%d",result);
return 0;
}

int sum()
{
 int x,y,r;
 printf("Enter the two numbers\n");
 scanf("%d%d",&x,&y);
 r=x+y;
 return r;
}
```

4. Function with arguments and return value

The function of this category has arguments and receives the data from the calling function. After completing its task ,it returns the result to the calling function through return statement. Thus there is data transfer between called function and calling function using return values and arguments. These type of functions are defined as

```
return_type function_name(argument list)

{

/*body of the function*/

}
```

Program to illustrate the "function with arguments and return values"

```
#include<stdio.h>
int sum(int x ,int x);
int main()
{
 int a,b,result;
 printf("Enter the two numbers\n");
 scanf("%d%d",&a,&b);
 result=sum(a,b);
 printf("The sum of numbers=%d",result);
return 0;
}
```

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

Global vs Local variable

Global variables:

Global variables are accessible to all functions defined in the program. Global variables are declared outside any function, generally at the top of program after preprocessor directives. It is useful to declare variable global if it is to be used by many functions in the program. The default initial values for this variable is zero. The lifetime is as long as the program execution does not come to an end.

Local variables:

The variables that are defined within the body of a function or block and local to that function or block only are known as local variables. The name and value of local variables are valid within the function in which it is declared. They are unknown to other. The local variables are created when the function is called and destroyed automatically when function is exited function.

Example:

```
#include<stdio.h>
void fun();
int x=5;                    //global declaration
 int main()
 {
        int a=10;       //local declaration
        printf("Value of x=%d,a=%d\n",x,a);
        fun();
        return 0;

 }
 void fun()
 {
        int b=20;       //local declaration
        printf("Value of x=%d,b=%d\n",x,b);

 }
```

Output

```
Value of x=5, a=10
Value of x=5, b=20
```

Here x is global variable and both main () function and fun() function can access them. a and b are local variables a can be accessed by only main() function and b can be only accessed only by fun() function.

Storage classes in C

What do you mean by storage class? Explain different types of storage classes in C? Use examples to illustrate**.**

Write a short note on: storage class in c?

In C language, each variable has a storage class which decides the following things:

- **Scope**: where the value of the variable would be available inside a program.
- **Default initial value:** if we do not explicitly initialize that variable, what will be its default initial value.
- **Storage location of that variable**
- **Lifetime of that variable**: for how long will that variable exist.

Syntax: storage_class_specifier type_specifier variable_name

	Storage location	Default initial value	Scope	Lifetime
Auto	Memory	Garbage value	local to function in which variable is defined.	Till control remains within block where it is defined
register	CPU register	Garbage value	local to function in which variable is defined.	Till control remains within block where it is defined
Static	Memory	Zero	local to function in which variable is defined.	Value of the variable persist between different function call
extern	Memory	Zero	Everywhere in the program	Till the program doesn't finish its execution

Explanation:

1. Automatic or Local variables (auto storage class)

- The automatic variables are always declared within a function or block.
- As local variables are defined within the body of the function or the block other functions cannot access these variables, the compiler shows error in case other function try to access the variables.
- The local variables are created when the function is called and destroyed automatically when the function is exited.

- The keyword **auto** is used for storage class specification while declaration of variable.
- A variable declared inside a function without storage class specification **auto** is, by default, an automatic variable.

Storage Location	Memory
Default initial value	Garbage value
Scope	local to function in which variable is defined.
Life time	Till control remains within block where it is defined

Example:

```
#include<stdio.h>
void function1(void);
int main()
{
        auto int a=5;
        printf("a=%d\n",a);
        function1();
        return 0;

}
void function1()
{
  auto int b=10;
  printf("b=%d",b);
}
```

Output

```
a=5
b=10
```

2. Register variables (Register storage class)

Register variables inform the compiler to store the variable in CPU register instead of memory. Register variables have faster accessibility than a normal variable. Generally, the frequently used variables are kept in registers. But only a few variables can be placed inside registers. If the declaration of register variables exceeds the availability, they will be automatically converted into non register variables (automatic variable).One application of register storage class can be in using loops, where the variable gets used a number of times in the program, in a very short span of time. Register variable are declared by using the keyword **register.**

Storage Location	CPU Register
Default initial value	Garbage value
Scope	local to function in which variable is defined.
Life time	Till control remains within block where it is defined

Example:

```
#include<stdio.h>
int main()
{
  register int i;
  for (i=0;i<5;i++)
  {
    printf("%d\t", i);
  }
  return 0;
}
```

Output

```
0       1       2       3       4
```

3. Extern storage class

- ✓ Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used.
- ✓ External variables share the variables among the multiple C files.
- ✓ It tells the compiler that the variable exists and its definition can be found in another source file.

Storage Location	Memory
Default initial value	Zero
Scope	(Global) Everywhere in the program
Life time	As long as the program execution doesn't come to end.

Example 1:

somefile.h

```
int num=5;
```

demo.c

```
#include <stdio.h>
#include "somefile.h"
int main()
{
extern int num;
printf("value of num=%d",num);
return 0;
}
```

From the above-given program, we are trying to access the extern variables in the demo.c file, which is declared in the program somefile.h.

Output:

