FUNCTIONS IN ‘C’

Dear friends, let us go through an important and interesting concept of functions in C programming language which is essential to make use of modular programming technique.

What is modular programming? In modular programming, the problem is subdivided into a number of sub-problems or modules. It is clear that the effective problem solving must do problem decomposition. Breaking a problem it into small, manageable pieces is critical in writing large programs. In C language a self-contained block of code that performs a particular task is called a function. C supports the use of various library functions, which are used to carry out a number of commonly used operations or calculations. C also allows programmers to define their own functions for carrying out various individual tasks. In this session we will cover the creation and utilization of such user defined functions.

After going through this session you will be able to

- Explain what is a function
- Describe access to function
- Define parameters, data type specification
- Explain function prototype and recursion
- Define storage classes – automatic, external, static variables

Now let us briefly discuss about the modular approach in programming.

MODULAR PROGRAMMING APPROACH

The use of user-defined functions allows a large program to be broken down into a number of smaller, self-contained components, each of which has some unique, identifiable purpose. Thus a C program can be modularized through the intelligent use of such functions. There are several advantages to this modular approach to program development. For example many programs require a particular group of instructions to be accessed repeatedly from several different places within a program. The repeated instruction can be placed within a single function, which can then be accessed whenever it is needed. Moreover, a different set of data can be transferred to the function each time it is accessed. Thus, the use of a function avoids the redundancy in instructions. The decomposition of a program into individual program modules is generally considered to be an important part of good programming.

TYPES OF FUNCTIONS

A C program is made up of one or more functions. Based on the nature of creation functions can be classified in to **built-in functions** or **user-defined functions**. Built-in functions are predefined and supplied with the compiler. These are also called library functions and all these functions are available in C library. I hope, some of the built-in functions like the simple I/O functions for example getchar(), putchar(), scanf(), printf() and mathematical functions like sqrt(), sin(), cos() etc. have been discussed and made use of in the previous sessions.

The functions defined by the users are called as **user-defined functions.** So far you might have used only one user defined function called main(). Let us now consider an example

```
#include <math.h>
main()
{
float x,y;
scanf("%f", &x);
y=sqrt(x);
printf("Square root of %f is %f\n",x,y);
}
```

The **main()** function invokes (or calls) other functions within it. Here, three functions **scanf(), sqrt()** and **printf()** are invoked by the **main()** function. The function **main()** is the calling function and the other functions **scanf(), sqrt()** and **printf()** are called functions. A function that invokes another function is known as **calling function**. A function, which is invoked by another function is known as a **called function**. In C **main()** is the first calling function in any program. It is a special function which tells the compiler to start the execution of a C program from the beginning of the function **main().** It is not possible to have more than one **main()** function because the compiler will not know where to start execution in such a situation.

An identifier (other than keywords) followed by an open parentheses is recognized as a function name by the compiler. The items within the parentheses are called as **parameters** or **arguments** through which information is passed to the function. A function may be used to calculate and return that value to the calling function. The information is returned from functions via return statement.

Once the function has been executed, control will be returned to the point from which the function was accessed. It is not necessary that every function must return information; there are some functions which do not return any information. For example the system defined function **printf().**

To make use of the user defined functions you must be able to

- Define a function
- Declare the prototype of a function and
- Invoke the function

FUNCTION DEFINITION

Function is a self-contained program segment that carries out some specific well-defined task. The definitions of functions may appear in any order in a program file because they are independent of one another. A function can be executed from anywhere within a program.

Before using any function it must be defined in the program. A function definition describes what a function does, how its actions are achieved and how it is used. It consists of a function header and a function body. The general format for defining a function is

```
return_type function_name (parameter list) /*Function Header */
{
declarations()
statements()                          /* Function body */
return(expression);
}
```

The first line of a function definition contains the data type of the information return by the function, followed by function name, and a set of arguments or parameters, separated by commas and enclosed in parentheses. As it heads the function it is known as **Function Header**. The set of arguments may be skipped over. The data type can be omitted if the function returns an integer or a character. An empty pair of parentheses must follow the function name if the function definition does not include any argument or parameters.

The general form of the function header is

data-type function-name (formal argument 1, formal argument 2…formal argument n)

For example: int swap(int a, int b)

Some points to be kept in mind while specifying **return_type**

- Specifies the data type of the value returned by the function
- Return value may be of any data type other than array and function types
- If return_type is omitted the value returned is assumed to be an int type by default
- Void is specified in the place of return_type if the function returns no value

function_name

- Function name is an identifier
- It can not begin with underscore because such names are reserved for the
  use of C library
- It can contain up to 31 characters, but it is wise to have a maximum of six
  characters to distinguish different function names by a linker.

