CHAPTER 1

USER DEFINED FUNCTIONS

1. INTRODUCTION

Every C program should consist of one or more functions. Among these functions *main( )* function is compulsory. All programs start execution from *main( )* function.

Functions: Functions are independent program modules that are designed to carry out a particular task.

There are two types:

1. Built-in (Library) functions
2. User defined functions

1. ***Built-in functions:*** These are C language functions already available with C compliers and can be used by any programmers.

   Ex: *printf( ),scanf( )*

2. ***User-defined functions:*** These are written by programmers for their own purpose and are not readily available.

Advantages of using Functions:

Functions based modular programming is advantageous in many ways:

1. Managing huge programs and software packages is easier by dividing them into functions/modules—Maintenance is easier
2. Error detection is easier—Debugging is easier
3. Functions once written can be re-used in any other applications – Reusability is enhanced
4. We can protect our data from illegal users—Data protection becomes easier

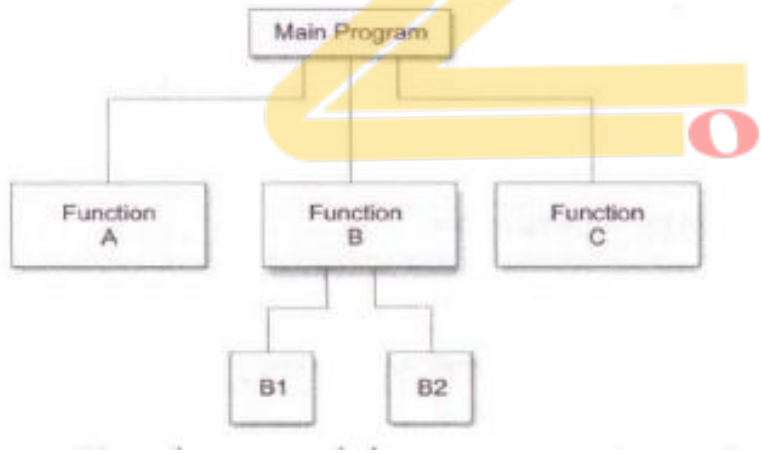
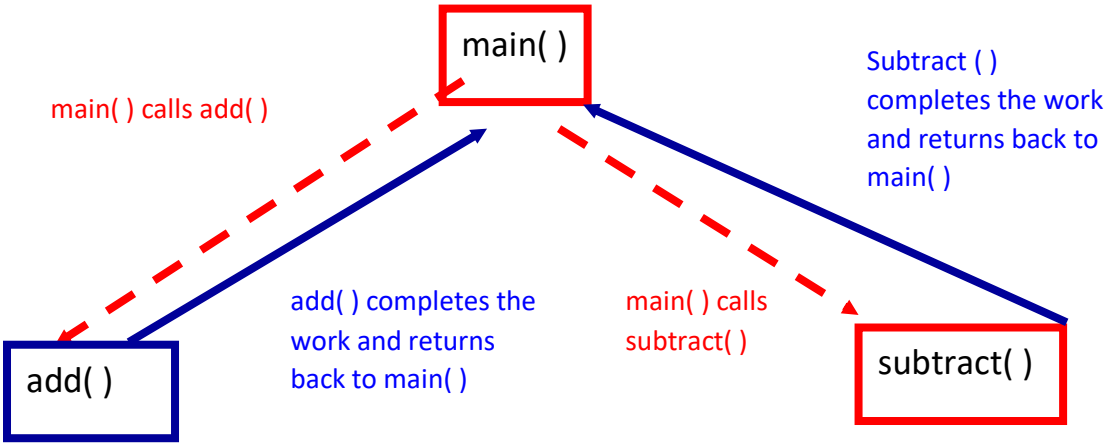


Figure 1: Top-down modular programming using functions

The below block diagram represents how functions are useful: Instead of writing entire program within *main( )* function we can write independent modules as shown:



A function can accept any number of inputs but, returns only one value as output as shown below:

2. MODULAR PROGRAMMING

- ✓ Modular programming is defined as organizing a large program into small, independent program segments called modules that are separately named and individually callable program units.
- ✓ It is basically a "divide-and-conquer" approach to problem solving.
- ✓ The characteristics of modular programming are:

1. Each module should do only one thing.
2. Communication between modules is allowed only by calling module.
3. No communication can take place directly between modules that do not have calling-called relationship.
4. All modules are designed as single-entry, single-exit systems using control structures.
5. A module can be called by one and only higher module.