```
value of num=5
```

4. Static variables (static storage class)

A static variable tells the compiler to persist/save the variable until the end of program. Instead of creating and destroying a variable every time when it comes into and goes out of scope, static variable is initialized only once and remains into existence till the end of the program.

Storage Location	Memory
Default initial value	Zero
Scope	local to function in which variable is defined.
Life time	Value of the variable persist between different function call

Example:

```
#include<stdio.h>
void increment();
int main()
{
increment();
increment();
increment();
increment();
return 0;
}
void increment()
{
static int i = 1 ;
printf ( "%d\t ", i ) ;
i++;
}
```

Output

1 2 3 4

Difference between extern and global variables

Extern	global
extern is only a declaration, meaning it tells the compiler that a variable exists elsewhere, but does not allocate or initialize it.	A global variable is both declared and defined within the same file, thus it has both storage and an initial value.
extern provides access to a variable defined elsewhere (between multiple files).	A global variable is defined in the current file or translation unit and has global visibility within that file.
extern is useful for inter-file communication.	global variables are used for data shared across different parts of a single file or throughout the entire program.
Does not allocate memory.	Allocates memory and store the value.

Preprocessor directives

What is preprocessor directive?

Preprocessor is a program that process source code before it passes through the compiler. It operates under the control of which is known as preprocessor directives.
Preprocessor directives are preceded by a hash (#) sign.
Preprocessor directives are often placed in beginning of program before main function. No semicolon (;) is expected at the end of the preprocessor directive.

Example:

#include
#define
#ifdef
#undef
#if
#else
#endif

We use preprocessor directive for

1. File inclusion

Source code of the given "filename" is included in the main program in specified place.
Syntax:
#include<filename>
#include "filename"
Eg.#include<stdio.h>

2. Conditional compilation

Set of commands are included or excluded in source program before compilation with respect to condition.
Eg.

#ifdef	Returns true if this macro is defined
#ifndef	Returns true if this macro is not defined
#if	Test if compile time condition is true
#else	The alternative for if
#endif	Ends preprocessor conditional

3. Macro expansion

Macros are a piece of code in a program which is given some name. Whenever this name is encountered by the compiler the compiler replaces the name with the actual piece of code. The '#define' directive is used to define a macro.

Eg. #define PI 3.1415

Here, when PI is used in program its value is replaced with 3.1415

MACROS

Write a short notes on: **macros**

A macro is a fragment of code which has been given a name. Whenever the name is used, it is replaced by the contents of the macro. Macro is defined by #define directive.

#define macro_name macro_expansion

- macro_name is any valid C identifier, and it is generally in capital letters to distinguish it from other variables.
- macro_expansion can be any text

There are two types of macros:

1) Object-like Macros

The object-like macro is an identifier that is replaced by value. It is widely used to represent numeric constants.

For example: #define PI 3.14

Here, PI is the macro name which will be replaced by the value 3.14.

Example

```
#include <stdio.h>
#define PI 3.14
int main()
{
  float r,area;
  printf("Enter the radius: ");
  scanf("%f", &r);
  area = PI*r*r;
   printf("Area=%.2f",area);
   return 0;
}
```

2) Function-like Macros

The function-like macro looks like function call. For example:

#define max(a,b) ((a)>(b)?(a):(b)) Here, max is the macro name.

```
#include<stdio.h>
#define max(a,b) ((a>b)?(a):(b))
int main()
{
        int a,b;
        printf("Enter the two numbers\n");
        scanf("%d%d",&a,&b);
        printf("Maximum number=%d",max(a,b));
        return 0;
 }
```

Differentiate between macro and function.

Macro	Function
1. Macro is preprocessed.	1. Function is compiled.
2. Before Compilation macro name is replaced by macro value.	2. During function call , Transfer of Control takes place
3. Macros are faster in execution than function.	3. Functions are bit slower in execution.
4. Useful where small code appears many time	4. Useful where large code appears many time
5. Generally Macros do not extend beyond one line	5. Function can be of any number of lines
6. Increase the program size.	6. Makes program smaller and compact.
7. Macros do not check for compilation error which leads to unexpected output.	7. Function checks for compilation error and there is a less chance of unexpected output
8. Macro can't call it recursively	8. Function can call it recursively.
9. A macro is defined using #define preprocessor directive. Therefore it is preprocessed before submission of source code to the compiler.	9. A function is defined outside main program. It is processed after submission of source code to compiler.

Example:

Program to find the square of a number by using both macro and function

```
#include<stdio.h>
#define NUM 5
#define square(num) (num*num)
void func();
 int main()
 {
  printf("Square of number using macro=%d\n",square(NUM));
  func();
  return 0;
 }
void func()
{
        int result,a=5;
        result=a*a;
        printf("Square of number using function=%d",result);
}
```

Output:

