




Unit –IV

Functions

Definition: A Function is a set of instructions that are executed wherever an event occurs. A Function perform a specific task. These functions are also called as “ *user defined functions*”.

In order to make use of a user defined function. We need to establish 3 elements that are related to functions.

-  Function definition
-  Function call
-  Function declaration

Function Definition: A function definition also known as a function implementation. Function implementation shall includes

- Function header
- Function body

A general format of a function definition is as follows

function type function name(parameter list)

```
{
    local variable;
    statement 1;
    statement 2;
    .
    .
    return statement;
}
```

Function header: A function header consists of 3 parts. The function type also known as return type like int, float, char etc., the function name and formal parameter list.

Function body: the function body contains the declaration and statement necessary for performing the task. The body is enclosed in braces. It contains

- Local variable
- Function statements
- A return statement to return the result

Function Call: A function can be called by simply using the function name followed by a list of actual parameter.

```
Ex:    main( )
      {
          int y;
          Y=add(10,5);/*function call*/
          Printf(“%d”,y);
      }
```

When the compiler encounters a function call, the control is transferred to the function add()

```
main( )
{
    int y;
    y=add(10,5);
    printf(“%d”,y);
}

int add(int x, int y)
{
    int p;
    p=x+y;/*local variable*/
    return p;
}
```

Function declaration: like variable all function in a C program must be declared, before they are included. It contains

Function type function name(parameter list);

The above statement is also called function prototype.

```
Ex: int mul(int, int);
    float add(float, float);
    float sub(float, float);
```

A proto type declaration may be placed in two places.

1. Above all the functions
2. Inside a function definition.

Parameter passing: An argument passing is a simple process of transferring data from calling function to the called function.

There are 2 techniques of passing arguments in C Language. These are

1. Call from value
2. Call from reference/address.

1. **Call from value:** This is the default method of argument passing in which a copy of actual argument is made and is passed to the newly created formal arguments. It contains only the copy of actual arguments which are stored at separate memory locations; therefore, any changes made to these are not reflected back to the actual arguments. One more thing, the changes made to the formal arguments are local to the block in which they are made; once the control is shifted back to the calling function, they are lost.

Syntax:

Function definition:

```
return type function name(parameter list)
{
    statements;
    .....
    .....
    return
}
```

Function declaration:

```
return type function name(parameter list);
```

Function calling:

```
var=function name(arguments);
```

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

```
void main()
```

```
{
```

```
    void swap(int, int)
```

```
    int a=10,b=20;
```

```
    printf("before swap a=%d, b=%d", a,b);
```

```
    swap(a,b);
```

```
    printf("after swap a=%d,b=%d",a,b);
```

```
}
```

```
void swap(int x,int y)
```

```
{
```

```
    int t;
```

```
    t=x;
```

```
    x=y;
```

```
    y=t;
```

```
}
```

Call from reference / address: This is another technique of passing arguments but instead of passing the value we pass the address of those arguments. If a program wants to reflect back, the modifications made in formal arguments to the actual method for argument passing.

In this we use & operator with the name of the variable to send the address of that variable, and to receive the address of arguments, special variables are used called pointer variables.

Function Definition:

```
Return type function name(int var1, int var2);
```

```
{
```

```
    Statements
```

```
    Return
```

```
}
```

Function Declaration:

```
Return type function name(int *, int*);
```

Function Call:

```
Name(&var1, &var2);
```

Here *var1, *var2 are pointer variables &var1, &var2 are the addresses of those variables.

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

```
{
```

```
    void swap(int *, int*);
```

```
    int a=10,b=20;
```

```
    printf("before swap a=%d,b=%d",a,b);
```

```
    swap(&a,&b);
```

```
    printf("after swap a=%d,b=%d",a,b);
```

```

}
void swap(int *x, int *y)
{
    int t;
    t=*x;
    *x=*y;
    *y=t;
}

```

Recursive Function: When a called function in turn calls another function a process of chaining occurs recursive is a special case of this process. Where a function calls itself. A simple example of recursion is

```

Main( )
{
    Printf("example of recursion");
    Main();
}

```

When executed the program will produce an output something like this

```

Example of recursion
Example of recursion
Example of recursion

```

The execution will continuous infinitely.

