UNIT-IV

FUNCTIONS

In c, we can divide a large program into the basic building blocks known as functions.

The function contains the set of programming statements enclosed by { }.

The function is also known as Procedure (or) Subroutine in other programming languages.

Advantage of functions in c :-

There are the following advantages of c functions.

→ By using functions, we can avoid rewriting same logic/code again and again in a program.

→ we can call c functions any number of times in a program and from any place in a program.

→ Reusability is the main achievement of c functions.

→ we can track a large c program easily when it is divided into multiple functions.

Types of Functions :—

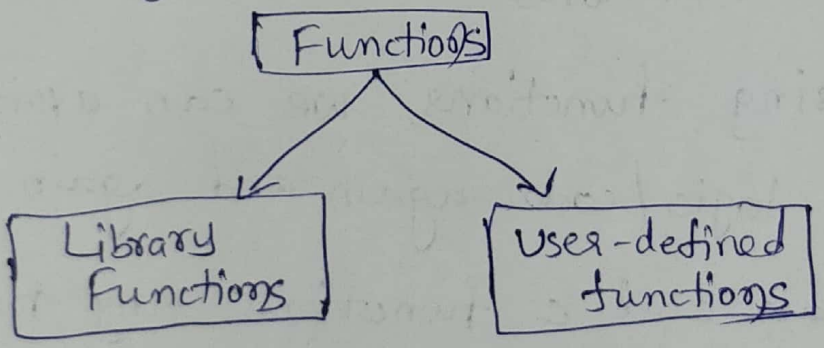
There are two types of functions in C Programming:

→ Library Functions

→ User-defined functions.

1. Library Functions: are the functions which are declared in the C header files such as scanf(), printf(), sqrt(), gets(), puts() etc.

2. User-defined Functions :— are the functions which are created by the C programmer, so that he/she can use it many times. It reduces the complexity of a big program and optimizes the code.



Function Aspects :—

There are three aspects of a C function.

→ Function declaration (function Prototype)

→ Function call

→ Function definition.

① Function declaration :—

In C, while defining user - defined functions it is must to declare its prototype.
A prototype statement helps the compiler to check the return type and argument type of the function.

→ A function declaration consists of the function's return type, function name and argument list.

→ when the programmer defines the function, the defination of the function must be same like its prototype declaration.

Syntax :— return type function-name(argument list);

Eg : float sum(float, int);
int add (int x, int y);

② Function call :—

A function can be called simply using its name followed by Parameters, terminated by Semicolon (;)

Function can be called from any where in the Program. The parameter list must not differ in function calling and function declaration.

Syntax :— function_name(argument_list);

3) Function definition :—

A function definition, also known as function implementation.

It includes the following elements:

1. function name
2. function Type
3. List of Parameters
4. local variable declarations
5. Function statements
6. Return statement.

All the six elements are grouped into two Parts, namely

→ function header (First three elements)
→ function body (second three elements)

A general format of function definition :

Syntax :—

```
returntype function-name(argument list)
{
  Local variable declaration;
  St1 ;
  St2 ;          } body of function
  Return (value);
}
```

Eg:

```
int add( int x, int y)
{
    int z;
    z = x + y;
    return(z);
}
```

Different aspects of function calling :-

A function may or may not accept any argument. It may or may not return any value. Based on these facts, There are four different aspects of function calls.

→ Function without arguments and without return value

→ function without arguments and with return value

→ function with arguments and without return value

→ function with arguments and with return value.

① no arguments and no return value :-

In this category, the function has no arguments, it does not receive any data from the calling function. Similarly, it doesn't return any value. The calling doesn't receive any data from the called function. So, there is no communication between calling and called functions.

Ex:

```
// addition of two no's
#include<stdio.h>
void main()
{
  void add();  // Function prototype
  add();  // function call
}
void add()  // function definition
{
  int a=10, b=20, c;
  c=a+b;
  Pf("\n%d", c);
}
```

o/p: 30

2) Functions with arguments & return value :-

In this category, function has no arguments and it doesn't receive any data from the calling function, but it returns a value to the calling function. The calling function receives data from the called function. So, it is one way data communication between calling and called functions.

