Functions

Definition- A function is a self-contained block of statements that perform a coherent task of some kind. Every C program can be thought of as a collection of these functions.

Example- Consider the following program-

```
void main()
{
   example( ) ;       //function call statement
   printf ( "\nHope you have got it!" ) ;
}

 void  example( )    //function definition
  {
     printf ( "\nThis is how a function works." ) ;
  }
```

```
Output: This is how a function works.
        Hope you have got it!
```

Note: Here, **main( )** itself is a function and through it we are calling the function **example( )**. When we say that **main( )** ‘calls’ the function **example( ), w**e mean that the control passes to the function **example( ).** The activity of **main( )** is temporarily suspended; it falls asleep while the **example( )** function wakes up and goes to work. When the **example( )** function runs out of statements to execute, the control returns to **main( )**, which comes to life again and begins executing its code at the exact point where it left off. Thus, **main( )** becomes the ‘calling’ function, whereas **example( )** becomes the ‘called’ function.

Remember:

1. Any C program contains at least one function.
2. If a program contains only one function, it must be **main( )**.
3. If a C program contains more than one function, then one (and only one) of these functions must be **main( )**, because program execution always begins with **main( )**.
4. There is no limit on the number of functions that might be present in a C program.
5. Each function in a program is called in the sequence specified by the function calls in **main( )**.
6. After each function has done its thing, control returns to **main( )**.When **main( )** runs out of function calls, the program ends.
7. A function can be called from other function, but a function cannot be defined in another function.

Types of functions: C has two types of functions – 1) Library functions.
2) User defined functions.

1) Library functions – library functions are nothing but commonly required functions grouped together and stored in a Library. This library of functions is present on the disk and is written by people who write compilers.

Ex.-**printf( )**, **scanf( )** etc.

2) User-defined functions- These functions are defined by a user in order to perform a particular task. These functions do not exist in a library.

Ex.-**example()** etc.

Advantage of using function:

1. Writing functions avoids rewriting the same code over and over.
2. Using functions it becomes easier to design, write and debug a program (as the function divides the operation of a program into separate activities which are written and checked independently).

Passing Values between Functions: We can pass values between the calling and called function. The mechanism used to convey information to the function is the 'argument' or 'parameter' (You have unknowingly used the arguments in the **printf( )** and **scanf( )** functions; the format string and the list of variables used inside the parentheses in these functions are arguments).

- The arguments which are passed in the calling function are called 'actual arguments'.
- The arguments which appear at the function definition are called 'formal arguments'.

Consider the following example-

```
void  main()
{
  int a,b,c, sum;
  printf("Enter three numbers");
  scanf("%d%d%d",&a,&b,&c);
  sum=calculate(a,b,c);     //a,b,c are actual arguments.
  printf("\nSum=%d",sum);
}

calculate(int x, int y, int z)          //x,y,z are formal arguments.
{
  int total;
   total=x+y+z;
   return (total);
}
```

Remember:

- The **return** statement serves two purposes:
  1. On executing the **return** statement it immediately transfers the control back to the calling program.
  2. It returns the value present in the parentheses after **return**, to the calling program.
- If we want that a called function should not return any value, in that case, we must mention so by using the keyword **void**.
- A function can return only one value at a time. Thus the following statements are invalid-

```
return(a,b);
return(x,y,12);
```

- There is no restriction on the number of **return** statements that may be present in a function. Also, the **return** statement need not always be present at the end of the called function.

Calling Convention: Calling convention indicates the order in which arguments are passed to a function when a function call is encountered. There are two possibilities here: i) Arguments might be passed from left to right.

ii) Arguments might be passed from right to left.

C language follows the second order. Consider the following example-

```
int a = 1 ;
printf ( "%d %d %d", a, ++a, a++ ) ;
```

It appears that this **printf( )** would output 1 2 2. This however is not the case. Surprisingly, it outputs 3 3 1. This is because C's calling convention is from right to left. That is, firstly 1 is passed through the expression **a++** and then **a** is incremented to 2. Then result of **++a** is passed. That is, **a** is incremented to 3 and then passed. Finally, latest value of **a**, i.e. 3, is passed. Thus in right to left order 1, 3, 3 get passed. Once **printf( )** collects them it prints them in the order in which we have asked it to get them printed (and not the order in which they were passed). Thus 3 3 1 gets printed.

Function Prototypes- Whenever we call the library functions we must write their prototype before making the call. This helps the compiler in checking whether the values being passed and returned are as per the prototype declaration. But since we don't define the library functions (we merely call them) we may not know the prototypes of library functions. Hence when the library of functions is provided a set of '.h' files is also provided. These files contain the prototypes of library functions. But for the user defined functions we need to write the prototype manually within the program.

Note: Any C function by default returns an **int** value. More specifically, whenever a call is made to a function, the compiler assumes that this function would return a value of the type **int**. If we desire that a function should return a value other than an **int**, then it is necessary to explicitly mention so in the calling function as well as in the called function.