```
Square of number using macro=25
Square of number using function=25
```

Header files in C

A header file is a file with extension .h which contains C function declarations and macro definitions to be shared between several source files. There are two types of header files: the files that the programmer writes and the files that comes with your compiler.

Including a header file is equal to copying the content of the header file but we do not do it because it will be error-prone and it is not a good idea to copy the content of a header file in the source files, especially if we have multiple source files in a program.

Both the user and the system header files are included using the preprocessing directive #include. It has the following two forms :

#include <file>

This form is used for system header files. It searches for a file named 'file' in a standard list of system directories.

#include "file"

This form is used for header files of your own program. It searches for a file named 'file' in the directory containing the current file.

Why header files in C are included in program? Give reasons. Also list out different header files you know. Illustrate the program showing the use of header file.

Header files are the special files which stores the predefined library functions. Suppose if we are using printf() & scanf() functions in your program then we have to include <stdio.h> header file. It is for standard i/o. If we don't include the the header files we can't run our program. Therefore we have to include the header files.

Another header file is #include<conio.h> which is for console i/o. This header file stores clrscr() & getch() functions. clrscr() is for clearing the screen & getch() is holding the screen until we press any single character from keyboard.

Also we have lots of different header files which stores different functions

Header file	function
#include<stdio.h>	Used to perform input and output operations in C like scanf() and printf().
#include<string.h>	Perform string manipulation operations like strlen and strcpy
#include<conio.h>	Perform console input and console output operations like clrscr() to clear the screen and getch() to get the character from the keyboard.
#include<stdlib.h>	Perform standard utility functions like dynamic memory allocation, using functions such as malloc() and calloc().
#include<math.h>	Perform mathematical operations like sqrt() and pow(). To obtain the square root and the power of a number respectively.
#include<signal.h>	Perform signal handling functions like signal() and raise(). To install signal handler and to raise the signal in the program respectively
#include<errno.h>	Used to perform error handling operations like errno().

Program to illustrate the use of header file

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
int main()
{
 int num,result;
 printf("Enter the number");
 scanf("%d",&num);
 result=pow(num,2);
 printf("Square of a given number is %d",result);
return 0;
}
```

Here,

Function used	Header file must included to use function
printf() and scanf()	stdio.h
pow()	math.h
getch()	conio.h

Recursive function (Nesting of function)

A function that calls itself is known as a recursive function. Recursion is a process by which function calls itself repeatedly until some specified condition will be satisfied.

To solve a problem using recursive method, two conditions must be satisfied. They are

1. Problem could be written or defined in terms of previous result
2. Problem statement must include stopping condition

Advantages

- Avoid unnecessary calling of functions.
- Through Recursion one can solve problems in easy way while its iterative solution is very big and complex
- Reduces time complexity.
- Using recursion, the length of the program can be reduced.
- Extremely useful when applying the same solution repeatedly.

Disadvantages

- Recursive solution is always logical and it is very difficult to trace. (debug and understand).
- For each step we make a recursive call to a function. For which it occupies significant amount of stack memory with each step.
- It is usually slower due to the overhead of maintaining the stack.
- May cause stack-overflow if the recursion goes too deep to solve the problem
- If the programmer forgets to specify the exit condition in the recursive function, the program will execute out of memory.

1. **Write to find the factorial of a given number using recursive function.**

```
#include<stdio.h>
long int fact(int n);
int main()
{
  int num;
  long int f;
  printf("Enter the number\n");
  scanf("%d",&num);
  f=fact(num);
  printf("The factorial of a given number is %ld",f);
  return 0;
}
long int fact(int n)
{
        if(n==1||n==0)
        {
                return 1;
        }
        else
        {
                return n*fact(n-1);
        }
}
```

2. **Write a program to find the sum of first n natural number using recursive function.**

```
#include<stdio.h>
int snatural(int n);
int main()
{
int num,s;
printf("Enter the value of n \n");
scanf("%d",&num);
s=snatural(num);
printf("The sum of n natural number is %d",s);
return 0;
}
int snatural(int n)
{
        if(n==1)
        {
                return 1;
        }
        else
        {
                        return n+snatural(n-1);
        }

}
```

3) Write a program to generate the Fibonacci series upto nth term using recursive function. Fibonacci series is 0,1,1,2,3,5….

```
#include<stdio.h>
int fib(int n);
int main()
{
        int n,result,i;
        printf("Enter how many terms you want to generate\n");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
        result=fib(i);
        printf("%d\t",result);
   }
        return 0;
}

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

}
```

4) Write a recursive program to generate the first 15 numbers of Fibonacci series.

```
#include<stdio.h>
int fib(int n);
int main()
{
        int i,result;
        for(i=0;i<15;i++)
        {
        result=fib(i);
        printf("%d ",result);
        }
        return 0;
}
```

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

 }
```

5) Write a recursive program to generate 10 terms Fibonacci sequence starting from 2.

```
#include<stdio.h>
int fib(int n);
int main()
{
        int i,result;
        for(i=2;i<12;i++)
        {
        result=fib(i);
        printf("%d\t",result);
         }
        return 0;
}

int fib(int n)
{
        if (n==2||n==3)
        {
        return n;
        }
        else
        {
                return(fib(n-1) + fib(n-2));
        }

}
```

6) Write a program to find nth term of Fibonacci number using recursive function.