3. ELEMENTS OF USER-DEFINED FUNCTIONS

There are three elements that are related to functions:

i. Function definition
ii. Function call
iii. Function declaration/Function prototype

i. **Function definition:** It is an independent program module that is specially written to implement the requirements of the function.

ii. **Function call:** The function should be invoked at a required place in the program which is called as Function call.

- ✓ The program/function that calls the function is referred as calling program or calling function.
- ✓ The program/function that is called by program/function is referred as called program or called function.

iii. **Function declaration:** The calling program should declare any function that is to be used later in the program which is called as function declaration.

FUNCTION DEFINITION/ DEFINITION OF FUNCTIONS

A function definition, also known as function implementation will include:

i. Function type/ Return type
ii. Function name
iii. List of parameters/ Arguments
iv. Local variable declaration
v. Function statements/ Executable statements
vi. A return statement

All the above six elements are grouped into two parts. Namely:

- Function header (First three elements)
- Function body (Second three elements)

Syntax:

```
function_type  function_name(list of parameters)
{
    local variable declaration;
    executable statement1;
    executable statement2;
    . . . . . .
    . . . . . .
    return statement;
}
```

Note: The first line ***function_type function_name(parameter list)*** is known as the function header and the statements within opening and closing braces is known as the function body.

i. Function Header: It consists of three parts: Function type, function name and list of paraemeters. The semicolon is not present at the end of header.

Function type:

- ✓ The function type specifies the type of value that the function is expected to return to the calling function.
- ✓ If the return type or function type is not specified, then it is assumed as int.
- ✓ If function is not returning anything, then it is void.

Function name: The function name is any valid C identifier and must follow same rules as of other variable.

List of Parameters: The parameter list declares the variables that will receive the data sent by the calling program which serves as input and known as Formal Parameter List.

ii. Function Body: The function body contains the declarations and statements necessary for performing the required task. The body is enclosed in braces, contains three parts:

1. *Local declarations* that specify the variables needed by the function.
2. *Function statements* that perform the task of the function.
3. *A return statement* that returns the value evaluated by the function.

FUNCTION DECLARATION

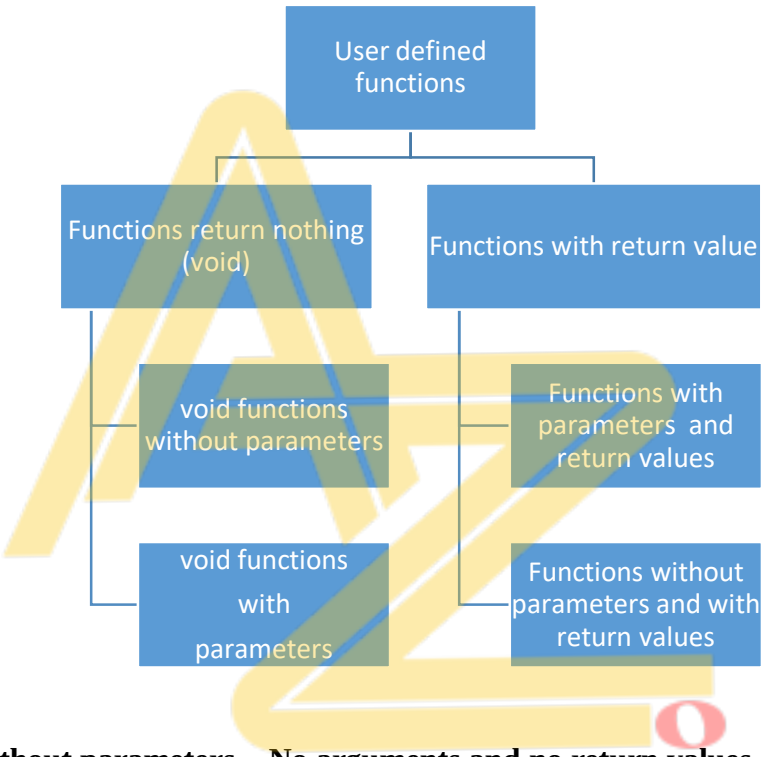
A function declaration consists of four parts:

- Function type/ return type
- Function name
- Parameter List
- Terminating semicolon

Syntax: *function_type function_name(list of parameters);*

Ex: int sum(int, int);
int isprime(int);

4. CATEGORIES OF FUNCTIONS



1. Void Functions without parameters – 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 does not return a value, the calling function does not receive any data from the called function.

