Modular Programming:The Functions

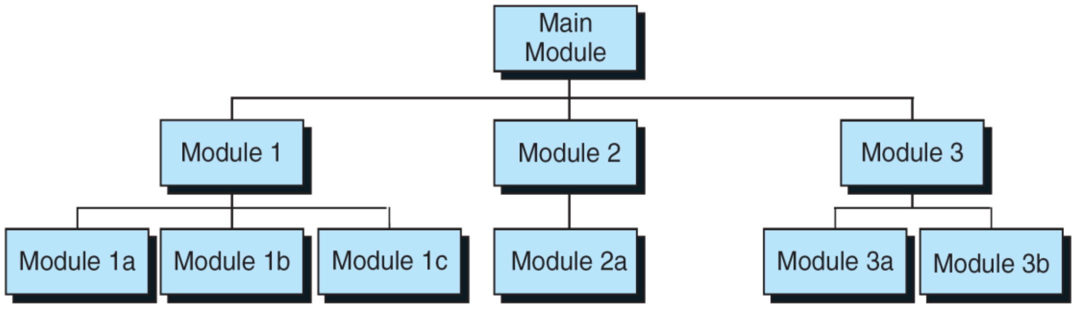
- **Function** type is a derived data type.
- It is used to support top-down design of structured programming approach (modular programming approach).

Top – Down Design:-

It is used for large & complex programs generally. Here first we have to understand the problem fully. Then we have to break (& divide) the problem into smaller & easy parts. These parts of a program (problem) are called modules(functions). This process of subdividing a problem is – top-down design.

In Top-down design, a program is divided into a main module & it is again divided into sub-modules & this process continues until a sub-module is so easy that there is no need to divide it further.

Top-down design is represented in a visual form, & it is called –*Structure chart*



Here in above structure chart we have divided our main module into 3 sub-modules – module1, module2 & moudle3.Then further module1 is again divided into 3 sub-modules.

So here we are moving from Top module-(main) to submodules – m.1a…..m3b.

- Here main module is called as – a calling module.
- Here module 1 is known as – a called module.

  but for sub-module module1c, module1 is called module.
- Communication (Data Transferring) between modules & sub-modules-

  If module1a wants to say something (transfer any data) to module 3b then –
- m-1a will send data to main then module1 sends data to main module, which passes data to module3, and then data is sent to module3b.

  "The tech. used to insert data in to a function is known as – "parameter passing".

Features (characteristics) of Modular Programming

1. Each module should be only one thing.
2. Communication between modules is allowed only by a *calling* module (to a *called* module)
3. A module can be called by one & only one higher module.
4. No communication can take place directly between modules that do not have *calling-called* relationship.
5. All modules are designed as single-entry, single-exit systems using control structures.
6. Modules are practically implemented using functions in c programming.

Facts about functions :-

1. Any C program contains at least 1 function.
2. If a program has only 1 function, then it must be main(), so main() is one most essential function, in any program, since execution of program, starts with main().

3. A C program can have many (unlimited) functions.
4. In main() we decide the sequence of calling of different functions.
5. When the called function completes (executes) its tasks, it returns the control to calling function. Finally all function called by main() return control to main() & when main() is complete, control returns to operating system.
6. *A function cannot be defined within another function.*

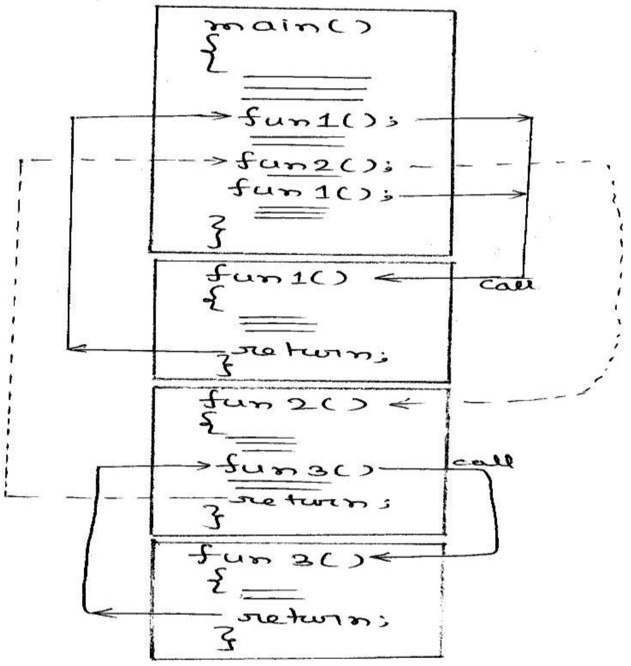
Advantages of Functions in C-

1. Functions support Top-down modular programming.
2. Functions provide-code reusability, so the source program size can be reduced.
3. Functions support easy debugging of program to find & remove errors.
4. Functions (library function) support-easy & less tedious program creation.
5. Function support-data protection. It is generally for the local data to that particular function.

Functions are of 2 types- User defined & predefined (system defined as: printf(), scanf() etc.).
User defined Functions:-

- Like any other variable, functions are also *declared, defined* and then *used by calling* them.
- Declaration is done before calling (using) a function, & here we give name of function, return type, type & order of formal parameters of function. In declaration we use only name of function & terminate it with semicolon.
- Definition of function is coded outside the main(), which has the body of function - complete description of the task.