For example consider the following program –

```
/*Program to find the square of any number*/
    void main( )
    {
      float square ( float ) ; // function prototype declaration
      float a, b ;
      printf ( "\nEnter any number " ) ;
      scanf ( "%f", &a ) ;
      b = square ( a ) ;
      printf ( "\nSquare of %f is %f", a, b ) ;
     }
    float square ( float x )
      {
        float y ;
        y = x * x ;
        return ( y ) ;

      }
```

Note: The statement '**float square (float) '** is called the prototype declaration of the **square( )** function. What it means is **square( )** is a function that receives a **float** and returns a **float**. We have done the prototype declaration in **main( )** because we have called it from **main( )**. There is a possibility that we may call **square( )** from several other functions other than **main( )**. Does this mean that we would need prototype declaration of **square( )** in all these functions. No, in such a case we would make only one declaration outside all the functions at the beginning of the program.

Call by Value and Call by Reference: We can call a function by value or reference.

- When we call a function and pass the 'values' of variables to it, this is known as 'call by value' method.
- When we call a function and pass the 'address' of variables to it, this is known as 'call by reference'.

In the **call by value** method the 'value' of each of the 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 of actual arguments in the calling function.

In the **call by reference** method the 'addresses' of actual arguments in the calling function are copied into formal arguments of the called function. This means that using these addresses we would have an access to the actual arguments and hence we would be able to manipulate them.

Example-Call by value	Example-Call by reference
void main( )	void main( )
{	{
int a = 10, b = 20 ;	int a = 10, b = 20 ;
swapv (a, b) ;	swapr (&a, &b) ;
printf ("\na = %d b = %d",a, b) ;	printf ("\na = %d b = %d", a, b) ;
}	}
swapv (int x, int y)	swapr(int *x, int *y)
{	{
int t ;	int t ;
t = x ;	t = *x ;
x = y ;	*x = *y ;
y = t ;	*y = t ;
printf ("\nx = %d y = %d", x, y);	}
}	
Output:	Output:
x = 20 y = 10	a = 20 b = 10
a = 10 b = 20	

Note: 1. Using a call by reference intelligently we can make a function return more than one value at a time, which is not possible ordinarily. This is shown in the program given below-

```
void main( )
 {
  int radius ;
  float area, perimeter ;
  printf ( "\nEnter radius of a circle " ) ;
  scanf ( "%d", &radius ) ;
  areaperi ( radius, &area, &perimeter ) ;
  printf ( "Area = %f", area ) ;
  printf ( "\nPerimeter = %f", perimeter ) ;
 }

 areaperi ( int r, float *a, float *p )
 {
   *a = 3.14 * r * r ;
   *p = 2 * 3.14 * r ;
  }
```

Here, we are making a mixed call, in the sense, we are passing the value of **radius** but, addresses of **area** and **perimeter**. And since we are passing the addresses, any change that we make in values stored at addresses contained in the variables **a** and **p**, would make the change effective in **main( )**. Thus, we have been able to indirectly return two values from a called function, and hence, have overcome the limitation of the **return** statement, which can return only one value from a function at a time.

2. If you want that the value of an actual argument should not get changed in the function being called, pass the actual argument by value.

3. If you want that the value of an actual argument should get changed in the function being called, pass the actual argument by reference.

Recursion

A function is called 'recursive' if a statement within the body of a function calls the same function. Sometimes called 'circular definition', recursion is thus the process of defining something in terms of itself.

Example- Consider the following program which finds the factorial of a given number with recursive & non-recursive function-

Non-recursive function	Recursive function
main( )	main( )
{	{
int a, fact ;	int a, fact ;
printf ("\nEnter any number ") ;	printf ("\nEnter any number ") ;
scanf ("%d", &a) ;	scanf ("%d", &a) ;
fact = factorial (a) ;	fact = rec(a) ;
printf ("Factorial value = %d", fact) ;	printf ("Factorial value = %d", fact) ;
}	}
	rec(int x)
factorial (int x)	{
{	int f ;
int f = 1, i ;	if (x == 1)
for (i = x ; i >= 1 ; i--)	return (1) ;
f = f * i ;	else
return (f) ;	f = x * rec(x - 1) ;
}	return (f) ;
	}

Let us understand this recursive factorial function thoroughly. In the first run when the number entered through **scanf( )** is 1, let us see what action does **rec( )** take. The value of **a** (i.e. 1) is copied into **x**. Since **x** turns out to be 1 the condition **if (x == 1)** is satisfied and hence 1 (which indeed is the value of 1 factorial) is returned through the **return** statement. When the number entered through **scanf( )** is 2, the **(x == 1)** test fails, so we reach the statement,

f = x * rec (x - 1) ;

And here is where we meet recursion. How do we handle the expression **x * rec (x - 1)**? We multiply **x** by **rec (x - 1)**. Since the current value of **x** is 2, it is same as saying that we must calculate the value (2 * rec (1)). We know that the value returned by **rec (1)** is 1, so the expression reduces to (2 * 1), or simply 2. Thus the statement,

x * rec (x - 1) ;

evaluates to 2, which is stored in the variable **f**, and is returned to **main( )**, where it is duly printed as

Factorial value = 2
Now perhaps you can see what would happen if the value of **a** is 3, 4, 5 and so on.