```
#include<stdio.h>
int fib(int n);
int main()
{
        int n,result;
        printf("Enter the term:");
        scanf("%d",&n);
        result=fib(n);
        printf("The %d th fibonacci term is %d",n,result);
        return 0;
}

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

7) Write a program to read an integer number and find the sum of digits using recursive function.

```
#include<stdio.h>
int sum(int n);
int main()
{
  int num, result;
  printf("Enter the number\n");
  scanf("%d", &num);
  result = sum(num);
  printf("Sum of digits of given number=%d",result);
  return 0;
}
```

```
int sum (int n)
{
   if (n == 0)
   {
         return 0;
    }
   else
   {
         return (n % 10 + sum(n / 10));
    }

}
```

8) Use recursive functions calls to evaluate.

f(x)=x- $\frac{x3}{3!}$ + $\frac{x5}{5!}$ - $\frac{x7}{7!}$ +.....

```
#include<stdio.h>
#include<math.h>
int fact(int n);
int main()
{
  float sum=0,nume,deno,x;
  int n,i,sign;
  printf("Enter the value of x and n\n");
  scanf("%f%d",&x,&n);
  for(i=1;i<=n;i++)
  {
   nume=pow(x,2*i-1);
   deno=fact(2*i-1);
   sign=pow(-1,i+1);
   sum=sum+sign*nume/deno;
  }
  printf("sum of series=%f",sum);
  return 0;
}

int fact(int n)
{
  if(n==1||n==0)
  {
          return 1;
  }
  else
  {
          return (n*fact(n-1));
  }
}
```

9) Write a program to read an integer and find the number of digits present in it using recursive function.

```
#include<stdio.h>
int count(int n);
int main()
{
  int num, result;
  printf("Enter the number: ");
  scanf("%d", &num);
  result = count(num);
  printf("Number of digits in a given number=%d",result);
  return 0;
}

int count(int n)
{
   if (n == 0)
   {
     return 0;
   }
   else
   {
     return (1+ count(n/10));
   }

}
```

10) Write a recursive program to raise the power of X to n. (i.e. X^n)

OR

Write a program to find power of a given number using recursive function.

```
#include<stdio.h>
float power (float x,int n);
int main()
{
float x,result;
int n;
printf("Enter the value of x\n");
scanf("%f",&x);
printf("Enter the value of n\n");
scanf("%d",&n);
result=power(x,n);
printf("Result=%f",result)
return 0;
}
```

```
float power(float x,int n)
{
	if(n==0)
	{
		return 1;
	}
	else if (n>0)
	{
		return x*power(x,n-1);
	}
	else
	{
		return (1/x)*power(x,n+1);
	}

}
```

11) Write a Program to check whether the given number is Armstrong number or not using recursive function. (for three digit number)

```
#include<stdio.h>
int armstrong(int);
int main()
{
    int num,a;
    printf("Enter any three digit number");
    scanf("%d",&num);
    a=armstrong(num);
    if(a==num)
    {
		printf("The number is armstrong number");
    }
    else
    {
		printf("The number is not armstrong number");
    }

}
int armstrong(int x)
{
    if(x==0)
    {
		return 0;
    }
    else
    {
		return((x%10)*(x%10)*(x%10)+armstrong(x/10));
    }

}
```

Recursion vs Iteration

Recursion	Iteration
1. The statement in a body of function calls the function itself.	1. Allows the set of instructions to be repeatedly executed.
2. Recursion is always applied to functions.	2. Iteration is applied to iteration statements or "loops".
3. In recursive function, only termination condition (base case) is specified	3. Iteration includes initialization, condition, execution of statement within loop and updation (increments/decrements) the control variable
4. If the function does not converge to some condition called (base case), it leads to infinite recursion.	4. If the control condition in the iteration statement never become false, it leads to infinite iteration
5. The stack is used to store the set of new local variables and parameters each time the function is called	5. Does not uses stack.
6. Infinite recursion can crash the system.	6. Infinite loop uses CPU cycles repeatedly.
7. Slow in execution	7. Fast in execution.

Describe the output generated by each of the following program.

```
#include<stdio.h>
int a=100,b=200;
int funct1(int c);
main()
{
int count,c;
for(count=1;count<=10;++count)
{
        c=4*count;
        printf("%d\n",funct1(c));
}
}
funct1(int x)
{
        int c;
        c=(x<30)?(a-x):(b+x);
        return (c);
}
```

Here a=100,b=200 Where a and b are global variables.so that they can accessed from anywhere.

count	c=4*count	funct1(c)	c=(x<30)?(a-x) : (b+x)	print c
1	c=4*1=4	funct1(4)	(4<30)?(True) c=a-x=100-4=96	96
2	c=4*2=8	funct1(8)	(8<30)?(True) c=100-8=92	92
3	c=4*3=12	funct1(12)	(12<30)?True c=100-12=88	88
4	c=4*4=16	funct1(16)	(16<30)?True c=100-16=84	84
5	c=4*5=20	funct1(20)	(20<30)? True c=100-20=80	80
6	c=4*6=24	funct1(24)	(24<30)?True c=100-24=76	76
7	c=4*7=28	funct1(28)	(28<30)?True c=100-28=72	72
8	c=4*8=32	funct1(32)	(32<30)?False c=b+x=200+32=232	232
9	c=4*9=36	funct1(36)	(36<30)?False c=b+x=200+36=236	236
10	c=4*10=40	funct1(40)	(40<30)?False c=b+x=200+40=240	240