Flow of control in a multi-function program:



Three aspects of working(operations) with user-defined functions

- Function declaration
- Function definition
- Function use/call

Function declaration (or function prototyping)

All the functions need to be declared or defined before they are used. The general form of **function declaration** is:

[return_type] **function_name(**[parameter_list or parameter_type_list]**) ;**

The terms enclosed within the **square brackets** are **optional** and terms shown in **bold** are the **mandatory parts** of a function declaration.

important points about the function declaration:

1. *Function names are **identifiers/variables***. All syntactic rules discussed earlier for writing identifier names are applicable for writing the function names as well. The name of a function is also termed as **function designator**.
2. *The specification of the return type is optional*. If specified, the return type of a function can be any type (e.g. char, int, float, double, int*, int**, void etc.) except array type and function type.
3. *Following are the syntactic rules for writing **parameter type list** and **parameter list** in a function declaration*:
   - The **parameter type list** is a comma separated **list of parameter types** e.g. char, int, float, int*, int**, void etc. except function type. This is **abstract parameter declaration.**
   - **A parameter name can optionally follow each parameter type.** If parameter names follow parameter types in a parameter type list, it becomes **parameter list**. This is **complete parameter declaration.**
   - **Using a combination of complete parameter declaration and abstract parameter declaration is also allowed**. For example:

- int add(int x, int); //allowed
- int add(int, int y); //allowed

   - *No two parameter names appearing in the parameter list can be same. The shorthand declaration of parameters in the parameter list is not allowed.*

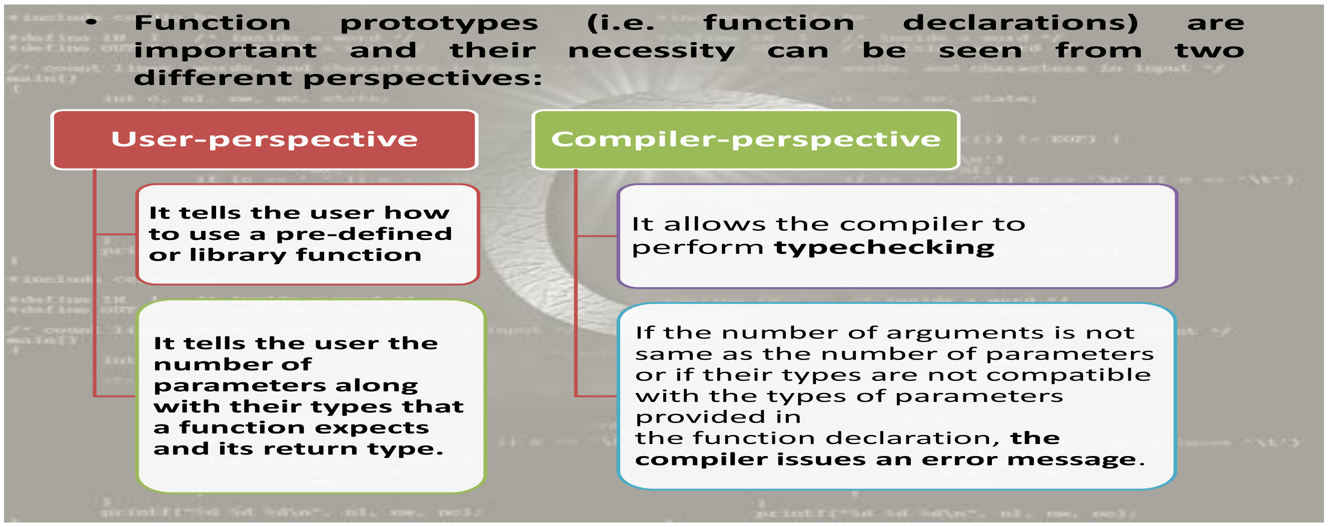
4. **Function declaration is a statement, so it must be terminated with a semicolon. A function need not be declared, if it is defined before it is called.**

- **The following function declarations are valid:**

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

- **The following function declarations are not valid:**

```
int add(int a, float a);
int add &sub(int, int);  //←Name of the function is not valid as it contains the special character &
int add(int a,b);        //←Shorthand declaration of parameters is not allowed
int add (int a, int b)   //←The declaration is not terminated with a semicolon
```



- **Function prototypes (i.e. function declarations) are important and their necessity can be seen from two different perspectives:**

User-perspective

- **It tells the user how to use a pre-defined or library function**
- **It tells the user the number of parameters along with their types that a function expects and its return type.**

Compiler-perspective

- It allows the compiler to perform **typechecking**
- If the number of arguments is not same as the number of parameters or if their types are not compatible with the types of parameters provided in the function declaration, **the compiler issues an error message.**

Definition of Function(function implementation)***:***

In C, function (including main()) is an independent module that will be called to do a specific task.

Function definition means composing a function.The function definition has code for a function. This code is made of 2 important parts

1. Function header (or signature)

    a- Name of function

    b- return_type of a function (by default integer)

    c- list of parameters (arguments) passed in function or void(no parameter passed).

Function header:

    <return_type> function_name(<parameter list>)

2. Function body

    a- local variable declaration.

    b- statements (to be executed).

    c- return statement.

- In definition of function we never use semicolon, (to terminate) in its header.
- But we use semicolon in all (a, b, c) parts of body of function.

Function body:

```
{
local declarations;
statements to be executed;
:
return statement;
}
```

eg:

```
int f1  (---)                         void f1  (---)
{                                       {
   ----------                              ---------
   return(x+2); }                           return;}
```

Integer (X+2) is returned *nothing is returned*

- We should use keyword return, at the end of function, even if it does not return anything.
- **Unlike function declaration, the header of a function can only have complete parameter declaration**. The variables declared in the parameter list will receive the data sent by the calling function. They serve as the inputs to the function.
- List of parameters – In the function-definition the parameters passed are called as – “Formal Parameters”.
- **No two parameter names appearing in the parameter list can be same.**
- **The shorthand declaration of parameters in the parameter list is not allowed.**
- The return type and, the number and the types of parameters in the function header should exactly match the corresponding return type and, the number and types of parameters in the function declaration, if it is present
- **It is not mandatory to have the same names for the parameters in the function declaration and function definition.**
- If no parameters are passed in function definition then we may use “void”.

