USER DEFINE FUNCTION

- The functions which are created by the user are known as user define function.
- A single c program can have multiple user define function

PARTS OF USER DEFINE FUNCTION

1. Function declaration: A function must be declared globally to tell the compiler about the function name and return type.

- SYNTAX:

```
Returntypefunction_name(argument_list);
```

2. Function calling: all user define functions should be called in the main function.

- SYNTAX:

```
Function_name(argument_list);
```

3. Function definition: this part represents the actual statements and the body of user define function.

- SYNTAX:

```
Returntypefunction_name(argument_list)
   {
       //body of the user define function;

    }
```

- RETURN TYPE:
  - If a function doesn't want to return any value then void should be used.
  - Void main()
    {
    Printf("hello world"); }

- If a function wants to return any value then any datatypecan be used.
- Int main()
  {
  Return 0;
  }
- Float main()
  {
  Return 10.1;
  }

- ARGUMENT: Argument is a value which is passed to the function.

TYPES OF USER DEFINE FUNCTION

1. WITHOUT ARGUMENT WITHOUT RETURNTYPE

```
#include<stdio.h>
void sum();       //decaration
void main()
{
   sum();    //calling
getch();
}
void sum()  //defination
{
   int a=1,b=1;
   printf("The sum is %d",a+b);
}
```

2. WITHOUT ARGUMENT WITH RETURNTYPE

```
#include<stdio.h>
int sum();    //decaration
void main()
{
   int result;
   result = sum();
   printf("%d",result);
getch();
}
int sum()
{
   int a=1,b=1;
   return a+b;
}
```

3. WITH ARGUMENT WITHOUT RETURNTYPE

```
#include<stdio.h>
void sum(int, int);     //
void main()
{
   int a=1,b=1;
   sum(a,b);
getch();
}
void sum(int a, int b)
{
   printf("\nThe sum is %d",a+b);
}
```

4. WITH ARGUMENT WITH RETURNTYPE

```
#include<stdio.h>
int sum(int, int);  //decaration
void main()
{
   int a=1,b=1,result;
   result = sum(a,b);
   printf("\nThe sum is : %d",result);
getch();
}
int sum(int a, int b)
{
   return a+b;
}
```

CALL BY VALUE

- In call by value method the actual parameter is copied into formal parameter.
- The value of variable is used in function calling.
- ACTUAL PARAMETER is the parameter used in function calling.
- FORMALPARAMETER is the parameter used in function definition.
- EXAMPLE:

```
#include<stdio.h>
#include<conio.h>
void demo(int);
void main()
{
        int x=100;
        printf("before function call value of x is:%d",x);
        demo(x);
        printf("after function call value of x is:%d",x);
        getch();
}
void demo(int num)
{
        printf("value of num before addition:%d",num);
        num=num+100;
        printf("value of num after addition:%d",num);
}
```

o/p:

```
before function call value of x is:100
value of num before addition:100
value of num after addition:200
after function call value of x is:100
```

- EXAMPLE 2:

```
#include <stdio.h>
void swap(int , int);
int main()
{
   int a = 10;
   int b = 20;
   printf("Before swapping the values in main a = %d, b = %d\n",a,b);
   swap(a,b);
   printf("After swapping values in main a = %d, b = %d\n",a,b);
}
void swap (int a, int b)
{
   int temp;
   temp = a;
   a=b;
   b=temp;
   printf("After swapping values in function a = %d, b = %d\n",a,b);
}
```

O/P:

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

CALL BY REFERENCE

- In call by reference, the address of the variable is passed into the function call as the actual parameter.
- The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
- In call by reference, the memory allocation is similar for both formal parameters and actual parameters.
- EXAMPLE:

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

Output

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

- EXAMPLE 2:

```
#include <stdio.h>
void swap(int *, int *); //prototype of the function
int main()
{
   int a = 10;
   int b = 20;
   printf("Before swapping the values in main a = %d, b = %d\n",a,b);
   swap(&a,&b);
   printf("After swapping values in main a = %d, b = %d\n",a,b);
= 20
}
void swap (int *a, int *b)
{
   int temp;
   temp = *a;
   *a=*b;
   *b=temp;
   printf("After swapping values in function a = %d, b = %d\n",*a,*b);
}
```

O/P:

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

RECURSIVE FUNCTION

- Any function which calls itself is called recursive function
- Recursion is the process which comes into existence when a function calls a copy of itself

NORMAL FUNCTION	RECURSIVE FUNCTION

void main( )
{
add();
}
void add()
{
.....
}
	void main( )
	{
	add();
	}
	void add()
	{
	add();
	}

- EXAMPLE 1: FACTORIAL NUMBER

```
#include<stdio.h>
int fact(int i)
{
   if(i<=1)
   {
      return 1;
   }
   return i*fact(i-1);
}
void main()
{
   int i;
   printf("enter i");
   scanf("%d",&i);
   printf("factorial is:%d",fact(i));
}
```

O/P:
Enter i 5
Factorial is 120

- EXAMPLE 2: FIBOACCI SERIES

```
#include<stdio.h>
void main()
 {
   int n,i;
   printf("enter n");
   scanf("%d",&n);
   for(i = 0;i<n;i++)
   {
      printf("%d ",fibbonacci(i));
   }
}
int fibbonacci(int n)
{
   if(n == 0)
   {
      return 0;
   }
   else if(n == 1)
   {
      return 1;
   }
   else
   {
      return (fibbonacci(n-1) + fibbonacci(n-2));
   }
}
```

O/P:
Enter n: 5
0 1 1 2 3