Output:

```
96
92
88
84
80
76
72
232
236
240
```

Describe the output generated by each of the following program.

```
#include<stdio.h>
void func1(int n);
int main()
{
        int i;
        for(i=1;i<=4;i++)
        {
                func1(i);
        }
         return 0;
}
```

```
void func1(int n)
{
        int num=3;
        printf("%d\n\n",n*num);
}
```

Tracing

i	i<=4	func1(i)	n	num	print (n*num)
1	1<=4(T)	func1(1)	1	3	3
2	2<=4(T)	func1(2)	2	3	6
3	3<=4(T)	func1(3)	3	3	9
4	4<=4(T)	func1(4)	4	3	12
5	5<=4(F)	terminate			

Output:

```
3

6

9

12
```

PROGRAMS USING FUNCTION

1. Write a Program to calculate the area of two circles having different radius using the same user-defined function named area.

```
#include<stdio.h>
float area(float r);
int main()
{
        float r1,r2,a1,a2;
        printf("Enter the radius of first circle\n");
        scanf("%f",&r1);
        a1=area(r1);
        printf("Enter the radius of second circle\n");
        scanf("%f",&r2)         ;
        a2=area(r2);
        printf("Area of first circle=%f\n",a1);
        printf("Area of second circle=%f\n",a2);

}
```

```
float area(float r)
{
        float a;
        a=3.14*r*r;
        return a;
}
```

2. Write a program to find the factorial of a given positive integer using user-defined function.

```
#include<stdio.h>
long int factorial(int n);
int main()
{
        int num;
        long int result;
        printf("Enter the given positive integer\n");
        scanf("%d",&num);
        result=factorial(num);
         printf("Factorial of a given number is %d",result);
        return 0;
}
long int factorial(int n)
{
 int i,fact=1;
 for(i=1;i<=n;i++)
 {
        fact=fact*i;
 }
 return fact;
 }
```

3. Write a program to find the sum of n natural numbers using user-defined function**.**

```
#include<stdio.h>
int snatural(int n);
int main()
{
   int num,s;
    printf("Enter the value of n\n");
    scanf("%d",&num);
    s=snatural(num);
     printf("Sum of n natural number is %d",s);
      return 0;
}
```

```
int snatural(int n)
{
  int i,sum=0;
  for(i=1;i<=n;i++)
  {
          sum=sum+i;
  }
  return sum;
}
```

4. Write a Program to calculate the area and perimeter of rectangle using user-defined function.

```
#include<stdio.h>
void area(float,float);
void perimeter(float,float);
int main()
{
float l,b;
printf("Enter the length and breadth of rectangle\n");
scanf("%f%f",&l,&b);
area(l,b);
perimeter(l,b);
return 0;
}
void area(float x,float y)
{
          float area;
          area=x*y;
          printf("area=%f\n",area);
}

void perimeter(float x,float y)
{
float peri;
peri=2*(x+y);
printf("Perimeter=%f",peri)    ;
}
```

5. Write a program to check maximum of 3 numbers using user-defined function.

```
#include<stdio.h>
int max(int,int,int);
int main()
{
int a,b,c,result;
printf("Enter the three numbers\n");
scanf("%d%d%d",&a,&b,&c);
result=max(a,b,c);
printf("Maximum number=%d",result);
return 0;
}

int max(int x,int y,int z)
{
    if(x>y&&x>z)
    {
    return x;
    }
    else if(y>z)
    {
    return y;
    }
    else
    {
    return z;
    }
}
```

Write a Program to check the maximum of two numbers using function.(Try yourself)

6. Write a Program to check the given number is prime or not using user defined function.

```
#include<stdio.h>
void checkprime(int n);
int main()
{
int num;
printf("Enter the number you want to check\n");
scanf("%d",&num);
checkprime(num);
return 0;
}
```

```
void checkprime(int n)
{
    int i;
    for(i=2;i<n;i++)
    {
      if(n%i==0)
            {
            printf("Number is not prime");
            break;
            }
     }
    if(i==n)
    {
    printf("Number is prime");
    }
 }
```

7. Write a program to check given number is palindrome or not using user-defined function.

```
#include<stdio.h>
void checkpalindrome(int n);
int main()
{
int num;
printf("Enter the number you want to check\n");
scanf("%d",&num);
checkpalindrome(num);
return 0;
}
void checkpalindrome(int n)
{
        int rem,rev=0,a;
        a=n;
        while(n!=0)
        {
                rem=n%10;
                rev=rev*10+rem;
                n=n/10;
        }
        if(a==rev)
        {
        printf("Number is palindrome");
        }
        else
        {
        printf("Number is not palindrome");
        }
 }
```

Write a Program to find the reverse of a number using user-defined function. (Try yourself)

8. By using user-defined function write a program to generate the Fibonacci series upto nth term when initial value is given by user.

```
#include<stdio.h>
void fibo(int n,int x,int y);
int main()
{
int a,b,num;
printf("Enter the number of terms you want to generate\n");
scanf("%d",&num);
printf("Enter the two inital values\n");
scanf("%d%d",&a,&b);
fibo(num,a,b);
return 0;
}
void fibo(int n,int x,int y)
{
        int i,c;
        printf("%d\t%d",x,y);
        for(i=1;i<=n-2;i++)
        {
                c=x+y;
                printf("\t%d",c);
                x=y;
                y=c;
        }
}
```

Passing Array to Function

Passing one dimensional Array to function

In C, arrays are automatically passed by reference to a function. The name of the array represents the address of its first element. By passing the array name, we are in fact, passing the address of the array to the called function. The array in the called function now refers to same array stored in the memory. Therefore any changes in the array in the called function will be reflected in the original array.

