Unit-3

Functions: Introduction – using functions – Function declaration/ prototype – Function definition – function call – return statement – Passing parameters – Scope of variables – Storage Classes – Recursive functions.

3.1: Introduction: Functions are a group of statements that have been given a name. This allows you to break your program down into manageable pieces and reuse your code.

- **The advantages of functions are**:
  - Function makes the lengthy and complex program easy and in short forms. It means large program can be sub-divided into self-contained and convenient small modules having unique name.
  - The length of source program can be reduced by using function by using it at different places in the program according to the user's requirement.
  - By using function, memory space can be properly utilized. Also less memory is required to run program if function is used.
  - They also allow more than one person to work on one program.
  - Function increases the execution speed of the program and makes the programming simple.
  - By using the function, portability of the program is very easy.
  - It removes the redundancy (occurrence of duplication of programs) i.e. avoids the repetition and saves the time and space.
  - Debugging (removing error) becomes very easier and fast using the function sub-programming.
  - Functions are more flexible than library functions.
  - Testing (verification and validation) is very easy by using functions.
  - User can build a customized library of different functions used in daily routine having specific goal and link with the main program similar to the library functions.

The functions are classified into standard functions and user-defined functions.

The standard functions are also called library functions or built in functions. All standard functions, such as sqrt(), abs(), log(), sin(), pow() etc. are provided in the library of functions. These are selective.

Most of the applications need other functions than those available in the software. These functions must be written by the user (programmer), hence, the name user-defined functions.

You can write as many functions as you like in a program as long as there is only one main ().

As long as these functions are saved in the same file, you do not need to include a header file. They will be found automatically by the compiler.

There are 3 parts for using functions:

Declaring them (prototypes).
Calling them,
Defining them and

3.2 Function Declaration (Prototype):

Function declaration means specifying the function as a variable depending on the return value.

It is declared in the declaration part of the main program.

The default return value from a function is always an integer.

If the function is returning a value other than an integer, it must be declared with the data type of the value it returns.

The general form of function declaration in ANSI form is as follows:

Type_of_return_value function_name (parameter types);

3.3 Function Definition :

Defining a function means writing an actual code for the function which does a specific and identifiable task.

The general form of a ANSI method of function definition is as follows:

```
 type_specifier function_name (formal parameters)
{
variable declaration;           /* with in the function */
body of function;
*
*
return (value_computed);
 }
```

type_specifier specifies the type of value that the function's return statement returns. If nothing is returned to the calling function, then data type is **void**.

function_name is a user-defined function name. It must be a valid C identifier.

formal parameters is the type declaration of the variables of parameter list.

return is a keyword used to send the output of the function, back to the calling function.

{ is the beginning of the function.

} is the end of function.

All the statements placed between the left brace and the corresponding right brace constitute the **body of a function**.

3.4 Calling functions:

- When you wish to use a function, you can "call" it by name.
- Control is sent to the block of code with that name.
- Any values that the function needs to perform its job are listed in the parentheses that immediately follow the name are called arguments.
- The computer will retrieve the values held by the variables and hand them off to the function for its use.
- The number of arguments (or values) must match the variables in the definition by type and number.
- A semicolon must be present, when the function is called within the main() function.
- The general form of a ANSI method of function call is as follows:
  - function_name (actual parameters);

3.5 return statement

The return causes an immediate exit from a function to the point from where the function is called. It may also be used to return one value per call. All functions, except those of type void, return a value. If no return statement is present, most compilers will cause a 0 to be returned. The return statement can be any one of the types as shown below:

1. return;
2. return ();
3. return (constant);
4. return (variable);
5. return (expression);

6. return (conditional expression);
7. return (function);

The first and second return statements, does not return any value and are just equal to the closing brace the function. If the function reaches the end without using a return statement, the control is simply transferred back to the calling portion of the program without returning any value. The presence of empty return statement is recommended in such situations.

The third return statement returns a constant function. For example:

```
if (x <= 10)
        return (1);
```

The fourth return statement returns a variable to the calling function. For example:

```
if (x <= 10)
        return (x);
```

The fifth return statement returns a value depending upon the expression specified inside the parenthesis. For example:

```
return (a + b * c);
```

The sixth return statement returns a value depending upon the result of the conditional expression specified inside the parenthesis. For example:

```
return (a > b ? 1 : 2);
```

The last return statement calls the function specified inside the parenthesis and collects the result obtained from that function and returns it to the calling function. For example:

```
return (pow (4, 2));
```

The parenthesis used around the expression or value used in a return statement is optional.