Body of a function

- The **body of a function** consists of a set of statements enclosed within curly braces. **The body of a function can have non-executable statements and executable statements** .
- The non-executable statements declare the local variables of function and the executable statements determine its functionality i.e. what the function does. **A function can optionally have special executable statement known as return statement**.
- **Remember that non-executable statements can only come *before* executable statements.**

“Any program designed without using function can be designed using functions too.”

2 values received
from calling function

eg:
```
double Avg(int x, int y)
{
double sum;
sum = x+y;
return (sum/2);
}
```
1value returned to calling function

Program without using function	**Program using function**
main()	double Avg(int x, int y)
{	{ // Avg is user defined function.
double a=15,b=20,sum,result;	double sum;
sum = a+b;	sum = x+y;
result=sum/2;	return (sum/2);
printf("Average is=%lf",result);	}
return 0;	main()

}	{
	double a=15,b=20,result;
	result=Avg(a,b);
	printf("Average is=%lf",result);
	}

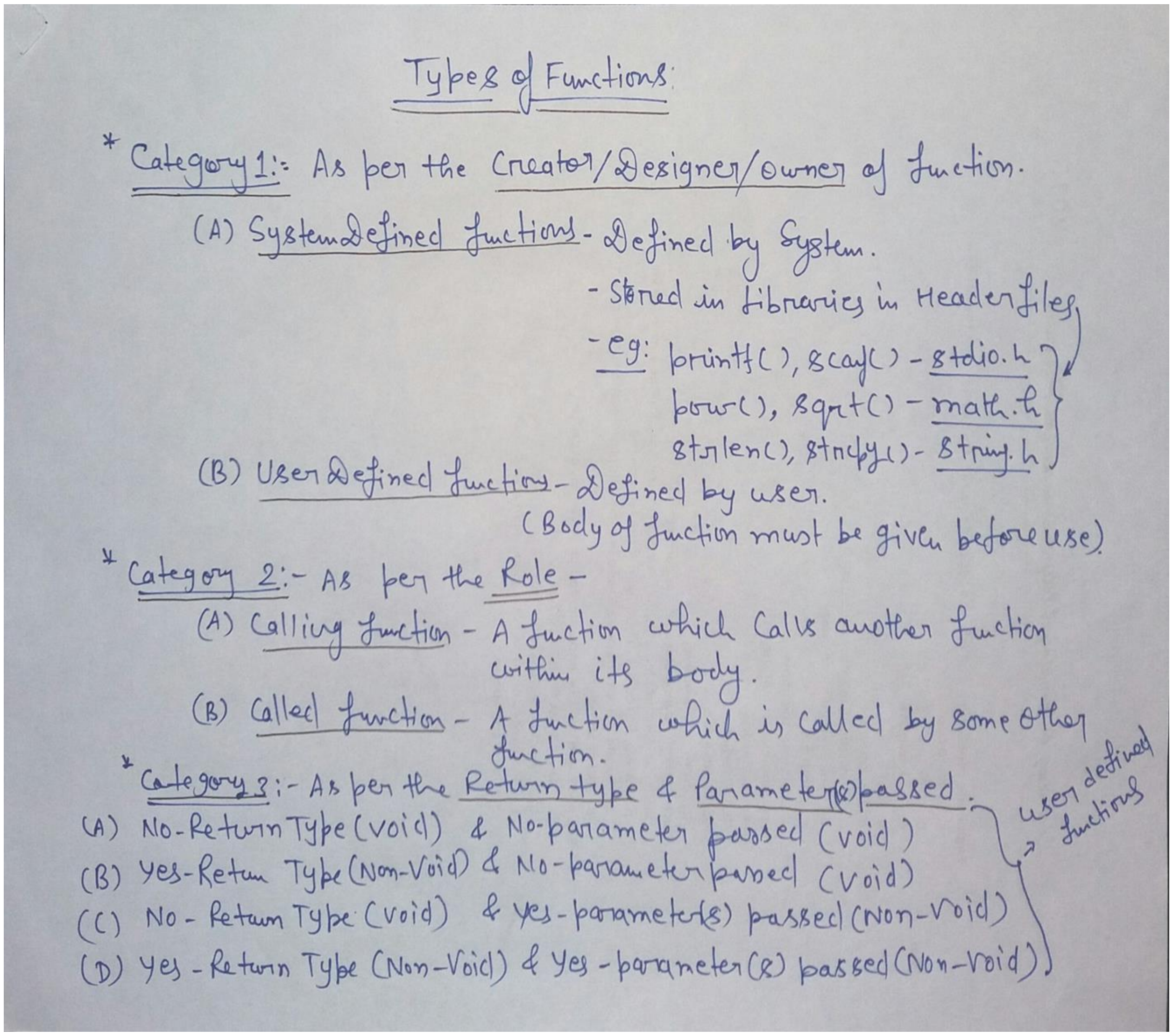
In above program– x & y are formal parameters. a, b are actual parameters used while calling function in main

- Sum is a local variable to the function Avg(), because it is declared, defined & used within this function, so out side the Avg(), sum is still unknown & not-reachable.
- The return statement can be used as any of the following:

```
return (variable_name) ≡ return (p);
return(expression)      ≡ return (x*y);
if(cond?)                 if (a >b)
return (p);      ≡        return (a);
else                      else
return (q);               return (b);
```

- Formal parameters (arguments) are variables that are given in the header of function, while **defining** the function.
- Actual parameters are variables that are given in the header of function, while **calling** the function. These are also called as arguments.

$$\left.\begin{array}{l}\text{Data\_type,}\\ \text{Order (sequence)}\\ \text{\& Number (counting}\end{array}\right\}\begin{array}{l}\text{of actual Parameters}\\ \text{must be equal to Formal parameters.}\end{array}$$



Types of Functions:

* Category 1:- As per the Creator/Designer/owner of function.

