# UNIT – III (PART-II) & UNIT IV(PART-I)

**Function:** it is defined as self contained block of code to perform a task.

Functions can be categorized to system-defined functions and user-defined functions.

**System defined functions:** These are functions that are supported along with the programming language. The block of instructions of these functions and the task to be performed is predefined. For e.g. clrscr( ), printf ( ), scanf( ), pow( ), sqrt( ) etc.

**User defined functions:** These are functions that are not supported along with the programming language. The block of instructions of these functions and the task to be performed is defined by the programmer.

The following are the advantages of functions:

- Reusability of code i.e. block of code defined can be used when required.
- Facilitates Modular Programming i.e. breaking the complex problem into small manageable sub-problems called as modules and also makes the process of debugging easier.
- Reduces the length of the program

**Note:** The block of instructions defined in the function gets executed only when a **function call** is invoked. Where **function call** is the reference made in the **calling function** by specifying the name of the function along with arguments if any exists and should be terminated with semicolon.

Where **calling function** is a function that comprises of references to the name(s) of functions along with arguments if any exists.

for e.g.

```
main( )   /* calling function */
{
    clrscr( );  /* function call  */
}
```

**Defining User defined functions:**

To understand user-defined functions one should understand the following elements:

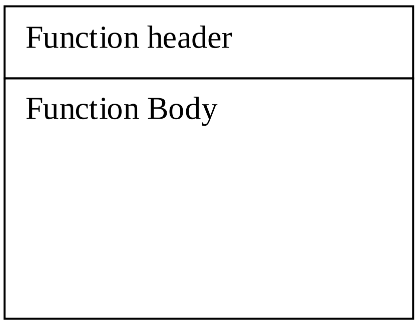
- Function definition
- Function declaration
- Function call

**Function Definition:** The function definition consists of two parts namely

1. Function header
2. Function body

The general format of the function definition is shown below:

```
Return type  function-name(arguments)
             Argument declaration;
             {
                Local variable declaration;
                Executable statement1;
                Executable statement 2;
                Return statement;
             }
```



**Function Header:**

The function header comprises of three parts namely

1. Return type

2. Function name
3. arguments

**Return type:** This part of the function header specifies the type of data (primitive/user-defined) that the function will return.
**Function Name:** This part of the function header takes a valid name to identify the function and this name should follow the rules for naming a user defined identifier.
**Arguments:** This part of the function header receives the values from the actual arguments of the function call. The number and type of arguments from function call should match with the arguments in the function definition. The arguments in the function definition can be declared for its type in the next line. The arguments in the function definition are called as **formal arguments.**

**Function body:**
The function body comprises of the following parts enclosed in flower Braces:
1. Local variable declaration
2. Executable statements
3. Return statement

**Local variable declaration:** Variables that are needed in that function are declared for its type.
**Executable Statements:** These are statements that are needed in the function to assign a value or to read a value or to print the value or to perform some computations.
**Return statement:** Every function body ends with return statement. This statement is used to return a value to the calling function. If the function does not return any value then the return statement is written as
**Return;** or **return 0;**
If the function returns any value then the return statement is written as **Return value;** or **return expression;**
The function definition can be done either above the **main function** or after the **main function.**

**Example of function definition:**

```
float mul(float x, float y)
{
        float result;
        result=x*y;
        return result;
}
```

In the above example in the function header the return type is float and the name of the function is mul and the arguments are x and y which are declared of type float. In the function body, result is a variable that is needed in that function and is declared locally for its type. Result=x*y is the executable statement and the return statement returns the value available in the variable result.

**Function Declaration:** Declaring a function also called as **function prototyping** is used to inform the compiler about the specification of the function that will be defined and the syntax for invoking the function call such as
- Return type of the function
- Function name
- Type, name, order of arguments that are used in the function (Here arguments are optional)

The general format or syntax of function declaration is given below:

r**eturn_type function_name(type param_name1,type param_name2,……type param_nameN);**

Where parameters/arguments specify the type of data (primitive/user-defined) and each of these arguments should be separated with comma. The function declaration is done above the **main function.**

**Example of function declaration:**

```
float mul(float , float);
```

In the above example the return type is float and the name of the function is mul and the arguments that should be passed to the function definition through function call should be of type float.