3.6 Passing parameters

Arguments and parameters are the variables used in a program and a function. Variables used in the calling function are called arguments. These are written within the parentheses followed by the name of the function. They are also called actual parameters, as they are accepted in the main program (or calling function).

Variables used in the function definition (called function) are called parameters. They are also referred to as formal parameters, because they are not the accepted values. They receive values from the calling function. Parameters must be written within the parentheses followed by the name of the function, in the function definition.

Parameter passing mechanisms:

A method of information interchange between the calling function and called function is known as parameter passing. There are two methods of parameter passing:

1. Call by value
2. Call by reference

3.6.1. Call by Value:

The call-by-value method copies the value of an argument into the formal parameter of the function. Therefore, changes made by the function to the parameters have no effect on the variables used to call it.

Example:

```
# include <stdio.h>
void swap (int x, int y);               /* function prototype */
main ()
{
        int x, y;
        x = 10;
        y = 20;
        swap (x, y);                    /* values passed */
}

void swap (int a, int b)                /* function definition */
{
        int temp;

        temp = a;
        a = b;
        b = temp;
        printf ("%d %d\n", a, b);
 }
```

3.6.2. Call by Reference:

The call by reference method copies the address of an argument into the formal parameter. Inside the function, this address is used to access the argument used in the call. In this way, changes made to the parameter affect the variable used in the call to the function.

Call by reference is achieved by passing a pointer as the argument. Of course, in this case, the parameters must be declared as pointer types. The following program demonstrates this:

Example:

```
# include <stdio.h>
void swap (int *x, int *y);                     /* function prototype */
main ()
{
        int x, y;
        x = 10;
        y = 20;
        swap (&x, &y);                  /* addresses passed */
        printf ("%d %d\n", x, y);
}
void swap (int *a, int *b)
{
        int temp;
        temp = *a;
        *a = *b;
        *b = temp;
 }
```

The differences between call by value and call by reference:

Call by value	Call by reference
int a;	int a;
Formal parameter ‘a’ is a local variable.	Formal parameter is ‘a’ local reference.
It cannot change the actual parameter.	It can change the actual parameter.
Actual parameter may be a constant, a variable, or an expression.	Actual parameter must be a variable.

3.7 Types of functions

We have some other type of functions where the arguments and return value may be present or absent. Such functions can be categorized into:

1. Functions with no arguments and no return value.
2. Functions with arguments and no return value.
3. Functions with no arguments but return value.
4. Functions with arguments and return value.

Functions with no arguments and no return value:

Here, the called function does not receive any data from the calling function and, it does not return any data back to the calling function. Hence, there is no data transfer between the calling function and the called function.
Example:

This program illustrates the function with no arguments and no return value.

```
# include <stdio.h>
void read_name ();
main ()
{
        read_name ();
}
void read_value ()                    /*no return value */
{
        char name [10];
        printf (“Enter your name: ”);
        scanf (“%s”, name);
        printf (“\nYour name is %s: ”, name);
}
```

Output:

```
        Enter your name: Herbert
        Your name is: Herbert
```

Functions with arguments and no return value:

Here, the called function receives the data from the calling function. The arguments and parameters should match in number, data type and order. But, the called function does not return and value back to the calling function. Instead, it prints the data in its scope only. It is one-way data communication between the calling function and the called function.
Example:

This program illustrates the function with arguments but has no return value.

```
#include <stdio.h>
void maximum (x, y);
main()
{
        int x, y;
        printf ("Enter the values of x and y: ");
        scanf ("%d %d", &x, &y);
        maximum (x, y);
}
void maximum (int p, int q)                    /* function to compute the maximum */
{
        if (p > q)
                printf ("\n maximum = %d", p);
        else
                printf ("\n maximum = %d" q);
        return;
}
```

Output:

```
        Enter the values of x and y: 14 10
        maximum = 14
```

Function with no arguments but return value:
Here, the called function does not receive any data from the calling function. It manages with its local data to carry out the specified task. However, after the specified processing the called function returns the computed data to the calling function. It is also a one-way data communication between the calling function and the called function.

Example:

This program illustrates the function with no arguments but has a return value.

```
#include <stdio.h>
float total ();
main ()
{
        float sum;
        sum = total ();
        printf ("Sum = %f\n", sum);
}
float total ()
{
        float x, y;
        x = 20.0;
        y = 10.0;
        return (x + y);
}
Output:
Sum = 30.000000
```

Function with arguments and return value:

When a function has arguments it receives data from the calling function and does some process and then returns the result to the called function. In this way the main() function will have control over the function.