(A) System Defined functions - Defined by System.

- Stored in Libraries in Header files
- eg: printf(), scanf() - stdio.h
  pow(), sqrt() - math.h
  strlen(), strcpy() - string.h

(B) User Defined functions - Defined by user.
(Body of function must be given before use)

* Category 2:- As per the Role -

(A) Calling function - A fuction which Calls another fuction within its body.

(B) Called function - A function which is called by some other function.

* Category 3:- As per the Return type & Parameter(s) passed:

(A) No-Return Type (void) & No-parameter passed (void)
(B) yes-Return Type (Non-Void) & No-parameter passed (void)
(C) No-Return Type (void) & yes-parameter(s) passed (Non-void)
(D) yes-Return Type (Non-Void) & yes-parameter(s) passed (Non-Void)

user defined functions

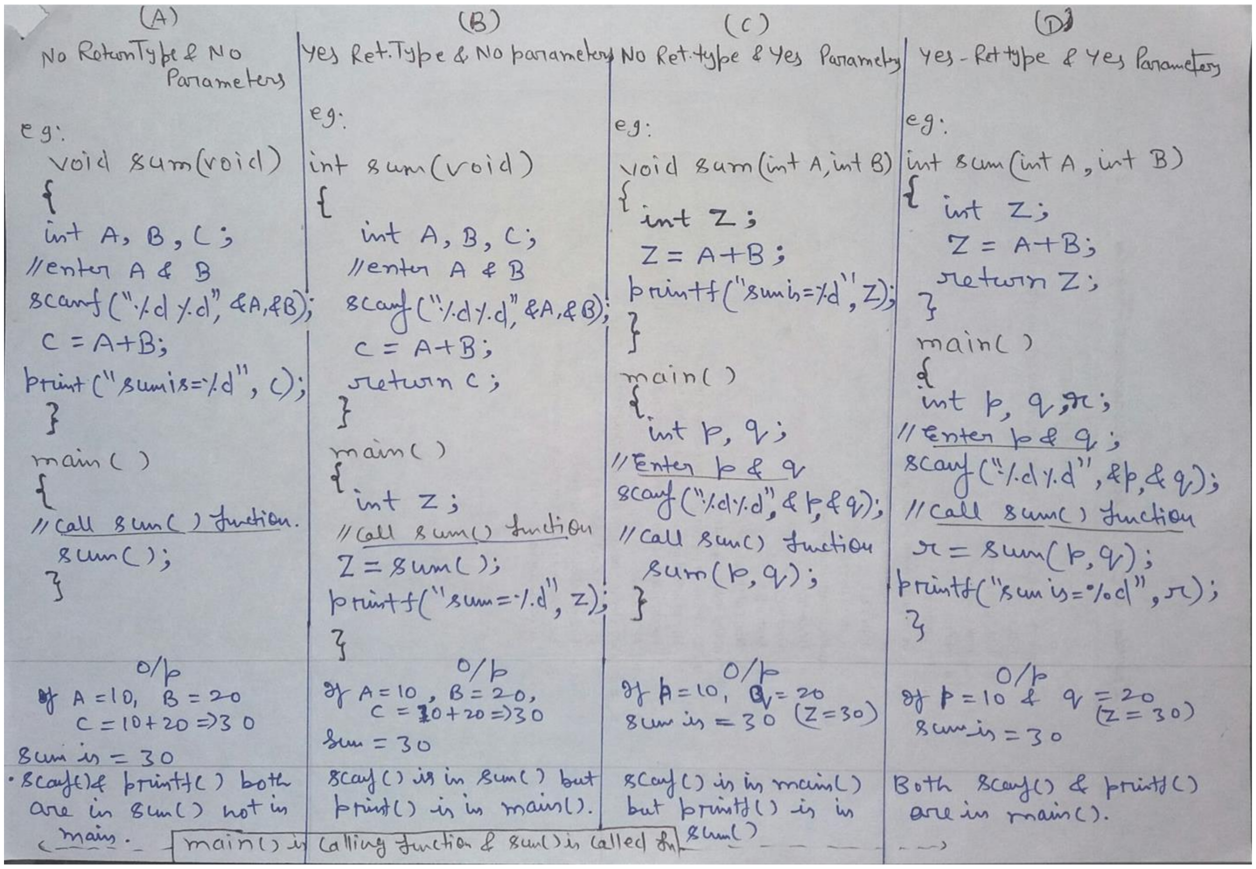
Function invocation/ call/ use/ implementation

The call to a function can be well described along with the discussion on the classification of functions. Depending upon their inputs (i.e. parameters) and outputs (return type), functions are classified as:

- **i.) Functions with no input-output (no parameter passed & no return type)**
- **ii.) Functions with yes output and no input (Yes return type & no parameter passed)**
- **iii.) Function with no output and yes input (no return type & yes parameter passed)**
- **iv.) Function with inputs and outputs (yes return type & yes parameter passed)**

facility of input is absent=> scanf() is present and output(return) is absent=> printf() is present.

void ***is used in place of*** **no** ***and non-void or*** **some value** ***is used for representing*** **yes.**



(A)

No Return Type & No Parameters

eg:

```
void sum(void)
{
int A, B, C;
//enter A & B
scanf("%d %d", &A,&B);
 C = A+B;
print("sum is=%d", C);
}
main ( )
{
// call sum( ) function.
  sum( );
}
```

O/p

If A =10, B = 20
C = 10+20 =>30
sum is = 30

• scanf() & printf( ) both are in sum( ) not in main.