**Function call:** The function definition is invoked by referring the function name along with arguments if any exists. The arguments in the function call if exists will take data or variables declared for its type matching to the function declaration and definition.

The arguments in the function are called as **actual** arguments.

The function call should be made always within another function.

The general format for function call which returns value is given below:

```
Variable=functionname(arguments);
                    Or
printf("format string",functionname(arguments));
```

Where arguments may or may not exist.

The general format for function call which does not return value is given below:

```
Functionname(arguments);
```

Where arguments may or may not exist.

**Example of function calls which returns values:**

```
main( )
{
   float y;
    y = mul(23.4,12.5);   /* function call */
    printf("product of two numbers is %f\n",y);
}
```

**Example of function call which does not return value:**

```
main( )
{
          mul( );   /* function call */
}
```

**/* complete program showing the usage of function declaration, function call and definition */**

```
#include <stdio.h>

float mul(float,float);   /* function declaration */

main()
{
            float y;
             y=mul(23.4,12.5);   /* function call */
              printf("product of two numbers is %f\n",y);
  }

/* function definition */
float mul(float x, float y) /* argument declaration */
{
      float result;         /* local variable declaration */
```

```
        result=x*y;         /* executable statement */
        return result;   /* returns the computed value in result to the calling function */
}
```

**Comments:**

In the above program a function is declared by name mul that takes two arguments of type float and returns a value of type float which is advance intimation to the compiler about the function that will be used in this program.

In the main function, function call is invoked by passing 23.4 and 12.5 as arguments of type float and is assigned to y as the function returns a value of type float. (The function call should be assigned to a variable if it returns a value and that variable should be declared for a type matching to the return type of that function specified in the function declaration.)

The function call leads to assigning the values 23.4 and 12.5 to arguments in the function definition i.e. x and y and these arguments should match with the data type of the function declaration. In the function body the assignment statement is executed by computing the product of x and y and assign the value to the variable result which is locally declared in the function body and the return statement transfers the value of the variable result to the calling function and assigns the value to y.

**Input Arguments or Actual Arguments:**

These are arguments that are passed from the function call to the function definition.

**Formal Arguments:**

These are arguments in the function definition that receive the value or arguments from the function call.

**Output Arguments:**

These are arguments that return value to the calling function.

**Categories of Functions** (Based on the arguments and return type)

Functions fall into four categories as listed below:

1. Function with Arguments and with return value.
2. Function with Arguments and no return value.
3. Function with no arguments and with return value.
4. Function with  no arguments and no return value.

Below we will see the comparative study of each of these categories of functions:

| **Function with Arguments and with return value** | **Function with arguments and no return value** | **Function with no arguments and with return value** | **Function with no arguments and no return value** |
|---|---|---|---|
| **Description:**<br><br>**Function call:**<br>• Function call is made by passing arguments<br>• The function call is assigned to a variable that matches to the return type of the function definition or used in the printf function with format specifier matching to the return type of the function definition.<br>**Function Definition:** | **Description:**<br><br>**Function call:**<br>• Function call is made by passing arguments<br>• Function call is not assigned to any variable or not used in printf function.<br>**Function Definition:**<br>• The number of arguments in the function definition should match with the number of arguments passed from the | **Description:**<br><br>**Function call:**<br>• Function call is made by passing no arguments<br>• The function call is assigned to a variable that matches to the return type of the function definition or used in the printf function with format specifier matching to the return type of the function definition. | **Description:**<br><br>**Function call:**<br>• Function call is made by passing no arguments<br>• Function call is not assigned to any variable or not used in printf function.<br>**Function Definition:**<br>• The function definition will not receive any arguments. |