Example:

```
/* program to find the sum of first N natural numbers */
#include <stdio.h>
int sum (int x);                        /* function prototype*/
void main ()
{
        int n;
        printf (“Enter the limit: ”);
        scanf (“%d”, &n);
        printf (“sum of first %d natural numbers is: %d”, n, sum(n));
}
int sum(int x)
{
        int i, result = 0
        for (i=1; i <= x; i++)
                result += i;
        return (result);
}
```

Output:

```
Enter the limit: 5
Sum of first 5 natural numbers is: 15
```

The main() is the calling function which calls the function sum(). The function sum() receives a single argument. Note that the called function (i.e., sum ()) receives its data from the calling function (i.e., main()). The return statement is employed in this function to return the sum of the n natural numbers entered from the standard input device and the result is displayed from the main() function to the standard output device. Note that int is used before the function name sum() instead of void since it returns the value of type int to the called function.

Important points to be noted while calling a function:

- Parenthesis are compulsory after the function name.
- The function name in the function call and the function definition must be same.
- The type, number, and sequence of actual and formal arguments must be same.

- A semicolon must be used at the end of the statement when a function is called.
- The number of arguments should be equal to the number of parameters.
- There must be one-to-one mapping between arguments and parameters. i.e. they should be in the same order and should have same data type.
- Same variables can be used as arguments and parameters.
- The data types of the parameters must match or be closely compatible. It is very risky to call a function using an actual parameter that is floating point data if the formal parameter was declared as an integer data type. You can pass a float to a double, but should not do the opposite. You can also pass a short int to an int, etc. But you should not pass any type of integer to any type of floating point data or do the opposite.

Nested Functions:

We have seen programs using functions called only from the main() function. But there are situations, where functions other than main() can call any other function(s) used in the program. This process is referred as nested functions.

Example:

```
#include <stdio.h>
void func1();
void func2();
void main()
{
        printf ("\n Inside main function");
        func1();
        printf ("\n Again inside main function");
}
void func1()
{
        printf ("\n Inside function 1");
        func2();
        printf ("\n Again inside function 1");
}
void func2()
{
        printf ("\n Inside function 2");
}
```

Output:

```
        Inside main function
        Inside function 1
        Inside function 2
        Again inside function 1
        Again inside main function
```

Uses two functions func1() and func2() other than the main() function. The main() function calls the function func1() and the function func1() calls the function func2().

3.8 Scope of variables

- A scope is a region of the program, and the scope of variables refers to the area of the program where the variables can be accessed after its declaration.
- In C every variable defined in scope.
- In C programming, variable declared within a function is different from a variable declared outside of a function.

- The variable can be declared in three places. These are:

Position	**Type**
Inside a function or a block.	local variables
Out of all functions.	Global variables
In the function parameters.	Formal parameters

Local Variables

- Variables that are declared within the function block and can be used only within the function is called local variables.
- Local scope or block scope
    - A local scope or block is collective program statements put in and declared within a function or block (a specific region enclosed with curly braces) and variables lying inside such blocks are termed as local variables.
    - All these locally scoped statements are written and enclosed within left ({) and right braces (}) curly braces.
    - There's a provision for nested blocks also in C which means there can be a block or a function within another block or function.
    - So it can be said that variable(s) that are declared within a block can be accessed within that specific block and all other inner blocks of that block, but those variables cannot be accessed outside the block.

```
#include <stdio.h>
int main ()
{
        /* local variable definition and initialization */
        int x,y,z;
        /* actual initialization */
        x = 20;
        y = 30;
        z = x + y;
        printf("value of x = %d,y = %d and z = %d\n", x, y, z);
        return 0;
}
```

Global variables

- Variables that are declared outside of a function block and can be accessed inside the function is called global variables.
- Global scope
    - Global variables are defined outside a function or any specific block, in most of the case, on the top of the C program.
    - These variables hold their values all through the end of the program and are accessible within any of the functions defined in your program.
    - Any function can access variables defined within the global scope, i.e., its availability stays for the entire program after being declared.

```
#include <stdio.h>
/* global variable definition */
int z;
int main ()
{
        /* local variable definition and initialization */
        int x,y;
        /* actual initialization */
        x = 20;
        y = 30;
        z = x + y;
        printf ("value of x = %d, y = %d and z = %d\n", x, y, z);
        return 0;
}
```

- A program can have same name for local and global variables but the value of local variable inside a function will take preference. Here is an example –

```
        #include <stdio.h>
         /* global variable declaration */
        int g = 20;
        int main ()
        {               /* local variable declaration */
        int g = 10;
        printf ("value of g = %d\n", g);
        return 0;
        }
```

When the above code is compiled and executed, it produces the following result –

```
value of g = 10
```