(B)

Yes Ret.Type & No parameters

eg:

```
int sum(void)
{
   int A, B, C;
   //enter A & B
   scanf("%d %d", &A,&B);
    C = A+B;
   return C;
}
main( )
{
  int Z;
  // call sum() function
  Z = sum( );
printf("sum=%d", Z);
}
```

O/p

If A = 10, B = 20,
C = 10+20 =>30
sum = 30

scanf () is in sum( ) but print() is in main().

main() is calling function & sum() is called fn

(C)

No Ret.type & Yes Parameters

eg:

```
void sum(int A, int B)
{
 int Z;
 Z = A+B;
printf("sum is=%d", Z);
}
main( )
{
 int p, q;
//Enter p & q
scanf("%d%d", &p,&q);
// call sum() function
  sum(p,q);
}
```

O/p

If p=10, q= 20
sum is = 30 (Z=30)

scanf() is in main() but printf() is in sum()

(D)

Yes - Ret type & Yes Parameters

eg:

```
int sum (int A, int B)
{ int Z;
  Z = A+B;
  return Z;
}
main( )
{
 int p, q, r;
// Enter p & q;
scanf("%d %d", &p, & q);
// call sum( ) function
 r = sum(p,q);
printf("sum is=%d", r);
}
```

O/p

If p=10 & q = 20
sum is = 30 (Z = 30)

Both scanf() & printf() are in main().

IN above A,B are formal & p, q are actual parameters

There are two forms of the return statement:

- **return;**
- **return expression;**

1. **First form of return statement i.e. return;**

a) This form of return statement is used when a function does not return any value (i.e. inside void functions).
b) It cannot be used inside a function whose return type is not void.
c) It terminates the execution of the called function and transfers the program control back to the calling function without returning any value.

2. **Second form of return statement i.e. return expression;**
   a) **It cannot be placed inside the body of a void function and can only appear inside the body of a function whose return type is not void.**
   b) The expression following keyword return in return statement is known as **return expression**.
   c) The return expression can be an arbitrarily complex expression and can even have function calls. For example, in the statement return n*fact(n-1);
   d) **The return expression is evaluated and the result of evaluation of the return expression is returned back.**
   e) If the return type of a function and the type of the result of evaluation of return expression is not same, **the result of evaluation of the return expression is implicitly type-casted to the**

return type of the function, if they are compatible. If they are incompatible, there will be a compilation error

3. **There is no constraint on the number of return statements that can be placed inside a function's body.** Although, a number of return statements can be placed inside the body of a function, but only one of them which appears first in the logical flow of control gets executed and the **rest of the statements that appear after this return statement remains unreachable**.

4. **"A function can return only one value"**. It is not possible to return more than one value by writing multiple return statements as mentioned in the point 3 above or by writing return value1, value2, …….valueN;.

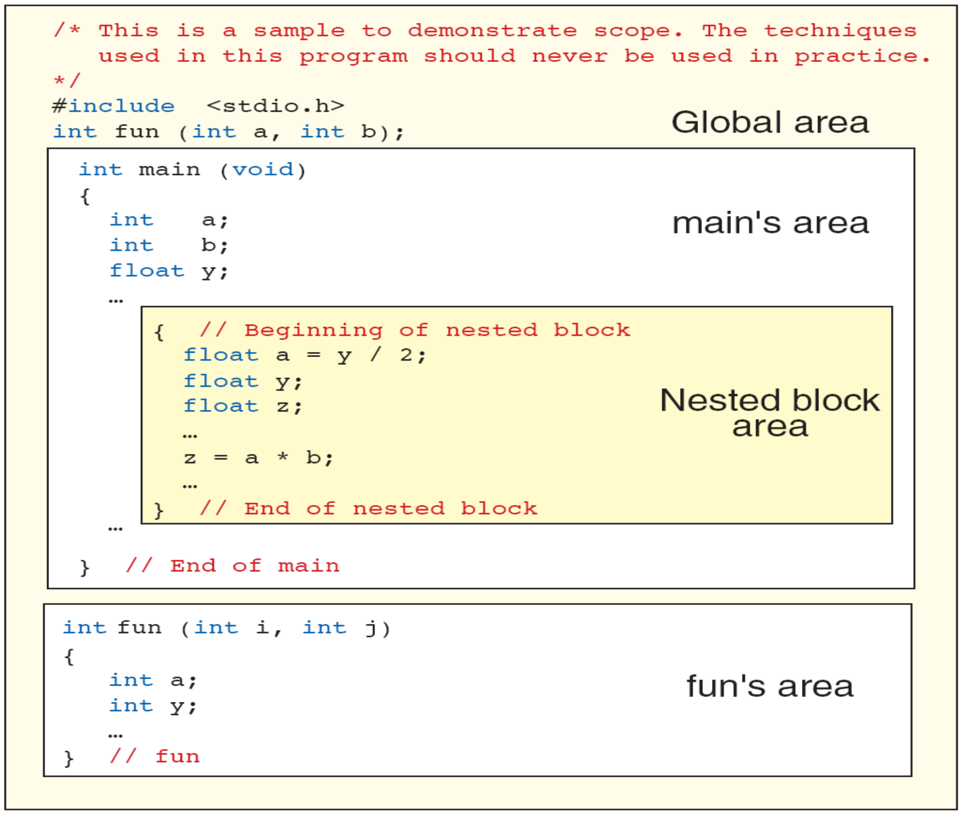
The return expression consists of comma operators. The comma operator guarantees left-to-right evaluation and returns the result of the rightmost sub-expression. Hence, the expression value1, value2, …….valueN evaluates to valueN and this value is returned.