Eg: addition of two no's using functions

```
#include<stdio.h>
void main()
{
  int result;
  int add( ); // function prototype
  result=add( ); // function call
  printf("the result is :%d\n", result);
}
int add( ) // function definition
{
  int a=10, b=20, c;
  c=a+b;
  return(c); // return statement
}
```

control back to main

o/p: 30 result is = 30

3) Function with arguments & no return value :-

In this category, function has some arguments. It receives data from the calling function, but it doesn't return a value to the calling function. The calling function doesn't receive any data from the called function. So, it is one way data communication between called and calling functions.

Eg. // addition of two no's using 3 category

```
#include<stdio.h>
void main()
{
int a=10, b=20;
  void add(int a, int b); //function declaration
    add(a,b); // function call
}
void add(int x, int y); //function definition
  {
    int c;
    c=x+y;
    Printf("sum=%.d\n", c);
  }
```

o/p : Sum = 30.

4 Function with arguments and return value :-

In this category, functions has some arguments and it receives data from the calling function. Similarly, it returns a value to the calling function. The calling function receives data from the called functions. So, it is two-way data communication between calling and called functions.

Eg: // addition of two no's using 4th category

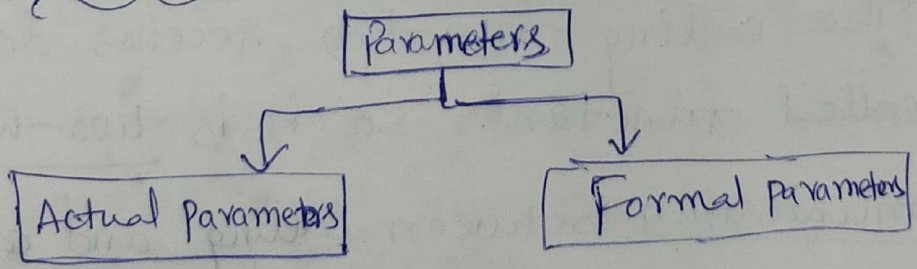
```
#include<stdio.h>
void main()
  {
  int a,b, Result;
    int add(int a,int b); //function prototype
Printf("enter the a,b values\n");
Scanf("%d%d", &a,&b);
      Result = add(a,b); // function call
  Printf("add = %d\n", Result);
   }
int add(int x, int y)  // function definition
     {
       int z;
       z = x+y;
       return(z);  // return statement
     }
```

O/P: ~~adder~~ enter the a b values

10 20

add = 30

Types of Parameters :-



Parameters are also called arguments.

In C, there are two types of parameters and they are as follows...

→ Actual parameters
→ Formal parameters

→ The actual parameters are the parameters that are specified in calling function.

→ The formal parameters are the parameters that are declared at called function. when a function gets executed, the copy of actual parameter values are copied into formal parameters.

Parameter Passing Techniques (or) Methods :—

In c programming Language, there are two methods to pass parameters from calling function to called function, They are

→ call by value (or) Pass by value
→ call by Reference (or) Pass by Reference

1) Call by value :—

In call by value parameter passing method, the copy of actual parameter values are copied to formal parameters and these formal parameters are used in called function. The changes made on the formal parameters does not effect the values of actual parameters. This means, after the execution

control comes back to the calling function, the actual parameter values remains same.

Eg:- // swaping of two numbers using call by value

```
#include<stdio.h>
Void main()
 {
   int a=10, b=20;
   Void swap(int, int); // function declaration
 Printf("before swap: a=%d, b=%d", a, b);
   swap(a, b); // calling function
 Printf("After swap: a=%d, b=%d", a, b);
 }
 Void swap(int x, int y) // called function
   {
     int temp;
     temp=x;
     x=y;
   y=temp;
   }
```

In the above example Program, the variables a and b are called actual parameters and the variables x and y are called formal parameters. The value of a is copied into x and the value of b is copied into y. The changes made on variables x and y does not effect the values of a and b.

2) Call by Reference :-

In call by reference parameter passing method, the memory location address of the actual parameters is copied to formal parameters. In this method of Parameter Passing, the formal parameters must be Pointer variables.

In call by reference parameter passing method, the address of the actual parameters is copied to formal parameters. The changes made on the formal parameters effects the values of actual parameters.