| | | | |
|---|---|---|---|
| • The number of arguments in the function definition should match with the number of arguments passed from the function call.<br>• The return value should match with the return type of the function definition.<br><br>Below shows an example of how the programming instruction changes for this category of function.<br><br>#include &lt;stdio.h&gt;<br><br>/* function declaration*/<br>int printvalue(int);<br><br>main( )<br>{<br><br>int value;<br>clrscr( );<br>/* function call */<br>value=printvalue(10);<br>printf("%d",value);<br>}<br><br>/* function definition */<br>int printvalue(int a)<br>{<br>return a;<br>} | function call.<br>• The return statement will not return any value.<br><br>Below shows an example of how the programming instruction changes for this category of function.<br><br>#include &lt;stdio.h&gt;<br><br>/* function declaration*/<br>void printvalue(int);<br><br>main()<br>{<br><br>clrscr();<br>/* function call */<br>printvalue(10);<br>}<br><br>/* function definition */<br>void printvalue(int a)<br>{<br>printf("%d",a);<br>return 0;<br>} | **Function Definition:**<br>• The function definition will not receive any arguments.<br><br>• The return value should match with the return type of the function definition.<br>Below shows an example of how the programming instruction changes for this category of function.<br><br>#include &lt;stdio.h&gt;<br><br>/* function declaration*/<br>int printvalue(void);<br><br>main( x)<br>{<br><br>int value;<br>clrscr();<br>/* function call */<br>value=printvalue();<br>printf("%d",value);<br>}<br><br>/* function definition */<br>int printvalue(void)<br>{<br>int a;<br>a=10;<br>return a;<br>} | • The return statement will not return any value.<br><br>Below shows an example of how the programming instruction changes for this category of function.<br><br>#include &lt;stdio.h&gt;<br><br>/* function declaration*/<br>void printvalue(void);<br><br>main()<br>{<br><br>clrscr();<br>/* function call */<br>printvalue();<br>}<br><br>/* function definition */<br>void printvalue(void)<br>{<br>int a;<br>a=10;<br>printf("%d",a);<br>return 0;<br>} |

**Function with two arguments and no return value:**

Below is given an example showing how two arguments are passed from the function call to the function definition and the function definition does not return any value.

```
/* program to swap two numbers */
#include <stdio.h>

/* function declaration */
void swap(int,int);

main()
{
int a,b;
clrscr();
printf("Enter the values of a and b:\n");
scanf("%d %d",&a,&b);
/* function call with arguments and no return value */
swap(a,b);

printf(“a=%d b=%d”,a,b);
}



/* function definition with arguments and no return value */

void swap(int a1, int b1)
{
    int temp;
    temp=a1;
    a1=b1;
    b1=temp;
    return 0;
}
```

**Explanation:**

In the above program though we have written the logic to interchange two numbers , the two numbers will be interchanged only in the function definition and the interchanged values is not reflected in the calling program.

**Function with two arguments and multiple return values:**

Below is give an example showing how two arguments are passed from the function call to the function definition and the function definition returns multiple values.

In this example we pass the address of the arguments from the function call and receive in pointer arguments in function definition and as a result the changes made are automatically reflected to the variables/arguments in the calling program.

```
/* program to swap two numbers */
#include <stdio.h>

/* function declaration */
void swap(int *,int *);

main()
{
int a,b;
clrscr();
printf(“Enter the values of a and b:\n”);
scanf(“%d %d”,&a,&b);

/* function call with arguments and no return value */
swap(&a,&b);

printf(“a=%d b=%d”,a,b);
}


/* function definition with arguments and no return value */

void swap(int *a1, int *b1)
{
    int *temp;
    *temp=*a1;
    *a1=*b1;
    *b1=*temp;
}
```

**Explanation:**

In the above program the logic that we have written to interchange two numbers, the two numbers will be interchanged in the calling program.

**Scope:**

It determines the part of the program where the variable or a function is visible or available for use. There are two scopes namely

- Global scope
- Local scope

**Global Scope:**

Any variable or function declared above the main function is visible or available from its point of declaration to the end of the program.

**Local Scope:**

Any variable or function declared within a **block** is visible or available only within that block.

**Storage class:**

Storage class defines the following information for a variable.

