Unit 7: Functions

7.1 Library Functions,
7.2 User defined functions,
7.3 Function prototype, Function call, and Function Definition,
7.4 Nested and Recursive Function,
7.5 Function Arguments and Return Types,
7.6 Passing Arrays to Function,
7.7 Passing Strings to Function,
7.8 Passing Arguments by Value, Passing Arguments by Address,
7.9 Scope visibility and lifetime of a variable, Local and Global Variable

Function

Definition:

- **A function is a named, independent section of C code that performs a specific task and optionally returns a value to the calling program.**
- **A function is named :-**
  - **Each function has a unique name. By using that name in another part of the program, we can execute the statements contained in the function. This is known as calling the function.**
  - **A function can be called from another function.**

- **A function is independent:**
  - **A function can perform its task without interference from or interfering with other parts of the program.**
- **A function performs a specific task:**
  - **This is the easy part of the definition.**
  - **A task is a discrete job that our program must perform as part of its overall operation, e.g. sorting an array into numerical order, or calculating a cube root.**
- **A function can return a value to the calling program:**
  - **When our program calls a function, the statements it contains are executed. If we want them to, these statements can pass information back to the calling program**

How a Function Works

- A C program doesn't execute the statements in a function until the function is called by another part of the program.
- When a function is called, the program can send the function information in the form of one or more arguments.
- An argument is program data needed by the function to perform its task. The statements in the function then execute, performing whatever task each was designed to do.
- When the function's statements have finished, execution passes back to the same location in the program that called the function.
- Functions can send information back to the program in the form of a return value.

Types of function

- **Function are of two types**
  - **Pre-defined function (Library Function)**
  - **User-defined function.**
- **Pre-defined functions** are C library function which are simply called providing necessary arguments, if any, by the programmer to do some specific task as defined by that function.
  - Example, printf(), scanf(), gets(), puts(), sqrt(), strlen() etc.
- The programmer must include the proper header file for the execution of such function.
- **User-defined functions** are the functions that are written by the programmers themselves and form the part of the source code.
- Those are compiled with other functions.

The components of a function:

- There are three components of the functions: **return type, name and argument lists.**
- **Return type** indicates the type of data that the function returns to the calling program.
- **Function name** is the unique name(identifier) given to the function. In the naming of function the same rule is applied as variable naming.
- **Argument list** is the list of the parameter to which arguments(program data) are

Process of Defining functions

1. Function Prototype:

- The function to be defined must have prototype**.**
- **A function prototype** is the declaration of the function to be defined y the user which provides the compiler with a description of a function that will be defined and used at a later point in the program. It should be before the call of the function.
- In C, function prototype is written before main() for global function and immediately after main() for local function.
- A prototype should always end with a semicolon.
- The syntax:

return_type function_name(arg_type1 ,arg_type2,….,arg_typeN);

- e.g. The prototype of the function to add two integer number and return the value as integer is

int sum(int, int);

Process of Defining functions

2. Function Call:

- Calling or invoking the function locates the function in the memory, furnishing it with arguments and causing it to execute.
- When a function is called then the control passed to the function where it is actually defined,the actually statements are executed and control passed again to the calling program.

- Call Syntax:
  - **variable= function_name(arg1,arg2,…,argN); for a function that returns a value.**
  - **function_name(arg1,arg2,…,argN); for a function without return type**
- The function prototyped above is called as:

  e.g. ***result= sum(num1,num2);***

- This calls the function sum with two integer arguments num1 and num2 and the value returned after executing the sum is assigned to the result.

Process of Defining functions

3. Function Definition:

- A function definition is the actual task of the function.
- The definition contains the statement(s) that will be executed to perform the task specified by the function itself.
- The function body should start with an opening bracket and end with a closing bracket { }.
- If the function does not returns any value the keyword void is used as return type.
- **The syntax:**

```
return_type          function_name( arg-type name-1,...,arg-type name-n)
{  /* statements; */

}
```

Example-1

```
#include<stdio.h>
int sum(int , int );  /* notice the semicolon */
main()
{
        int  x=20, y=30, s;
        s=sum(x,y);
        printf("Sum= %d", s);
        return 0;
}
int  sum(int a, int b)
{
        int total;
        total = a+ b;
        return total;
}
```

Example : 2

```
/* A program that calculate the factorial of an integer using function */
#include<stdio.h>
unsigned long factorial( int ); /* prototype  */
void main()
{
        int n;
        unsigned long fact =1;
        printf("\nInput an positive integer:");
        scanf("%d",&n);
        fact  =  factorial(n);    /* Calling  function */
        printf("\nThe factorial of %d %lu" ,n, fact);
}
/* function definition*/
unsigned  long factorial(int n)  /* function header */
{
        unsigned int fact =1;  /*local variable */
        int i;
        for(i=1,i<=n; i++)
           fact *= i;
        return fact;      /*returns the factorial value */
}
```

Example- 3

```
#include<stdio.h>
int square( int );    / * prototype */
main()
{
        int n, sq;
        printf("Enter a number: ");
        scanf("%d" , &n);
        sq = square(n);     /* function call */
        printf("\n Square of %d is %d", n,sq);
        return 0;
}
int  square( int x )      /* Function Definition*/
{
        return x * x;
}
```

Nested Function

- Some programmer thinks that defining a function inside an another function is known as “nested function”. But the reality is that it is not a nested function, it is treated as lexical scoping.
- Lexical scoping is not valid in C because the compiler cant reach/find the correct memory location of the inner function.