Formal Parameters

Formal parameters, are treated as local variables with-in a function and they take precedence over global variables. Following is an example –

```
#include <stdio.h>
Int a=20;   /* global variable declaration */
int main ()
{ /* local variable declaration in main function */
int a = 10; b=20;c=0;
printf ("value of a in main() = %d\n", a);
c = sum( a, b);
printf ("value of c in main() = %d\n", c);
return 0;
}
/* function to add two integers */
int sum(int a, int b)
{
printf ("value of a in sum() = %d\n", a);
printf ("value of b in sum() = %d\n", b);
return a + b;
}
```

When the above code is compiled and executed, it produces the following result –

```
value of a in main() = 10
value of a in sum() = 10
value of b in sum() = 20
value of c in main() = 30
```

- When a local variable is defined, it is not initialized by the system, you must initialize it yourself. Global variables are initialized automatically by the system when you define them as follows −

Data Type	**Initial Default Value**
int	0
char	'\0'
float	0
double	0
pointer	NULL

It is a good programming practice to initialize variables properly, otherwise your program may produce unexpected results, because uninitialized variables will take some garbage value already available at their memory location.

3.9 Storage Classes

Storage classes are used to define the scope (visibility) and life-time of variables and/or functions. Every variable and function in C has two attributes: type and storage class. There are four storage classes:

- auto
- static
- extern or global and
- register

The auto storage class:

Variables declared within function bodies are automatic by default. If a compound statement starts with variable declarations, then these variables can be acted on within the scope of the enclosing compound statement. A compound statement with declarations is called a block to distinguish it from one that does not begin with declarations.
Declarations of variables within blocks are implicitly of storage class automatic. The key-word auto can be used to explicitly specify the storage class. An example is:

```
auto int a, b, c;
auto float f;
```

Because the storage class is auto by default, the key-word auto is seldom used.

When a block is entered, the system allocates memory for the automatic variables. With in that block, these variables are defined and are considered "local" to the block. When the block is exited, the system releases the memory that was set aside for the automatic variables. Thus, the values of these variables are lost. If the block is reentered, the system once again allocates memory, but previous values are unknown. The body of a function definition constitutes a block if it contains declarations. If it does, then each invocation of the function set up a new environment.

The static storage class:

It allows a local variable to retain its previous value when the block is reentered. This is in contrast to automatic variables, which lose their value upon block exit and must be reinitialized. The static variables hold their values throughout the execution of the program. As an example of the value-retention use of static, the outline of a function that behaves differently depending on how many times it has been called is:

```
        void fun (void)
        {
                static int cnt = 0
                ++ cnt;
                if (cnt % 2 == 0)
                        . . .                           /* do something */
                else
                        . . .                           /* do something different */
        }
```

The first time the function is invoked, the variable cnt is initialized to zero. On function exit, the value of cnt is preserved in memory. Whenever the function is invoked again, cnt is not reinitialized. Instead, it retains its previous value from the last time the function was called. The declaration of cnt as a static int inside of fun () keeps it private of fun (). If it was declared outside of the function, then other functions could access it, too.

Static variables are classified into two types.

1. internal static variables
2. external static variables

External static variables are declared outside of all functions including the main() function. They are global variables but are declared with the keyword static. The external static variables cannot be used across multi-file program.

A static variable is stored at a fixed memory location in the computer, and is created and initialised once when the program is first started. Such a variable maintains its value between calls to the block (a function, or compound statement) in which it is defined. When a static variable is not initialized, it takes a value of zero.

The storage class **extern:**

One method of transmitting information across blocks and functions is to use external variables. When a variable is declared outside a function, storage is permanently assigned to it, and its storage class is extern. Such a variable is considered to be global to all functions declared after it, and upon exit from the block or function, the external variable remains in existence. The following program illustrates this:

```
# include <stdio.h>
int a = 1, b = 2, c = 3;                        /* global variable*/
int fun (void);                                 /* function prototype*/
int main (void)
{
        printf (“%3d\n”, fun ());               /* 12 is printed */
        printf (“%3d%3d%3d\n”, a, b, c);        /* 4   2  3 is printed */
}
int fun (void)
{
        int b, c;
        a = b = c = 4;                          /* b and c are local */
        return (a + b + c);                     /* global b, c are masked*/
}
```

Note that we could have written:

```
 extern int a = 1, b = 2, c = 3;
```

The extern variables cannot be initialised in other functions whereas we can use for assignment.

This use of extern will cause some traditional C compilers to complain. In ANSI C, this use is allowed but not required. Variables defined outside a function have external storage class, even if the keyword extern is not used. Such variables cannot have auto or register storage class.