- Location of the variable where it is stored
- Initial value of the variable
- Scope of the variable
- Lifetime of the variable

There are four storage classes supported in C Language. They are

1. Automatic storage class
2. External storage class
3. Static storage class
4. Register storage class.

| **Automatic** | **External** | **Static** | **Register** |
|---|---|---|---|
| • It is stored in the memory of the computer.<br>• The default value for a variable declared with this storage class is **garbage value.**<br>• Scope of the variable is within the block where it is declared.<br>• It is active and alive only in that block.<br>• The key word **auto** is used to declare the variable as automatic | • It is stored in the memory of the computer.<br>• The default value for a variable declared with this storage class is **0(zero).**<br>• Scope of the variable is global.<br>• It is active and alive until the end of the program.<br>• The key word **extern** is used to declare the variable as external storage class.<br>• By default all declarations assume as automatic storage class | • It is stored in the memory of the computer.<br>• The default value for a variable declared with this storage class is **0**(zero).<br>• Scope of the variable is within the block where it is declared.<br>• It is active and alive only in that block.<br>• The key word **static** is used to declare the variable as static storage class. | • it is stored in the CPU registers.<br>• The default value for a variable declared with this storage class is **garbage value.**<br>• Scope of the variable is within the block where it is declared.<br>• It is active and alive only in that block.<br>• The key word **register** is used to declare the variable as register storage class.<br>• By default all declarations assume as automatic storage class in the absence of the keyword register. |

| | | | |
|---|---|---|---|
| storage class.<br>• By default all declarations assume as automatic storage class in the absence of the keyword auto.<br>• E.g.,<br>auto int i;<br><br>main()<br>{<br>auto int a;<br>printf("%d",a);<br>}<br><br>**Explanation:** garbage value will be printed for the variable a. | in the absence of the keyword extern.<br>• E.g.,<br>extern int i;<br>main()<br>{<br>extern int a;<br>printf("%d",a);<br>}<br>int a=20;<br>f( )<br>{<br>}<br>**Note:** if any variable is declared after the main function and is to be accessed in the main function then that variable can be declared by preceeding with the keyword **extern** in the main function so that the reference is made to the original declaration of that variable and no additional memory is allocated. | • By default all declarations assume as automatic storage class in the absence of the keyword static.<br>• E.g.,<br>static int i;<br><br>main()<br>{<br>static int a;<br>printf("%d",a);<br>}<br><br>**Explanation:** zero will be printed for the variable a. | • E.g.,<br>register int i;<br><br>main( )<br>{<br>register int a;<br>printf("%d",a);<br>}<br><br>**Explanation:** garbage value will be printed for the variable a. |

**Type Qualifiers:**

C provides three types of qualifiers namely:

1. Constant variable
2. Volatile variable

**Constant Variable:**

To make the value of the variable remain unchanged during program execution. The variable is declared as constant. It is done by using the keyword **const** before the declaration.

e.g., const int x=10;

**Volatile Variable:**

To change the value of the variable at any time by external programs or the same program. The variable is declared as volatile. It is done by using the keyword **volatile** before the declaration.

e.g., volatile int x;

**Command line arguments in C:** main() function of a C program accepts arguments from command line or from other shell scripts by following commands. They are,

- argc
- argv[ ]

where,

argc -Number of arguments in the command line including program name
argv[ ] – This is carrying all the arguments

- In real time application, it will happen to pass arguments to the main program itself. These arguments are passed to the main () function while executing binary file from command line.
- For example, when we compile a program (test.c), we get executable file in the name "test".
- Now, we run the executable "test" along with 4 arguments in command line like below.