Types of Variables used in functions

- *Local variable* – A variable defined & used within a function body is local to that function, this variable will not be accessible from out of the function.
- *Global variable* – A variable that is declared & defined outside of the functions, it is generally declared out of main also. It can be accessed and changed by all the functions in a program.

Scope : Scope determines the region of the program in which a defined object is visible (accessible). Here objects are variables & functions.

- Scope is a source bearing effect on the run-time program.
- A block encloses '0' or more statements within it under a set of braces ({}). So a function body is a block.
- The global area of our program has all statements that are out-side the functions.
- A variable is in scope if it is visible to the statement being examined.
- Variables are in scope from the point of declaration until the end of their block.



```
/* This is a sample to demonstrate scope. The techniques
   used in this program should never be used in practice.
*/
#include  <stdio.h>
int fun (int a, int b);
  int main (void)
  {
    int   a;
    int   b;
    float y;
    …
       {  // Beginning of nested block
         float a = y / 2;
         float y;
         float z;
         …
         z = a * b;
         …
       }  // End of nested block
    …
  }  // End of main

 int fun (int i, int j)
 {
    int a;
    int y;
    …
 }  // fun
```

a, defined in main is integer, while it is float in nested block.
The y/2 will be- y is taken from main() until again we have defined y.

```
                 #include<stdio.h>
Global-variable  int n = 10;
                 void display ( );
                 void main ( )
                 {
Local variable   int x = 20;
                 printf ("\n%d", x);        //20 (local value)
                 display( );                //10 (global value)
                 }
                 void display( )                 OUTPUT: 20
                 {printf ("\n%d", n);}                    10
```

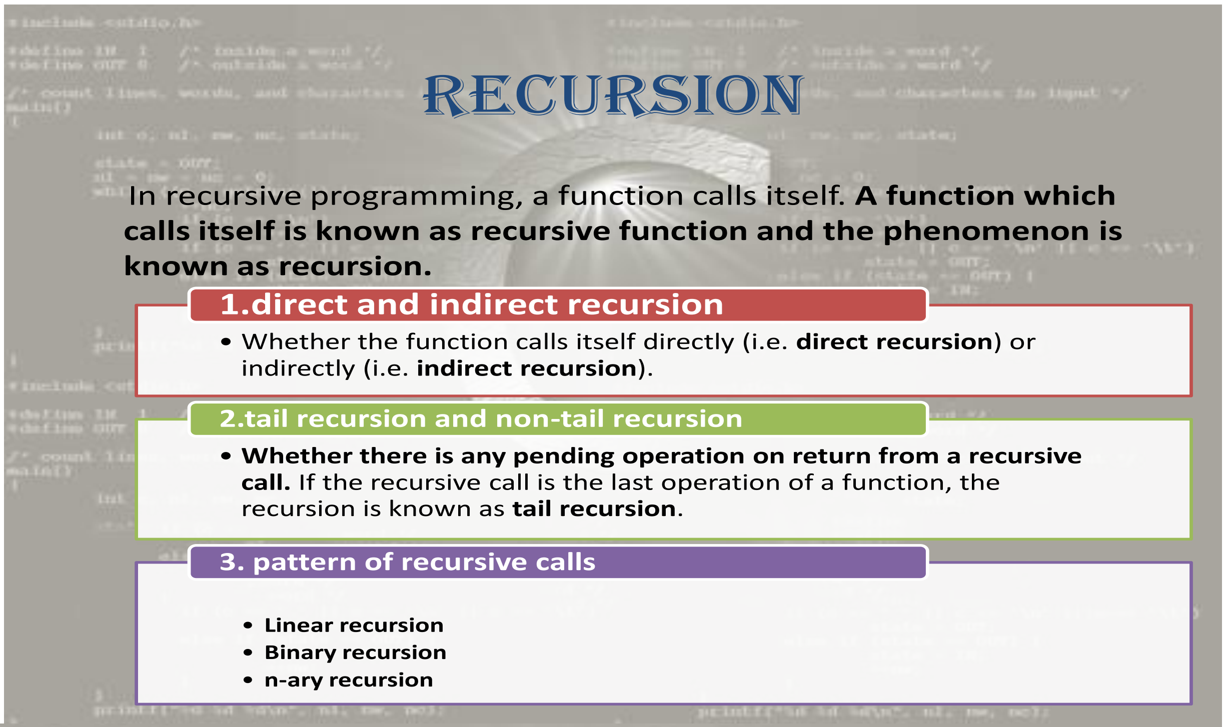
- A function can be called in another function, but a function cannot be defined in another function.
- Arguments are passed to a function in the right to left order.

For example:
int a = 1;
printf("%d%d%d", a, ++a, a++)
o/p will be: 3, 3, 1   ;   not: 1, 2 2

EXAMPLES of programs using functions: prime no, leap year, calculator, area & perimeter of shapes, Fibonacci series.

leap_year(int year)

```
{  if (year%4 == 0 && year%100 !=0 || (year%400 == 0) ) printf("\n LEAP YEAR");
   else printf("\n NOT A LEAP YEAR");  }
```



RECURSION

In recursive programming, a function calls itself. **A function which calls itself is known as recursive function and the phenomenon is known as recursion.**