The formal arguments allow information to be transferred from the calling portion of the program to the function. They are also known as parameters or **formal parameters**. The arguments are called **actual parameters** when they are using in function reference (function call). The names of actual parameters and formal parameters may be either same or different but their data type should be same. All formal arguments must be declared after the definition of function.

The function body follows the function header and it is always enclosed in braces. The body of the function is composed of declarations and statements. The statements describe the action to be performed by the functions. Information is returned from the function to the calling portion of the program via the **return** statement. The return statement also causes control to be returned to the point from which the function was accessed.

In general, the return statement is written as

return expression;

The value of the expression is returned to the calling portion of the program. The return statement can be written without the expression. Without the expression, return statement simply causes control to revert back to the calling portion of the program without any information transfer. The point to be noted here is that only one expression can be included in the return statement. Thus, a function can return only one value to the calling portion of the program via return.

It is not necessary to include a return statement altogether in a program. If a function reaches the end of the block without encountering a return statement, control simply reverts back to the calling portion of the program without returning any information.

There are some points that must be kept in mind while defining a function,

- A function cannot be defined more than once in a program
- One function can't be defined within another function definition
- Function definition may appear in any order
- If a program uses several functions, the functions can be distributed in many files. But a function definition cannot be distributed in more than one file by splitting the same definition.
- Usually the function name is given as the file name when several files are used to write a complete program.

FUNCTION DECLARATION

Whenever a function is invoked in another function, it must be declared before use. Such a declaration is known as **function declaration** or **function prototype**. Function declaration always end with semicolon. The general format is

```
return_type function_name (parameter_list);
```

in function declaration, parameter names in the parameter list are optional. Hence it is possible to have the data type of each parameter without mentioning the parameter name as shown below.

```
return_type function_name (data_type1, data_type2, ……, data_type n);
```

This declaration helps the compiler in detecting inconsistent type of function called and mismatching types of parameters used. If the value returned by a function is int type, the declaration of function is optional. For all the other type of functions, declaration is mandatory. The return_type is void when the function returns no value.
Function prototypes are desirable because they further facilitate error checking between the calls to a function and the corresponding function definition.

Let us examine examples of some function prototypes

```
int example (int, int); or int example (int a, int b);

void example (void);

void fun (char, long); or void fun (char c, long f );
```

The names of the arguments within the function declaration need not be declared elsewhere in the program, since these are “dummy” argument names recognized only within the declaration.

Observe that function declaration is different from function definition.

	Function Definition	Function Declaration
1	There is no semicolon at the end of the Closing parenthesis of parameter list	There is a semicolon at the end of the closing Parenthesis of parameter list
2	The function body follows it	The function body does not follow it
3	Mandatory for all functions	Optional for function returning int value

FUNCTION CALL

A function can be accessed by specifying its name, followed by a list of parameters or arguments enclosed in parentheses and separated by commas. If the function call does not require any arguments an empty pair of parentheses must follow the function’s name. The general format of a function call is function_name (e1, e2,…., en)

where e1, e2….., en are argument expressions named as actual arguments or actual parameters.
If a function returns a value, the function call may appear in any C expression and the returned value is used as an operand in the evaluation of the expression.

Let us now consider an example of function without returning any information for more clarity.

```
#include<stdio.h>
int main()

{
      int x, y;
      int maxin( int, int ); /*function declaration*/
      printf( "\n Enter two integer values:" );
      scanf( "%d %d", &x, &y );
      maxin(x,y); /*call to function*/
      return 0;
}
maxin( int x, int y) /*function definition*/
{
    int z;
    z = ( x >= y ) ? x : y;
    printf( "\n Maximum value %d", z );
    return;
}
```

This 'maxin' function do not return any value to the calling program, it simply returns the control to the calling programs, so if it is even not present, then also program will work efficiently.

Most C compilers permit the keyword **void** to appear as a type specifies when defining a function that does not return anything. So the function definition will look like this if void is add to it

```
void maxi (int, int);
```

Let us consider another example of function.

```
#include <stdio.h>
main()
{
int a, b, c;
printf("\n Enter two numbers");
scanf("%d %d", &a,&b);
c = sum_v( a, b );
printf("\n The sum of two variables is %d",c);
}

sum_v( int a, int b )
{
int d;
d = a + b;
return d;
}
```

This program returns the sum of two variables a and b to the calling program from where sum_v is executing. The sum is present in the variable c through the ‘return d’ statement. There may be several different calls to the same function from various places within a program. The actual parameters may differ from one function call to another. Within each function call, the actual arguments must correspond to the formal arguments in the function definition, i.e. the number of actual arguments must be same as the number of formal arguments and each actual argument must be of the same data type as it s corresponding formal argument.