- Nested function **is not supported** by C because we cannot define a function within another function in C. We can declare a function inside a function, but it’s not a nested function.
- If we try to approach nested function in C, then we will get compile time error.

Example: Nested Function

```
/* C program to illustrate the concept of Nested
function. */
#include<stdio.h>
void fun()
{
	printf("\nFun");
	void view()
	{
		printf("\nView");
	}
}
main()
{
	printf("Main");
	view();
}
```

Output:
Compile time error: undefined reference to `view'

Recursive Function:

- When a function calls itself , it is called recursion and the function is called recursive function.

```
function1()
{
        function1();
}
```

- If a solution of a problem is represented by the same type of smaller problem, then recursive function is used.
- For example, if we have to calculate the factorial of an integer N, then we can define the solution as
  - N ! = N * (N-1) ! For N >0
  - N! = 1 for N = 1.
- For the solution of this problem, we can define a factorial function as recursive function.

Example of recursive function

```
unsigned long factorial( int n)
{
        If(n==0)
                return  1;
        else
                return  n* factorial(n-1);
}
```

```
/* calculate the factorial of an integer using recursion */
#include<stdio.h>
unsigned long factorial(int);
int main()
{
        int n;
        unsigned long fact;
        printf("Enter an integer:");
        scanf("%d",&n);
        fact=factorial(n);
        printf("\n The Factorial of %d is %lu",n,fact);
        return 0;
}
unsigned long factorial( int n)
{
        if(n==0)
                return  1;
        else
                return  n* factorial(n-1);
}
```

Recursive algorithm for a^b

- We can define a^b as;
  - If b==0, then $a^b = 1$
  - if b>0 then $a^b = a * a^{b-1}$
- So recursive function for a^b is,

```
int power(int a,int b)
{
   if (b==0)
      Return 1;
   else  return a * power(a,b-1);
}
```

```
/* Program using Recursive function to calculate a raised to power
 where a,b are integers*/
#include<stdio.h>
int power(int,int);
int main()
{
        int a,b,exp;
        printf("Enter Base a:");
        scanf("%d",&a);
        printf("\n Enter Exponent b:");
        scanf("%d",&b);
        exp=power(a,b);
        printf("\n %d Raised to Power %d  = %d",a,b,exp);
        return 0;
}
int power(int a,int b)
{
        if(b==0)
                return  1;
        else
                return  a*power(a,b-1);
}
```

Passing **Array/String** to Function

- A function can have arrays as its arguments.
- For passing array, we must pass the address of the array on function call.
- Since array name itself is the address of the array(address of first element) , the name of array is passed to the function as argument which actually passes the address of array, not the actual array data.
- The syntax for passing array as function argument is:
  - **Return_type function_name(data_type []) ; /* Prototype */**
  - **Function_name(array_name) ; /* in function call */**
  - **Return_type function_name(data_type arg_name[])**

    {
    - **/* Actual Definition of the function */**

    }

Passing **Array/String** to Function

- Let us take an example that uses a function to sum the array element having array as argument.

```
#include<stdio.h>
int  addArray(int[], int);
int main()
{
        int a[100],n,i,sum;
        printf("Enter no of element in array a[]:");
        scanf("%d",&n);
        printf("\nEnter %d integers in array a[]:\n",n);
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        sum=addArray(a,n);
        printf("\nThe sum of array is: %d", sum);
        return 0;
}
```

Passing **Array/String** to Function

```
int  addArray(int a[], int n)
{
        int i, sum=0;
        for(i=0;i<n;i++)
                sum+=a[i];
        return sum;
}
```

The output:

```
Enter no of element in array a[]:5

Enter 5 integers in array a[]:
10 20 30 40 50

The sum of array is: 150
```

Passing **Array/String** to Function

- A string is also an array of characters.
- Similar to above example, we can pass string into function similarly.(Later we can use pointer for array and string)
- Let us take an example of a function passing string as an argument below.

```
#include<stdio.h>
int strLength(char[]);
int main()
{
        char str[100];
        int len;
        printf("Enter a string:");
        scanf("%s",str);
        len=strLength(str);
        printf("\nLength of Srting %s is : %d",str,len);
        return 0;
}
```

Passing **Array/String** to Function

```
/*Function  to find length of string */
int strLength(char s[])
{
        int i,length=0;
        for(i=0;s[i]!='\0';i++)
                length++;
        return length;
}
```

Output:

Enter a string:Kathmandu

Length of Srting Kathmandu is :9

Passing argument to Function

- In C, argument to the function can be passed in two ways.
  - Passing by value
  - Passing by address.
- In **passing by value**, the actual value is passed to the function on calling (value variable or constant)
- The value passed to the function call is copied to the actual argument variable defined in the function definition, those are local variable in the function definition.
- In **passing by address**, the address of variable is passed to the function and the function argument to be defined as pointer variable which can take the address of variable rather than value.
- When we have to access the actual variable of calling program from another function, address must be passed, otherwise we can use passing by value.