**./test this is a program**

Where,

| | | |
|---|---|---|
| argc | = | 5 |
| argv[0] | = | "test" |
| argv[1] | = | "this" |

```
argv[2]     =     “is”
argv[3]     =     “a”
argv[4]     =     “program”
argv[5]     =     NULL
```

**Example program for argc() and argv() functions in C:**

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])   //  command line arguments
{
if(argc!=5)
{
  printf("Arguments passed through command line " \
       "not equal to 5");
  return 1;
}

  printf("\n Program name  : %s \n", argv[0]);
  printf("1st arg  : %s \n", argv[1]);
  printf("2nd arg  : %s \n", argv[2]);
  printf("3rd arg  : %s \n", argv[3]);
  printf("4th arg  : %s \n", argv[4]);
  printf("5th arg  : %s \n", argv[5]);

return 0;
}
```

**Output:**

```
Program name : test
1st     arg    :   this
2nd      arg     :    is
3rd      arg     :     a
4th  arg  :  program
5th arg : (null)
```

<u>**C – Variable length argument**</u>

- Variable length arguments is an advanced concept in C language offered by c99 standard. In c89 standard, fixed arguments only can be passed to the functions.
- When a function gets number of arguments that changes at run time, we can go for variable length arguments.
- It is denoted as … (3 dots)
- stdarg.h header file should be included to make use of variable length argument functions.

**Example program for variable length arguments in C:**

```
#include <stdio.h>
#include <stdarg.h>

int add(int num,...);

int main()
{
   printf("The value from first function call = " \
        "%d\n", add(2,2,3));
   printf("The value from second function call= " \
        "%d \n", add(4,2,3,4,5));

   /*Note - In function add(2,2,3),
              first 2 is total number of arguments
              2,3 are variable length arguments
          In function add(4,2,3,4,5),
              4 is total number of arguments
              2,3,4,5 are variable length arguments
    */

    return 0;
}

int add(int num,...)
{
    va_list valist;
    int sum = 0;
    int i;

    va_start(valist, num);
    for (i = 0; i < num; i++)
    {
       sum + = va_arg(valist, int);
    }
    va_end(valist);
    return sum;
}
```

**Output:**

```
The value from first function call = 5
The value from second function call= 14
```

In the above program, function “add” is called twice. But, number of arguments passed to the function gets varies for each. So, 3 dots (…) are mentioned for function ‘add” that indicates that this function will get any number of arguments at run time.

**Return statement:** Every function body ends with return statement. This statement is used to return a value to the calling function.

If the function does not return any value then the return statement is written as

**Return;** or **return 0;**

If the function returns any value then the return statement is written as

**Return value;** or **return expression;**

- Always, Only one value can be returned from a function.
- If you try to return more than one values from a function, only one value will be returned that appears at the right most place of the return statement.
- For example, if you use “return a,b,c” in your function, value for c only will be returned and values a, b won’t be returned to the program.
- In case, if you want to return more than one values, pointers can be used to directly change the values in address instead of returning those values to the function.

## Recursion

Recursion is the process of repeating items in a self-similar way.

Same applies in programming languages as well where if a programming allows you to call a function inside the same function that is called recursive call of the function as follows.

```
void recursion()
{
  recursion(); /* function calls itself */
}

int main()
{
  recursion();
}
```

The C programming language supports recursion, i.e., a function to call itself.

But while using recursion, programmers need to be careful to define an exit condition from the function , otherwise it will go in infinite loop.
Recursive functions are very useful to solve many mathematical problems like to calculate factorial of a number, generating Fibonacci series, etc.

**Examples:**

**Number Factorial**
Following is an example, which calculates factorial for a given number using a recursive function:

```
#include <stdio.h>

int factorial(unsigned int i)
{
  if(i <= 1)
  {
    return 1;
  }
  return i * factorial(i - 1);
}
int  main()
{
   int i = 15;
   printf("Factorial of %d is %d\n", i, factorial(i));
   return 0;
}
```

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

```
Factorial of 15 is 2004310016
```

**Fibonacci Series**
Following is another example, which generates Fibonacci series for a given number using a recursive function:

```
#include <stdio.h>

int fibonaci(int i)
{
  if(i == 0)
  {
    return 0;
  }
  if(i == 1)
  {
    return 1;
  }
  return fibonaci(i-1) + fibonaci(i-2);
}

int  main()
{
   int i;
   for (i = 0; i < 10; i++)
   {
     printf("%d\t%n", fibonaci(i));
   }
   return 0;
}
```

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

```
0	1	1	2	3	5	8	13	21	34
```