Let us consider an example.

```
#include <stdio.h>
main()
{
        int a, b, c, d;
        printf( "\n Enter value of a , b,c and d" );
        scanf( "%d %d %d", &a, &b, &c, &d );
        d = maxin( a, b );
        printf("\n maximum = %d", maxin(c, d) );
}
```

```
maxi( int x, int y)
{
        int z;
        z = ( x >= y ) ? x : y;
        return z;
}
```

The function maxin is accessed from two different places in main. In the first call actual arguments are a, b and in the second call c, d are the actual arguments.

PASSING ARGUMENT TO A FUNCTION

Arguments can be passed to a function by two methods, they are called **passing by value** and **passing by reference** (address).

Pass by Value (Call by value)

When a single value is passed to a function via an actual argument, the value of the actual argument is copied into the function. Therefore, the value of the corresponding formal argument can be altered within the function, but the value of the actual argument within the calling routine will not change. This procedure for passing the value of an argument to a function is known as passing by value.

Let us consider an example for better understanding.

```
#include <stdio.h>
main()
{
        int x = 3;
        printf( "\n x = %d (from main, before calling the function)", x );
        change( x );
        printf( "\n\nx = %d (from main, after calling the function)", x );
}
```

```
change( int x)
{
        x = x + 3;
        printf( "\nx = %d (from the function, after being modified)",x );
        return;
=}
```

The original value of x (i.e. x=3) is displayed when main begins execution. This value is then passed to the function change, where it is sum up by 3 and the new value displayed. This new value is the altered value of the formal argument that is displayed within the function. Finally, the value of x within main is again displayed, after control is transferred back to main from change.

```
        x = 3 (from main, before calling the function)
        x = 6 (from the function, after being modified)
        x = 3 (from main, after calling the function)
```

Passing an argument by value allows a single-valued actual argument to be written as an expression rather than being restricted to a single variable. But it prevents information from being transferred back to the calling portion of the program via arguments. Thus, passing by value is restricted to a one-way transfer of information.

Pass by reference (Call by address)

When an address of variable is passed to a function via an actual argument, the value of the actual argument is copied into the function, where it is received in a pointer variable. Therefore, the value of the corresponding formal argument can be altered within the function, the effect of this will be reflected in the actual argument within the calling routine. This procedure of passing the address of an argument to a function is known as passing by address.

Let us consider an example which explains pass by reference method.

```
#include <stdio.h>
main()
{
        int x = 3;
        printf( "\n x = %d (from main, before calling the function)",x );
        change( &x );
        printf( "\n\nx = %d (from main, after calling the function)",x );
}
change( int *x)
{
        *x = *x + 3;
        printf( "\nx = %d (from the function, after being modified)",*x );
        return;
}
```

The original value of x (i.e. x=3) is displayed when main begins execution. The address of the variable x is then passed to the function change, where it is sum up by 3 and the new value displayed. This new value is the altered value of the formal argument that is displayed within the function. Finally, the value of x within main is again displayed, after control is transferred back to main from **change**.

```
        x = 3 (from main, before calling the function)
        x = 6 (from the function, after being modified)
        x = 6 (from main, after calling the function)
```

Arrays are passed to a function differently than single-valued entities. If an array name is specified as an actual argument, the individual array elements are not copied to the function. Instead the location of the array is passed to the function. If an element of the array is accessed within the function, the access will refer to the location of that array element relative to the location of the first element. Thus, any alteration to an array element within the function will carry over to the calling routine.

RECURSIVE FUNCTION CALLS

Based on the function call, functions can be classified as Recursive and Non-recursive functions. If a function calls itself in the function body of its function definition, it is known as a direct recursive call. If a function calls another function, which in turn calls the first function then it is called as an indirect recursive call. In either case, the definition of function results in a circular chain of calls of the same function. Hence the execution will continue indefinitely. To terminate this indefinite execution, a statement without recursive call must be executed. Therefore, every recursive function must have a proper terminating condition to provide a non-recursive exit.

The process by which a function calls itself repeatedly, until some specified condition has been satisfied, is known as **recursion**. The process is used for repetitive computations in which each action is stated in terms of a precious result. In order to solve a problem recursively, two conditions must be satisfied. The problem must be written in a recursive form, and the problem statement must include a stopping condition. The best example of recursion is calculation of factorial of an integer quantity, in which the same procedure is repeating itself.

Let us consider the example of computing factorial using a recursion

```
#include <stdio.h>
main()
{
        int number;
        long int fact( int number );
        printf( "\nEnter number" );
        scanf( "%d", & number );
        printf("Factorial of number is % ld\n", fact( number ));
}


long int fact( int number )
{
        if( number <= 1 )
```

```
                return( 1 );
        else
                return( number * fact( number - 1 ));

}
```

The point to be noted here is that the function ‘fact’ calls itself recursively, with an actual argument (n-1) that decrease in value for each successive call. The recursive calls terminate the value of the actual argument becomes equal to 1.