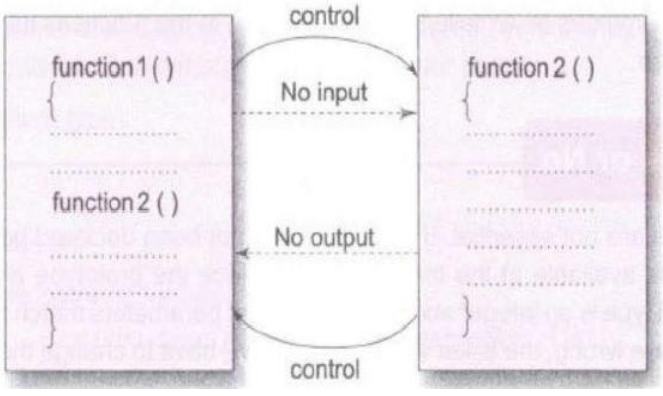


Figure: No data communication between functions

Example:

```
#include<stdio.h>
void add( );
main( )
{
        add( );
}
void add( )
{
        int a=5, b=5, sum=0;
        sum = a+b;
        printf("Result = %d", sum);
        return;
}
```

2. Void Functions with parameters – Arguments but no return values

- ✓ The calling function accepts the input, checks its validity and pass it on to the called function.
- ✓ The called function does not return any values back to calling function.
- ✓ The actual and formal arguments should match in number, type and order.