- ✓ The corresponding formal argument in function definition is written in the same manner, though it must be declared as an array.
- ✓ When declaring a one-dimensional array as a formal argument, the array name is written with a pair of empty square brackets. The size of the array is not specified within the formal argument declaration.
- ✓ To pass an array as an argument to function, the array name must be specified, without brackets or subscripts, and the size of an array as arguments.

Syntax for function declaration that receives array as argument

return_type function_name(data_type array_name[ ]);

Syntax to pass array to the function.

function_name(arrayname);

Note: We cannot pass a whole array by value as we did in the case of ordinary variables.

Program to illustrate passing array to a function one element at a time.

```
#include<stdio.h>
void display(int x);
int main()
{
        int i;
        int a[5]={5,10,15,20,25};
        for(i=0;i<5;i++)
        {
                display(a[i]);
        }
        return 0;
}
void display(int x)
{
printf("%d\t",x);
}
```

Write a program to pass one dimensional array to a function and display that array in that called function.

Program to illustrate passing an entire array to a function

```
#include<stdio.h>
void display(int x[],int n);
int main()
{
        int i;
        int a[5]={5,10,15,20,25};
        display(a,5);
        return 0;
}
void display(int x[],int n)
{
        int i;
        for(i=0;i<n;i++)
        {
                printf("%d\t",x[i]);
        }
}
```

1) Write a Program to pass the elements of an array to a function using pointer and use the function to find the sum of elements and print the sum.

```
#include<stdio.h>
void add(int x[],int n);
int main()
{
        int a[100],n,i;
        printf("Enter how many elements\n");
        scanf("%d",&n);
        printf("Enter %d array elements\n",n);
        for(i=0;i<n;i++)
        {
                scanf("%d",&a[i]);
        }
        add(a,n);
        return 0;
}
void add(int x[],int n)
{
        int i,sum=0;
        for(i=0;i<n;i++)
        {
        sum=sum+x[i];
        }
        printf("sum =%d",sum);
}
```

2) Using user defined function write a program to input n numbers in an one dimensional array and find the average of the numbers**.**

```
#include<stdio.h>
void average(int x[],int n);
int main()
{
    int a[100],n,i;
    printf("Enter how many elements\n");
    scanf("%d",&n);
    printf("Enter the array elements\n");
    for(i=0;i<n;i++)
    {
            scanf("%d",&a[i]);
    }
    average(a,n);
    return 0;
}
```

```
void average(int x[],int n)
{
	int i,sum=0;
	float avg;
	for(i=0;i<n;i++)
	{
	sum=sum+x[i];
	}
	avg=(float)sum/n;
	printf("Average=%f",avg);
}
```

3) Write a program that passes array to function and print the largest and smallest element.

```
#include<stdio.h>
void fun(int x[],int n);
int main()
{
	int a[100],n,i;
	printf("Enter how many elements\n");
	scanf("%d",&n);
	printf("Enter the array elements\n");
	for(i=0;i<n;i++)
	{
		scanf("%d",&a[i]);
	}
	fun(a,n);
	return 0;
}
void fun(int x[],int n)
{
	int i,small,large;
	large=x[0];
	small=x[0];
	for(i=0;i<n;i++)
	{
		if(large<x[i])
		{
			large=x[i];
		}
		if(small>x[i])
		{
			small=x[i];
		}
	}
	printf("Largest element =%d",large);
	printf("Smallest element=%d",small);
}
```

4) Write a program to read n number in an array and sort them in ascending order using function.

```
#include<stdio.h>
void sort(int x[],int n);
void disp(int d[],int m);
int main()
{
	int a[100],n,i;
	printf("Enter how many elements\n");
	scanf("%d",&n);
	printf("Enter the array elements\n");
	for(i=0;i<n;i++)
	{
		scanf("%d",&a[i]);
	}
	printf("Array elements before sorting are\n");
	disp(a,n);
	sort(a,n);
	printf("Array elements after sorting are\n");
	disp(a,n);
	return 0;
}
void sort(int x[],int n)
{
	int i,j,temp;
	for(i=0;i<n-1;i++)
	{
		for(j=0;j<n-1-i;j++)
		{
			if(x[j]>x[j+1])
			{
			temp=x[j];
			x[j]=x[j+1];
			x[j+1]=temp;
			}
		 }
	}
}
void disp(int d[],int m)
{
	int i;
	for(i=0;i<m;i++)
	{
		printf("%d\n",d[i])	;
	}

}
```