Another useful example of recursion is the evaluation of factorial of a given number. The factorial of a number n is expressed as a series of multiplication as follows

$N=n(n-1)(n-2).....1$

For example

```

4=4*3*2*1=24
Fact(int n)
{
    If(n==1)
        Return 1;
    Else f=n*fact(n-1);
}

```

Let us see how the recursion works. Assume n=3 since the value of n is not 1.

```

F=n*fact(3-1);
= 3*2*fact(1)
= 3*2*1
= 6

```

Write a C Program to find factorial of a number using recursion.

```

#include<stdio.h>
Void main( )
{
    Int n;
    Printf("enter a number");
    Scanf("%d",&n);
    Printf("factorial is %d", fact(n));
}
Long int fact(int n)
{
    If(n=0)
        Return(1);
    Else
        Return n*fact(n-1);
}

```

Storage Classes

In C Language each variable has storage class which includes

- Scope:** How much of a variable is accessed in a program.
- Default initial value:** if we don't explicitly initialize the variable, what will be its default value
- Life time:** How long a variable exist

The following storage classes are most used in C Programming.

1. Automatic Variables
2. External Variables
3. Static Variables
4. Register variables

Automatic Variables:

- Scope:** Variable defined with auto storage class are local to function block.
- Default value:** it's default value is garbage value
- Life time:** till the end of the function.

A variable declared inside a function without any storage class is by default an automatic variables. They are created when a function is called and are destroyed automatically when a function execution is completed.

```
#include<stdio.h>
Void main( )
{
    Int a;
    Auto int b;
}
```

External Variable:

- Scope:** Global i.e., every where we can access in the program.
- Default value:** Zero
- Life time:** till the program completes its execution

A variable that are defined outside of a program or function is called *Global Variables*. Global variable values are accessed from anywhere in the program.

```
#include<stdio.h>
Int a;
Extern int b;
Void main( )
{
    .....
    .....
}
```

Static Variable:

- Scope:** local to the block in which the variable is defined.
- Default Value:** Zero
- Life time:** till the whole program completes its execution.

A static variable tells the compiler to save the variable until the end of program. Instead of creating & destroying a variable everything, static variable create once but remains till program completes its execution.

```
#include<stdio.h>
Void main( )
{
    Static int a=10;
    .....
    .....
}
```

Register Variables:

- Scope:** Local to the function
- Default value:** Garbage Value
- Life time:** till end of the function

Register variables tell the compiler to store variable in CPU register instead of memory i.e., RAM. But only few variables can be stored inside register. We can access register variable faster.

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

```

Void main( )
{
    Register int a;
    .....
    .....
}

```

Scope of variable in C

A scope of variable defined how much of a variable is accessed in a program. There are 3 places where variable can be declared in C program language. They are

1. Local Variables
2. Global Variables
3. Formal parameter

1. **Local Variables:** The variable that are declared inside a function or block are called local variables. That are defined inside that function.

```

#include<stdio.h>
Void main( )
{
    Int a=10,b=20,c;
    C=a+b;/*local variables*/
    Print("sum is %d",c);
}

```

2. **Global Variables:** The variables that are declared out side a function or block are called *Global Variables*. They can be used everywhere in the program. The life time of global variable is entire program.

```

#include<stdio.h>
Int c;/*Global variable*/
Void main( )
{
    Int a=10,b=20;
    C=a+b;
    Printf("sum %d",c);
}

```

3. **Formal Parameter:** Formal parameters are treated as *local Variable* within a function.

```

#include<stdio.h>
Void main()
{
    Int c;
    C=sum(10,20);
    Printf("sum is %d",c);
}
Int sum(int x, int y)/*formal variables*/
{
    Return (x+y);
}

```

Return statement in C:

A function may or maynot return a value. A return statement returns a value to the following function. The return statement can be used in the following two ways.

1. Return
2. Return expression

1. **Return:** In this form a function doesn't return a value, the return type In the function definition and declaration is specifies a void.

```

#include<stdio.h>
Void main( )
{
    Int c;
    C=sum(10,30);
}
Void sum(int x, int y)
{
    Int z;
}

```

```
        Z=x+y;
        Printf("sum is %d",z);
        Return;
    }
```

- 2. Return expression:** In this form a function return a value to the calling function and assigns to the variable in the left side of calling function.

```
#include<stdio.h>
Void main( )
{
    Int c;
    C=sum(10,20);
    Printf ("sum is %d",c);
}
Int sum(int x, int y)
{
    Return(x+y);
}
```