The keyword extern is used to tell the compiler to "look for it else where, either in this file or in some other file". Let us rewrite the last program to illustrate a typical use of the keyword extern:

In file file1.c

```
#include <stdio.h>
# include "file2.c"
int a=1, b=2, c=3;            /* external variable */
int fun (void)
int main (void)
{
        printf ("The values of a, b and c are: %3d%3d%3d\n", a, b, c);
        printf ("The sum of a + b + c is: %3d\n", fun ());
        printf ("The values of a, b and c are: %3d%3d%3d\n", a, b, c);
        return 0;
}
```

In file file2.c

```
int fun (void)
{
        extern int a;                 /* look for it elsewhere */
        int b, c;
        a = b = c = 4;
        return (a + b + c);
}
```

Output:

```
The values of a, b and c are: 1   2   3
The sum of a + b + c is: 12
The values of a, b and c are: 4   2   3
```

The two files can be compiled separately. The use of extern in the second file tells the compiler that the variable a will be defined elsewhere, either in this file or in some other. The ability to compile files separately is important when writing large programs.

External variables never disappear. Because they exist throughout the execution life of the program, they can be used to transmit values across functions. They may, however, be hidden if the identifier is redefined.

Information can be passed into a function two ways; by use of external variable and by use of the parameter passing mechanism. The use of the parameter mechanism is the best preferred method. Don't overuse 'extern' variables. It is usually better to pass lots of arguments to functions rather than to rely on hidden variables being passed (since you end up with clearer code and reusuable functions).

The register storage class:

This is like `auto' except that it asks the compiler to store the variable in one of the CPU's fast internal registers. In practice, it is usually best not to use the `register' type since compilers are now so smart that they can do a better job of deciding which variables to place in fast storage than you can.

The use of storage class register is an attempt to improve execution speed. When speed is a concern, the programmer may choose a few variables that are most frequently accessed and declare them to be of storage class register. Common candidates for such treatment include loop variables and function parameters. The keyword register is used to declare these variables.

Examples:

1) register int x;
2) register int counter;

The declaration register i; is equivalent to register int i;

Storage Classes	**Storage Place**	**Default Value**	**Scope**	**Lifetime**
auto	RAM	Garbage Value	Local	Within function
extern	RAM	Zero	Global	Till the end of the main program Maybe declared anywhere in the program
static	RAM	Zero	Local	Till the end of the main program, Retains value between multiple functions call
register	Register	Garbage Value	Local	Within the function

3.10 Recursive functions

Recursion in C language is basically the process that describes the action when a function calls a copy of itself in order to work on a smaller problem. Recursive functions are the function that calls themselves and this type of function calls are known as recursive calls. The recursion in C generally involves various numbers of recursive calls. It is considered to be very important to impose a termination condition of recursion. Recursion code in the C language is generally shorter than the iterative code and it is known to be difficult to understand.

Recursion cannot be applied to all the problem, but Recursion in C language is very useful for the tasks that can be generally be defined in terms of similar subtasks but it cannot be applied to all the problems. For instance: recursion in C can be applied to sorting, searching, and traversal problems.

As function call is always overhead, iterative solutions are more efficient than recursion. Any of the problems that can generally be solved recursively; it can be also solved iteratively. Apart from these facts, there are some problems that are best suited to only be solved by the recursion for instance: tower of Hanoi, factorial finding, Fibonacci series, etc.

What is Recursive Function?

The working of a recursive function involves the tasks by dividing them generally into the subtasks. Some specific subtask have a termination condition defined that has to be satisfied by them. In the next step, the recursion in C stops and the final result is derived from the function.

The base case is the case at which the function doesn't recur in C and there are instances where the function keeps calling itself in order to perform a subtask and that is known as the recursive case.

Recursive functions are two types:

1. Direct Recursion
2. In Direct Recursion

Direct Recursion: if a function calls itself then it is direct recursion.

Example:
```
 void main()
{
------
main();
}
```

Indirect recursion: If a function calls another function and if that function calls it self. Then it is indirect recursion.

Example:
```
 Void main()
{
  --
  --
 Print();
--
--
}
Void Print()
{
 --
--
Print();
--
}
```

Check: Output: 0 1 2 3 4 5 6 7 8 9

Example:

```
#include <stdio.h>
void printnum ( int begin )
{
   printf( "%d", begin );
   if ( begin < 9 )         /* The base case is when begin is no longer */
   {                        /* less than 9 */
      printnum ( begin + 1 );
   }
   /* display begin again after we've already printed everything from 1 to 9
    * and from 9 to begin + 1 */
   printf( "%d", begin );
}

int main()
{
   printnum(0);
   getchar();
}
```