5) Write a program to find the sum of all prime numbers in a given array. The main function of your program should take the help of user-defined function that tests whether a given number is prime or not.

```
#include<stdio.h>
int checkprime(int n);
int main()
{
   int a[50],n,i,sum=0,result;
   printf("Enter the no. of elements\n");
   scanf("%d",&n);
   for(i=0;i<n;i++)
   {
      scanf("%d",&a[i]);
   }
  printf("prime numbers are\n");
  for(i=0;i<n;i++)
   {
          result=checkprime(a[i]);
          if(result==1)
          {
            printf("%d\t",a[i]);
            sum=sum+a[i];
          }
   }
   printf("The sum of prime no. is %d\n",sum);
}

int checkprime(int n)
{
  int i;
  for ( i = 2 ; i <n ; i++ )
  {
    if ( n% i == 0 )
    return 0;
  }
  if ( n == i )
  {
    return 1;
  }
}
```

Practice Questions:

1) Write a program to input n numbers in an array and count number of odd and even numbers and find their sum using function.
2) Write a Program to input n number in an array and print in reverse order using function.
3) Write a Program to input n numbers in an array and check if given number is present or not using function.

Passing 2D array to function

- Just as in 1 dimensional array, when two-dimensional array is passed as a parameter, the base address of the actual array is sent to function.
- Any change made to element inside function will carry over to the original location of array that is passed to function.
- The size of all dimensions except the first must include in the function declaration and in function definition.

Eg. void display(int[][10],int, int);

is a valid declaration.

1) Write a program to input m*n order matrix and find sum of all elements using user-defined function.

```
#include<stdio.h>
void summatrix(int s[][20],int x,int y);
void disp(int d[][20],int m,int n);
int i,j;
int main()
{
	int a[20][20],r,c;
	printf("Enter the row and column size of matrix\n");
	scanf("%d%d",&r,&c);
	printf("Enter the array elements\n");
	for(i=0;i<r;i++)
	{
		for(j=0;j<c;j++)
		{
		scanf("%d",&a[i][j]);
		}
	}
	disp(a,r,c);
	summatrix(a,r,c);
	return 0;
}
void summatrix(int s[][20],int x,int y)
{
	int sum=0;
	for(i=0;i<x;i++)
	{
		for(j=0;j<y;j++)
		{
			sum=sum+s[i][j];
		}
	}
	printf("sum of all elements=%d\n",sum);
}
```

```
void disp(int d[][20],int m,int n)
{
        printf("Entered matrix is\n");
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        printf("%d\t",d[i][j]);
                }
                printf("\n");
        }
}
```

2) Write a user defined function to find the sum of elements of m*n matrix and finally returns the sum to the calling function.

```
#include<stdio.h>
int summatrix(int s[][20],int x,int y);
void disp(int d[][20],int m,int n);
int i,j;
int main()
{
        int a[20][20],m,n,result;
        printf("Enter the row and column size of matrix\n");
        scanf("%d%d",&m,&n);
        printf("Enter the array elements\n");
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                scanf("%d",&a[i][j]);
                }
        }
        disp(a,m,n);
        result=summatrix(a,m,n);
        printf("sum of matrix elements=%d",result);
        return 0;
}
int summatrix(int s[][20],int x,int y)
{
        int sum=0;
        for(i=0;i<x;i++)
        {
                for(j=0;j<y;j++)
                {
                        sum=sum+s[i][j];
                }
        }
        return sum;
}
```

```
void disp(int d[][20],int m,int n)
{
        printf("Entered matrix is\n");
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        printf("%d\t",d[i][j]);
                }
                printf("\n");
        }
}
```

3) Write a program to input m*n order matrix and find its transpose using user-defined function.

```
#include<stdio.h>
void transpose(int t[][20],int x,int y);
void disp(int d[][20],int m,int n);
int i,j;
int main()
{
        int a[20][20],r,c;
        printf("Enter the row and column size of matrix\n");
        scanf("%d%d",&r,&c);
        printf("Enter the array elements\n");
        for(i=0;i<r;i++)
        {
                for(j=0;j<c;j++)
                {
                scanf("%d",&a[i][j]);
                }
        }
        disp(a,r,c);
        transpose(a,r,c);
        return 0;
}
void transpose(int t[][20],int x,int y)
{
        printf("Transpose of matrix is \n");
        for(i=0;i<y;i++)
        {
                for(j=0;j<x;j++)
                {
                printf("%d\t",t[j][i]);
                }
                printf("\n");
        }
}
```

```
void disp(int d[][20],int m,int n)
{
	printf("Entered matrix is\n");
	for(i=0;i<m;i++)
	{
		for(j=0;j<n;j++)
		{
			printf("%d\t",d[i][j]);
		}
		printf("\n");
	}
}
```

3.Write a program to perform addition of two m*n order matrix using user-defined function

```
#include<stdio.h>
void input(int a[][20],int x,int y);
void disp(int d[][20],int m,int n);
void addition(int a1[][20],int a2[][20],int p,int q);
int i,j;
int main()
{
	int matrix1[20][20],matrix2[20][20],r,c;
	printf("Enter the row and column size of matrix\n");
	scanf("%d%d",&r,&c);
	printf("Enter the first matrix");
	input(matrix1,r,c);
	printf("Enter the second matrix");
	input(matrix2,r,c);
	printf("The first matrix is\n");
	disp(matrix1,r,c);
	printf("The second matrix is\n");
	disp(matrix2,r,c);
	addition(matrix1,matrix2,r,c);
	return 0;
}
void input(int a[][20],int x,int y)
{
	for(i=0;i<x;i++)
	{
		for(j=0;j<y;j++)
		{
		scanf("%d",&a[i][j]);
		}
	}
}
```