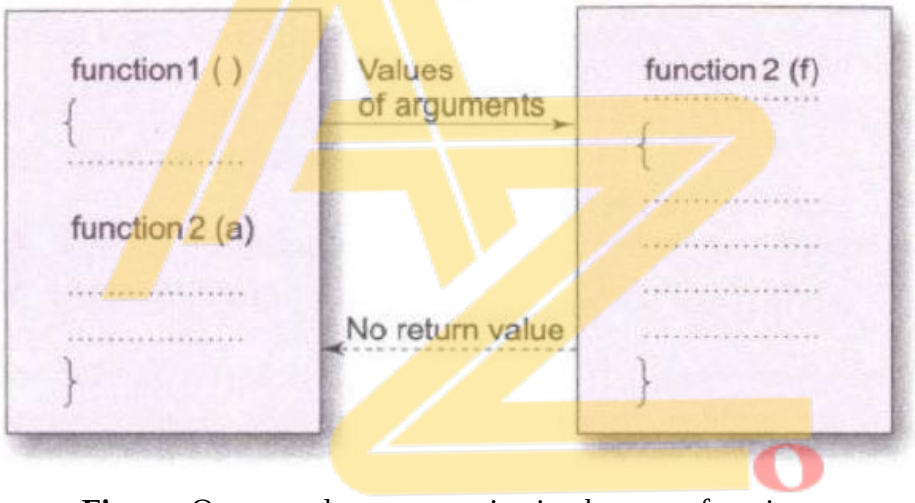


Figure: One-way data communication between functions

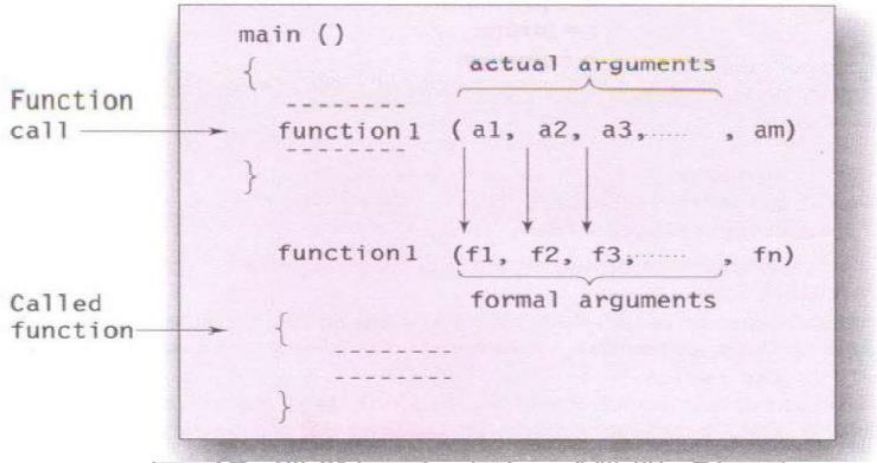


Figure: Arguments matching between the function call and the called function

Example:

```
#include<stdio.h>
void add(int, int);
main( )
{
        int a=5, b=5;
        add(a, b);
}
void add(int a, int b )
{
        int sum=0;
        sum = a+b;
        printf("Result = %d", sum);
        return;
}
```

3. Functions with parameters and return values – Arguments with return values

✓ The calling function sends the data to called function and also accepts the return values from called function.

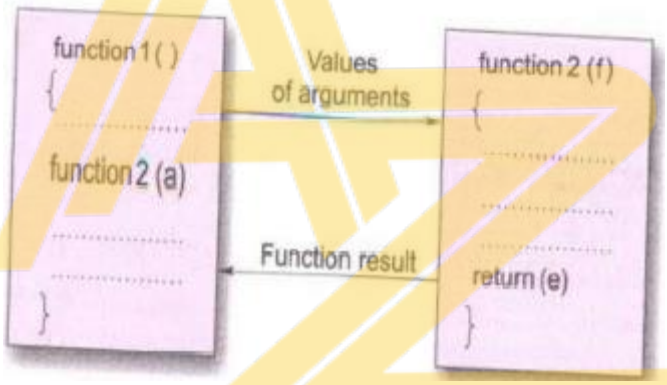


Figure: Two-way data communication between functions

Example:

```
#include<stdio.h>
int add(int, int);
main( )
{
        int a=5, b=5,sum=0 ;
        sum = add(a, b);
        printf("Result = %d", sum);
}
int add(int a, int b)
{
        int x;
        x = a+b;
        return x;
}
```

4. Functions without parameters and with return values – No arguments but return values

- ✓ The calling function does not send any data to called function and but, accepts the return values from called function.

Example:

```
#include<stdio.h>
int add( );
main( )
{
        int sum=0 ;
        sum = add( );
        printf("Result = %d", sum);
}
int add( )
{
        int a=5, b=5, x=0;
        x = a+b;
        return x;
}
```

- ✓ **Actual Parameters:** When a function is called, the values that are passed in the call are called actual parameters.
- ✓ **Formal Parameters:** The values which are received by the function, and are assigned to formal parameters.
- ✓ **Global Variables:** These are declared outside any function, and they can be accessed on any function in the program.
- ✓ **Local Variables:** These are declared inside a function, and can be used only inside that function.

5. INTER-FUNCTION COMMUNICATION

Two functions can communicate with each other by two methods:

1. Pass by value (By Passing parameters as values)
2. Pass by address (By passing parameters as address)

1. Pass by value:

In pass-by-value the calling function sends the copy of parameters (Actual Parameters) to the called function (Formal Parameters). The changes does not affect the actual value.

Example:

```
#include<stdio.h>
int swap(int, int);
int main( )
{
        int a=5, b=10 ;
        swap(a, b);
        printf("%d%d:, a, b);
```

```
}
int swap(int x, int y)
{
        int temp;;
        temp = x;
        x = y;
        y = temp;
}
```

In the above example even though x and y values get swapped, the a and b values remains unchanged. So swapping of two values in this method fails.

2. By Passing parameters as address (Pass by address)

In pass by reference the calling function sends the address of parameters (Actual Parameters) to the called function (Formal Parameters). The changes affects the actual value.

Example:

```
#include<stdio.h>
int swap(int *, int *);
int main( )
{
        int a=5, b=10 ;
        swap(&a, &b);
        printf("%d%d:, a, b);
}
int swap(int *x, int *y)
{
        int temp;;
        temp = *x;
        *x = *y;
        *y = temp;
}
```

In this method the values a and b have been changed or swapped.