1.direct and indirect recursion

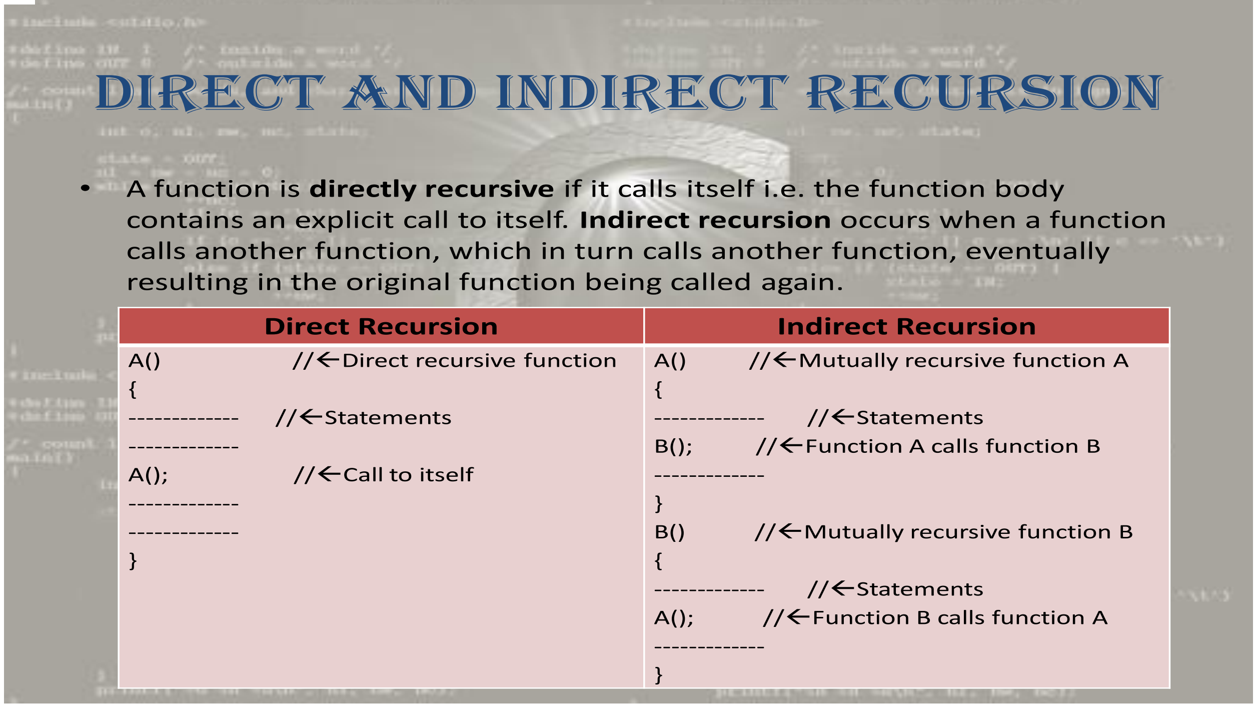
- Whether the function calls itself directly (i.e. **direct recursion**) or indirectly (i.e. **indirect recursion**).

2.tail recursion and non-tail recursion

- **Whether there is any pending operation on return from a recursive call.** If the recursive call is the last operation of a function, the recursion is known as **tail recursion**.

3. pattern of recursive calls

- **Linear recursion**
- **Binary recursion**
- **n-ary recursion**



DIRECT AND INDIRECT RECURSION

- A function is **directly recursive** if it calls itself i.e. the function body contains an explicit call to itself. **Indirect recursion** occurs when a function calls another function, which in turn calls another function, eventually resulting in the original function being called again.

Direct Recursion	Indirect Recursion
A() //←Direct recursive function	A() //←Mutually recursive function A
{	{
------------- //←Statements	------------- //←Statements
-------------	B(); //←Function A calls function B
A(); //←Call to itself	-------------
-------------	}
-------------	B() //←Mutually recursive function B
}	{
	------------- //←Statements
	A(); //←Function B calls function A

	}

Points about how to develop recursive functions

1. Thinking recursively is the first step to solve a problem using recursion.
2. Every recursive solution consists of two cases:

a) Base case: Base case forms the terminating condition of the recursion. Without base case, the recursion will never terminate and will be known as infinite recursion. For example, no= =1 is the base case of recursive fact function.

b) Recursive case: In recursive case, the problem is defined in terms of itself, while reducing the problem size. For example, when fact(n) is expressed as n×fact(n-1), the size of the problem is reduced from n to n-1.

3. Express the problem in the form of base cases and recursive cases.

4. Create the code of base cases and recursive cases separately using if-else statement

Tail recursion and non-tail recursion

Tail recursion is a special case of recursion in which the last operation of a function is a recursive call. In a tail recursive function, there are no pending operations to be performed on return from a recursive call.

Generally Recursion is referred as-Non-Tail Recursion

Example (Factorial Program):Non-tail Recursion and Tail-Recursion

//Non-tail recursive factorial function

```
#include<stdio.h>
int fact_norm(int);
void main()
{
     int no, fact;
   printf("Enter the number\t");
      scanf("%d",&no);
      fact=fact_norm(no);
printf("Resultant factorial is %d",fact);
}
```

//Non-tail recursive factorial function

```
int fact_norm(int no)
{
 if(no==1)
   return 1;
  else
   return no*fact_norm(no-1);
}
```

//Tail recursive factorial function

```
#include<stdio.h>
int fact_tail(int, int);
void main()
{
      int no, factorial;
printf("Enter the number\t");
      scanf("%d",&no);
      fact=fact_tail(no,1);
printf("Resultant fact is %d",fact);
}
```

//Tail recursive factorial function

```
int fact_tail(int no, int result)
{
      if(no==1)
          return result;
      else
return fact_tail(no-1,no*result) ;
}
```

Output window:

```
Enter the number
4
Resultant factorial
is 24
```

Remarks:

- ➢ fact_norm function in column 1 is a **non-tail recursive function.**
- ➢ **The last operation is a multiplication operation.**
- ➢ fact_tail function in column 2 is a **tail recursive function.**
- ➢ **The last operation of this function is a recursive function call.**

From above two types of recursion we can observe:

- The function fact_norm listed in **cloumn 1 is not tail recursive because there is a pending operation** i.e. multiplication to be performed on return from a recursive call.