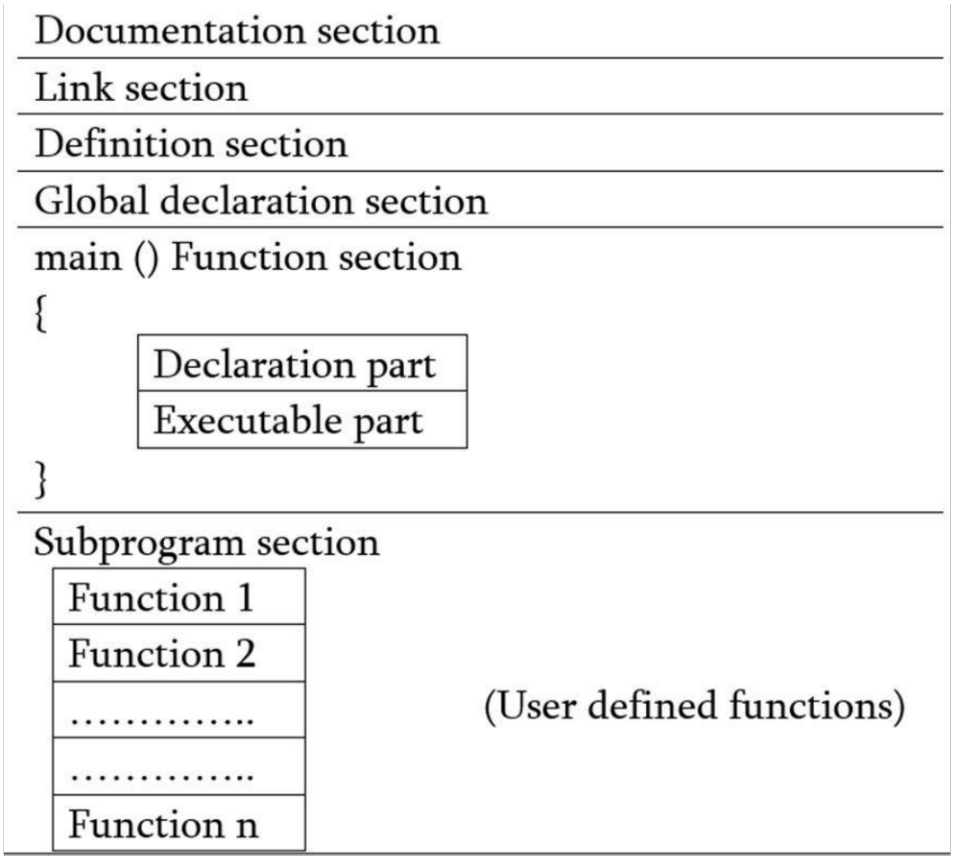
```
void disp(int d[][20],int m,int n)
{
	for(i=0;i<m;i++)
	{
		for(j=0;j<n;j++)
		{
			printf("%d\t",d[i][j]);
		}
		printf("\n");
	}
}

void addition(int a1[][20],int a2[][20],int p,int q)
{
	int sum[20][20];
	for(i=0;i<p;i++)
	{
		for(j=0;j<q;j++)
		{
			sum[i][j]=a1[i][j]+a2[i][j]	;
		}
	}
	printf("Resultant matrix is\n");
	disp(sum,p,q);
}
```

Practice Questions.

1. Write a Program to input m*n order matrix and count even number of elements present in matrix and find their sum using function.
2. Write a Program to input m*n matrix and find highest and lowest element using function.
3. Write a Program to input m*n order matrix and find highest and lowest element using function.

Basic structure of C program

A C program can be viewed as a group of building blocks called functions. A function is a subroutine that may include one or more statements designed to perform a specific task. To write a C program, we first create functions and then put them together. A C program may contain one or more sections as shown in figure.



- ✓ The documentation section consists of a set of comment lines giving the name of the program, the author and other details, which the programmer would like to use later.
- ✓ The link section provides instructions to the compiler to link functions from the system library.
- ✓ The definition section defines all symbolic constants.
- ✓ There are some variables that are used in more than one function. Such variables are called global variables and are declared in the global declaration section that is outside of all the functions. This section is also declared all the user defined functions.
- ✓ Every C program must have one **main()** function section .This section contains two parts, declaration part and executable part. The declaration part declares all the variables used in the executable part. There is at least one statement in the executable part. These two parts must appear between the opening and closing braces. The program execution begins at the opening brace and ends at the closing brace. The closing brace of the main function section is the logical end of the program. All statements in the declaration and executable parts end with the semicolon (;).
- ✓ The subprogram section contains all the user-defined functions that are called in the main function. User defined functions are generally placed immediately after the main function, although they may appear in any order.
- ✓ All sections except main function section may be absent when they are not required.