6. NESTING OF FUNCTIONS

A function within a function is called as nesting of functions.

Example:

```
#include<stdio.h>
float checkno(float );
float div(float, float);
main( )
{
        float a, b, res;
        printf("Enter two numbers");
        scanf("%f%f",&a,&b);
        div(a,b);
```

```
}
float div(float a, float b)
{
        if(checkno(b))
                printf("Result = %f", a/b);
        else
                printf("Division not possible");
}
float checkno(float b)
{
        if(b!=0)
                return 1;
        else
                return 0;
}
```

7. PASSING ARRAYS TO FUNCTIONS

Two ways of passing arrays to functions are:

1. Pass individual elements of array as parameter
2. Pass complete array as parameter

Pass individual elements of array as parameter	**Pass complete array as parameter**
Here, each individual element of array is passed to function separately.	Here, the complete array is passed to the function.

Example:
 	
#include<stdio.h>
int square(int);
int main( )
{
  int num[5], i;
  num[5] ={1, 2, 3, 4, 5};
  for(i=0; i<5; i++)
  {
  square(num[i]);
  }
}
int square(int n)
{
 int sq;
 sq= n * n;
 printf("%d ", sq);
}
	Example:
	
	#include<stdio.h>
	int sum(int [ ]);
	int main( )
	{
	  int marks[5], i;
	  marks[5] ={10, 20, 30, 40, 50};
	  sum(marks);
	}
	int sum(int n[ ])
	{
	  int i, sum=0;
	  for(i=0; i<5; i++)
	  {
	  sum = sum+n[i];
	  }
	  printf("Sum = %d ", sum);
	}

8. PASSING STRINGS TO FUNCTIONS

The strings are treated as character arrays in C and therefore the rules for passing strings to functions are very similar to those for passing arrays to functions.

Basic rules are:

1. The string to be passed must be declared as a formal argument of the function when it is defined.
   Ex: void display(char item_name[])
   {
   . . . . . .
   . . . . . .
   }
2. The function prototype must show that the argument is a string.
   Ex: void display(char str[])
3. A call to the function must have a string array name without subscripts as its actual argument.
   Ex: display(names);
   Where names is a properly declared string in the calling function.

CHAPTER 2

RECURSION

Programmers use two approaches to write repetitive algorithms:

- One approach is to use loops
- The other uses recursion

✓ A function calling itself repeatedly is called Recursion.
✓ All recursive functions have two elements each call either solves one part of the problem or it reduces the size of the problem.
✓ The statement that solves the problem is known as base case.
✓ The rest of the function is known as general case.

Ex: Recursive definition of factorial is

$$\text{Fact}(n) = \begin{cases} 1 & \text{if } n=0 \quad \text{base case} \\ n*\text{fact}(n-1) & \text{if } n>0 \quad \text{general case} \end{cases}$$

Limitations of Recursion

✓ Recursive solutions may involve extensive overhead because they use function calls.
✓ Each time you make a call, you use up some of your memory allocation. If recursion is deep, then you may run out of memory.

PROGRAMS

1. WACP to find a factorial of number using recursion.

```
#include<stdio.h>
int fact(int);
void main( )
{
        int n,res;
        printf("Enter a number");
        scanf("%d",&n);
        res=fact(n);
        printf("Factorial = %d",res);
}
int fact(int n)
{
        if(n==0)
           return 1;
        else
          return (n*fact(n-1));
}
```

2. Program to generate Fibonacci series using recursion

```
#include<stdio.h>
 int Fibonacci(int);
void main()
{
  int n, i = 0, c;
  printf(" Enter the number of terms");
  scanf("%d",&n);
  printf("Fibonacci series\n");
  for ( c = 1 ; c <= n ; c++ )
  {
    printf("%d\n", Fibonacci(i));
    i++;
  }
 }
int Fibonacci(int n)
{
  if ( n == 0 )
    return 0;
  else if ( n == 1 )
    return 1;
  else
    return (Fibonacci(n-1) + Fibonacci(n-2));
}
```