Passing argument to Function

- Let us take an example that swap the two number using function.

```
#include<stdio.h>
void swap(int,int);  /* Passing by value */
int main()
{
        int x=10,y=20;
        printf("Before swap, x= %d, y= %d",x,y);
        swap(x,y);
        printf("\nIn main(), after swap: x=%d,y=%d",x,y);
        return 0;
}
```

Passing by Value

```
void swap(int x,int y)
{
        int temp;
        temp=x,x=y,y=temp;
        printf("\nIn function Swap(): x=%d,y=%d",x,y);
}

/*Output:
Before swap, x= 10, y= 20
In function Swap(): x=20,y=10
In main(), after swap: x=10,y=20
*/
```

Passing by Address

- For passing by address, we use pointer in function argument (pointer will be discribed in next chapter)

```
#include<stdio.h>
void swap(int *,int *);  /* Passing by address */
int main()
{
        int x=10,y=20;
        printf("Before swap, x= %d, y= %d",x,y);
        swap(&x,&y);  /* passing address*/
        printf("\nIn main(), after swap: x=%d,y=%d",x,y);
        return 0;
}
```

Passing by Address

```
void swap(int *x,int *y)
{
        int temp;
        temp=*x,*x=*y,*y=temp;
        printf("\nIn function Swap(): x=%d,y=%d",*x,*y);
}

/*
Output:
Before swap, x= 10, y= 20
In function Swap(): x=20,y=10
In main(), after swap: x=20,y=10
*/
```

Scope visibility and lifetime of a variable

- **The scope** of a variable refers that to which different parts of a program have access to the variable. In other words, where the variable is visible.
- When speaking about scope, the term variable refers to all C data types: simple variables, arrays, structures, pointers, and so forth.
- It also refers to symbolic constants defined with the const keyword.
- **The life time** of variable refers that how long the variable persists in memory, or when the variable's storage is allocated and de-allocated.
- Depending upon the scope and lifetime of variable, C has its four storage classes for variables used in any functions.

  1.Automatic variables 2. External variables

  3. Static variables 4. Register variables

Local and Global variable

- The variables may broadly categorized , depending upon the place of their declaration as: internal(local) or External(global).
- The internal(local) variables are those which are declared inside a local scope i.e inside a function.
- The external(global) variables are those which are declared outside the function , generally before main() function.
- e.g.,

```
int  x;   / * Global variable */
main()
{
        int y;   /* Local variable */
        …
}
```

Automatic variables

- An automatic variable is defined inside the main() or function program. It is created when the function is called and get destroyed when the function segment is exited.
- Thus the visibility and lifetime of automatic variable is limited to the function execution. Generally called Local variables.
- The key word 'auto' is used to declare the automatic variable. It is declared as:
- auto data_type variable_name;
- Generally the keyword auto maybe dropped out. The variable declared inside function are default automatic if not any other keywords are used.
- These are also called local variables.
- e.g.

```
main() {
        int num;        /* num is automatic variable */
}
……..
```

External variable

- The external variables are declared outside the main or function block.
- These variables are set up memory immediately on the declaration and remain active for the entire period of program execution.
- Its visibility(scope) is that of the source file.
- If they are not initialized then they are automatically initialized to zero values.
- External variables are accessed by all the functions in the program. These variables are also known as global variables.
- The keyword ‘extern’ is used to declare these variables. But it is optional. We can use extern to avoid confusion.

```
extern  data_type  variable_name;     /* external variable */
```

- e.g.

```
int sum,difference;
char name[10];
main()
{ …..}
```

Static variable

- Static variables are generally defined inside main() or function block with keyword ‘static’ prefixed to it.
- They are sometimes called static auto variables as they have visibility of local (or auto) variables and lifetime of external (or global) variables.
- Static variables are initialized once during the first function call. These variables are declared as

  static data_type variable_name;

e.g. static int counter;

Register variables

- If we want to tell the compiler that a variable should be kept in one of the machine's register instead of keeping in the memory(where normal variables are stored), variables should be declared as register.
- The register access is much faster than memory. So keeping frequently access variables in register makes the execution of program faster.
- The declaraction is done as:

  register data_type variable_name;

  – e.g. **register int count;**
- Only a few variables can be stored in register.
- However C compiler converts register variables into non register variables once the limit is reached.