When a recursive program is executed, the recursive function calls are not executed immediately. Instead of it, they are placed on a stack until the condition that terminates the recursion is encountered. The function calls are then executed in reverse order, as they are popped off the stack.

The use of recursion is not necessarily the best way to approach a problem, even though the problem definition may be recursive in nature.

STORAGE CLASSES – AUTOMATIC, EXTERNAL, STATIC VARIABLES

There are four different storage-class specification in ‘C’, automatic, external, static and register. They are identified as auto, extern, static and register respectively.

Automatic variables are always declared within a function and are local to the function in which they are declared, that is their scope is confined to that function. Automatic variables defined in different functions will therefore be independent of one another. The location of the variable declarations within the program determine the automatic storage class, the keyword auto is not required at the beginning of each variable declaration.

These variables can be assigned initial value by including appropriate expressions within the variable declarations. An automatic variable does not retain its value once control is transferred out of its defining function. It means any value assigned to an automatic variable within a function will be lost once the function is exited. The scope of an automatic variable can be smaller than an entire function. Automatic variables can be declared within a single compound statement.

External variables are not confined to single functions. Their scope extends from the point of definition through the remainder of the program. External variable are recognized globally, that means they are recognized throughout the program, they can be accessed from any function that falls within their scope. They retain their assigned values within their scope. Therefore, an external variable can be assigned a value within one function and this value can be used within another function. With the use of external variables one can transfer the information between functions.

External variable definitions and external variable declarations are not the same thing. An external variable definition is written in the same manner as an ordinary variable declaration. The storage-class specifier **extern** is not required in an external variable definition, because these variables will be identified by the location of their definition within the program. An external variable declaration must begin with the storage class specifier **extern**. The name of the external variable and its data type must agree with the corresponding external variable definition that appears outside of the function.

The declaration of external variables cannot include the assignment of initial values. External variables can be assigned initial values as a part of the variable definitions, but the initial values must be expressed as constants rather than as expression. These initial values will be assigned only once, at the beginning of the program. If an initial value is not included in the definition of an external variable, the variable will automatically be assigned a value of zero.

Static variables are defined within individual functions and therefore have a same scope as automatic variables, i.e. they are local to the functions in which they are defined. **Static** variables retain their values throughout the program. Thus, if a function is exited and reentered later, the static variables defined within that function will retain their former values. Static variables are defined within a function in the same manner as automatic variables, but its declaration must begin with the static storage class designation. They cannot be accessed outside of their defining function. Initial values can be included in static variable declarations. The initial value must be expressed as constants, not expression, the initial values are assigned to their respective variables at the beginning of program execution. The variables retain these values throughout the program, unless different values are assigned during the program. This is all for storage classes auto, extern and static.

Let us consider one static variable

```
static int a;
```

If the keyword static is replaced with the keyword auto, the variable is declared to be of storage class auto. If a static local variable is assigned a value the first time the function is called, that value is still there when the function is called second time.

```
display_number()
{
        static int number=2;
        printf("number=%d\n", number);
        number++;
}
```

When the first time display_number is called, it prints the value 2, to which number is initialized. Then number is incremented to 3, and terminates. The second time display_number is called, it prints the value of 3. On the third call, the value printed is 4 and so on. Point to be noted here is that the initialization is not performed after the first call. An initialization used in a declaration occurs only once-when the variable is allocated. Since a static variable is allocated only once, the initialization occurs only during the entire program, no matter how many times the function is called. When the display_number function in the example is called the second time, the value found in the variable number is the value left there by the previous call to the function.

Now let us try to solve some assignments.

ASSIGNMENTS.

1. What is a function explain with a suitable example.
2. Is it possible to have more than one main() function in a C program? Why?
3. Explain **call by value** and **pass by reference** methods of parameter passing.
4. Write a C program to interchange the values given in two variables using a user defined function.
5. Write a function subprogram stat (a, n, sum, ave) which calculates the sum and average of the elements of a linear array A with N elements.
6. Write a program to print the first n Fibonacci sequence of numbers suing recursion.
   (Hint : The first 5 elements of a Fibonacci sequence : 1 1 2 3 5)

REFERENCE

- B. W. Kernighan and D. M. Ritchie, The C Programming Language, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
- Yashavant Kanetkar; Let us C, BPB Publications, New Delhi.
- Greg Perry, Absolute Beginners' guide to C, SAMS publishing, April 1994.

In this session we have learnt about functions, how to define, declare functions and function calls. Now you know how to execute functions from different places of a C program. We are also discussed a very useful feature of functions named recursion followed by the use of different storage classes auto, extern and static. In upcoming sessions we will discuss more and more interesting features of C language. Bye for now.