VTU SOLVED QUESTIONS

1 **WAP to find GCD and LCM of two numbers using concept of functions.**

```
#include<stdio.h>
int hcf(int, int);
int main( )
{
        int x, y, gcd, lcm;
        printf("Enter two numbers");
        scanf("%d%d", &x, &y);
        gcd = hcf(x, y);
        lcm = (x * y)/ gcd;
        printf("GCD = %d", gcd);
        printf("LCM = %d", lcm);
}
int hcf( int x, int y)
{
        if(x == 0)
                return y;
        while(y!=0)
        {
                if(x > y)
                        x = x-y;
                else
                        y = y-x;
        }
        return x;
}
```

2 **List the storage class specifiers. Explain any one of them.**

OR

Give the scope and lifetime of following:

- **i. External variable**
- **ii. Static variable**
- **iii. Automatic variable**
- **iv. Register variable**

Variable	Definition	Scope	Lifetime
External Variable/ Global Variable	The variables declared outside of all functions.	Global	Runtime of program
Static Variable	The variables declared using static keyword.	Local	Runtime of program
Automatic Variable/ Local Variable	The variables declared inside the function.	Local	Remains within the function in which it is declared.
Register Variable	The variables declared using register keyword.	Local	Remains within the block in which it is declared.

3 **WACP for evaluating the binomial coefficient using a function factorial(n).**

OR

WACP to find the binomial coefficient of a number using recursion.

```
#include<stdio.h>
int factorial(int);
int main( )
{
        int n, r, binom;
        printf("Enter value of n and r");
        scanf("%d%d", &n, &r);
        if(n<0 || r<0 || n<r)
                printf("Invalid Input");
        else
        {
                binom = factorial(n)/ factorial® * factorial(n-r);
                printf("Result = %d", binom);
        }
}
int factorial(int n)
{
        if(n == 0)
                return 1;
        else
                return(n*factorial(n-1));
}
```

4 **Write a C recursive function for multiplying two integers where a function call is passed with two integers m and n.**

```
#include<stdio.h>
int product(int, int);
int main( )
{
        int m, n, res;
        printf("Enter two numbers");
        scanf("%d%d", &m, &n);
        res = product(m, n);
        printf("Result = %d", res);
}
int product(int m, int n)
{
        if(m<n)
                return product(n, m);
        else if(n!=0)
                return (m + product(m, n-1));
        else
                return 0;
}
```

5 **Differentiate:**

i) User defined and Built-in function

ii) ii) Recursion and Iteration

User defined function	Built in function
These are requirement based functions.	These are pre-defined functions.
It is given by users.	It is given by developers.
The names of the functions can be changed.	The names of the functions cannot be changed.
These are part of program called during compile time.	These are part of header file called during run time.
Ex: isprime( ), fibonacci( )	Ex: printf( ), scanf( ), cos( ), sin( )

Iteration	Recursion
Allows set of instructions to be repeatedly executed.	The statement in a body of function calls itself.
Initialization, Condition and Updation are present.	Only termination condition is specified.
Applicable to iterative statements.	Applicable to functions.
The code size is bigger.	The code size is reduced.
No overhead of repeated function calls.	Processes overhead of repeated function calls.

6 **WACP to find the largest element in an array.**

```
#include<stdio.h>
void main( )
{
        int n, a[100], i, large=0;
        printf("Enter the size of array");
        scanf("%d", &n);
        printf("Enter the elements of array");
        for(i=0; i<n; i++)
        scanf("%d", &a[i]);
        for(i=0; i<n; i++)
        {
                if(a[i] > large)
                        large = a[i];
        }
        Printf("Largest number = %d", large);
}
```

7 **WACP using functions to swap two numbers using global variable concept and call by reference concept.**

```
#include<stdio.h>
int a, b;
void swap(int *, int *);
int main( )
{
        printf("Enter two numbers");
        scanf("%d%d", &a, %b);
        printf("Before Swapping a = %d, b= %d", a, b);
        swap(&a, &b);
        printf("After Swapping a = %d, b= %d", a, b);
}
```

```
void swap(int *p, int *q)
{
        int temp;
        temp = *p;
        *p = *q;
        *q = temp;
}
```