- The function fact_tail listed in **column 2 is tail recursive as it has no pending operation on return from a recursive call**.
- Tail recursive functions can be easily transformed into iterative functions to improve the efficiency of a program.
- Tail recursion is desirable because the amount of information which must be stored during the computation is independent of the number of recursive calls. **Due to this, conversion of a non-tail recursive function to a tail recursive function is often required.**

Programs using recursion in c:

1. WAP to find sum of 1 to n numbers,n is entered by user.

```
#include <stdio.h>
int sum(int n);//function declaration or prototype
int main(){
   int num,add;
   printf("Enter a positive integer:\n");
   scanf("%d",&num);
   add=sum(num);
   printf("sum of numbers from 1 to %d=%d",num,add);
}
int sum(int n){
   if(n==0)       return 0;
   else       return n+sum(n-1);    /*It-self call of function sum() */
}
```

2. C Program to find Sum of Digits of a Number using Recursion

```
 #include <stdio.h>
 int sum (int num);
 main()
{
   int num, result;
    printf("Enter the number: ");
   scanf("%d", &num);
   result = sum(num);
   printf("Sum of digits in %d is %d\n", num, result);
 }
 int sum (int num)
{
   if (num == 0)  return 0;
   else        return (num % 10 + sum (num / 10));    }
```

WAP to print Fibonacci series up to nth term by recursion

```
#include<stdio.h>
 int Fibonacci(int);
 int main()
{   int n, c;
```

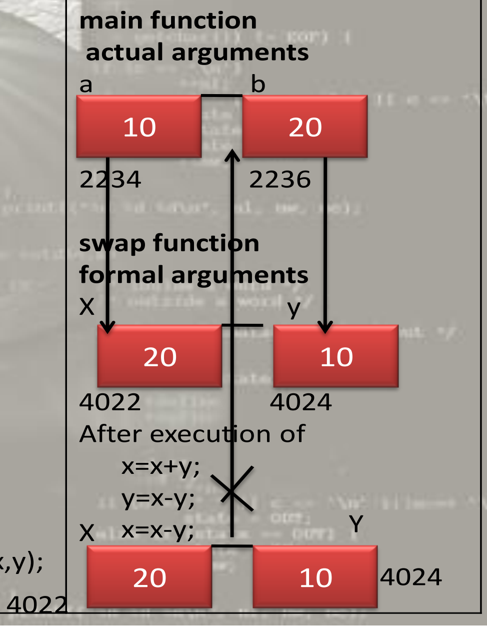
```
   printf("enter the last term (nth number): \n");
 scanf("%d",&n);
   printf("Fibonacci series up to nth number: \n");
   for ( c = 0 ; c < n ; c++ )
  {
     printf("%d ", Fibonacci(i));
      }
}
 int Fibonacci(int n)
{
  if ( n == 0 ||n == 1)
        return n;
  else   return ( Fibonacci(n-1) + Fibonacci(n-2));
}
```

Difference between Recursion and Iteration

S.No.	RECURSION	ITERATION
1	Recursive function – is a function that is partially defined by itself	Iterative Instructions –are loop based repetitions of a process
2	Recursion Uses selection structure	Iteration uses repetition structure
3	Infinite recursion occurs if the recursion step does not reduce the problem in a manner that converges on some condition.(base case)	An infinite loop occurs with iteration if the loop-condition test never becomes false
4	Recursion terminates when a base case is recognized	Iteration terminates when the loop-condition fails
5	Recursion is usually slower then iteration due to overhead of maintaining stack(activation-record)	Iteration does not use stack so it's faster than recursion
6	Recursion uses more memory than iteration	Iteration consume less memory
7	Infinite recursion can crash the system	infinite looping uses CPU cycles repeatedly
8	Recursion makes code smaller	Iteration makes code longer

PASSING ARGUMENTS BY VALUE

```
//Use of pass by value in swap function
#include<stdio.h>
//Function declaration
void swap(int,int);
//Function definitions
void main()
{
    int a=10,b=20;
    printf("Before swap values are %d %d\n",a,b);
    swap(a,b);
    printf("After swap values are %d %d\n",a,b);
}
void swap(int x, int y)
{
    x=x+y;
    y=x-y;
    x=x-y;
    printf("In swap function values are %d %d\n",x,y);
```



Output window:

```
Before swap values are 10 20
In swap function values are 20 10
After swap values are 10 20
```

Remarks:

- **On the execution of function call i.e. swap(a,b);, the values of actual arguments a and b are copied into the formal arguments x and y.**
- Formal arguments are allocated at separate memory locations.
- **A change made in the formal arguments is independent of the actual arguments.**
- **On returning back from the called function, the formal arguments are destroyed and the access to the actual arguments gives values unchanged.**
- **THUS, IF THE ARGUMENTS ARE PASSED BY VALUE, THE CHANGES MADE IN THE VALUES OF FORMAL PARAMETERS INSIDE THE CALLED FUNCTION ARE NOT REFLECTED BACK TO THE CALLING FUNCTION.**

ANALOGY

The main function i.e. the master function wants to get some changes done in a file from its sub-ordinate worker i.e. the swap function. The main function got the file (i.e. actual arguments) Xeroxed and has handed over the Xeroxed copy of the file (i.e. formal arguments) to the swap function for changes. The swap function has made changes in the Xeroxed copy and has returned the file back to the main function. On getting the control back, the main function is still referring to the original file and finds that no changes have been made in it. The changes have been made in the Xeroxed copy, so how can main function find changes in the original file.