Eg:- swapping of two no's using call by reference

```
#include<stdio.h>
    void main()
      {
        int a=10, b=20;
      void swap(int *, int *);//function prototype
printf("before swap: a=%d\t, b=%d\n", a,b);
        swap(&a, &b); // calling function
printf("After swap: a=%d\t, b=%d\n", a,b);
      }
```

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

In the above example program, the addresses of variables a and b are copied to pointer variables x and y. The changes made on the pointer variables x and y in called function effects the values of actual parameters a and b in calling function.

Recursion :—

A function that calls itself is known as a recursive function. And this technique is known as Recursion.

How does recursion work ?

```
int main()
{
    ---
    ---
    recursec();
    ---
}
void recursec()
{
    ----
    recursec();
    ------
}
```

The Recursion continues until some condition is met to prevent it.

Advantages of Recursion :-

Recursion is an important concept. It is frequently used in data structure and algorithms.

Eg: Recursion is used to calculate the factorial of a number.

```
#include<stdio.h>
void main()
{
  int n, result;
  int fact(int n); // function prototype
printf("enter the n value\n");
scanf("%d", &n);
  result = fact(n); // function call
printf("factorial = %d\n", result);
}
int fact(int n) // function definition
{
  if(n==0)
    return 1;
  else
    return n*fact(n-1);
}
```

```
if n=4
return 4*fact(3)=24
return 3*fact(2)=6
return 2*fact(1)=2
return 1*fact(0)=1
```

O/P: 1*2*3*4 = 24

Storage classes in C :-

Storage classes in c are used to determine the lifetime, visibility, memory location, and initial value of a variable.

There are four types of storage classes in c.

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

! Automatic variables :-

→ Automatic variables are also referred to as local (or) internal variables.

→ Automatic variables are declared inside a function in which they are to be utilized.

→ They are created when the function is called and destroyed automatically when the function is exited.

→ Automatic variables are private (or) local to the function in which they are declared.

→ A variable declared inside a function without storage class specification is by default, an automatic variable.

→ we may also use the keyword auto to declare

automatic variables explicitly.

Eg:
```
main()
{
    auto int a, b, c;
    ---
    ---
    ---
}
```

→ The automatic variables are initialized to garbage by default.

→ Every local variable is automatic in c by default.

Eg:-
```
void main()
{
    int a, b, c; // automatic variables
    Printf("%d %d %d", a, b, c);
}
```

O/P: garbage value.

Eg 2):-
```
void main()
{
    int a=10; // method (or) function scope.
    {
        int a; // block scope
        Printf("a=%d\n", a);
    }
    Printf("a=%d\n", a);
}
```

2) External variables :-

→ external variables are also known as global variables.

→ variables that are both alive and active throughout the entire program are known as external variables.

→ unlike local variables, global variables can be accessed by any function in the program.

→ external variables are declared outside a function.

→ external variables are initialized to zero by default.

Eg 1:-
```
int a;
void main()
{
printf("%d", a);
}
```
o/p: zero (0)

Eg 2:
```
int number;
void main()
{
number=10;
Pf("I am in main function. My value is %d\n", number);
fun1(); // function calling
fun2(); // function calling
}
fun1()
{
number=20;
Pf("I am in function fun1. My value is %d", number);
```

```
fun2()
{
  Pf("I am in function fun2. My value is -/.d\n", number);
}
```

O/P: I am in function main. My value is 10
I am in function fun1. My value is 20
I am in function fun2. My value is 20.

Here the global variable number is available to all three functions and thus, if one function changes the value of the variable, it gets changed in every function.

3 Register variable:—

→ The register keyword is used to declare register variable.

→ register is used to define local variables that should be stored in a register instead of RAM.

→ we can access a register variable fasterthan a normal variable.

→ It is similar to the auto storage class.

Eg:

```
void main()
{
  register int a;
  Printf("-/.d\n", a);
}
```

O/P: garbage value.

4) Static variable :-

→ The keyword used to define static variable is static.

→ default initial value of the static variable is zero (0)

→ A same static variable can be declared many times but can be assigned at only one time.

→ Static variable is initialized only once.

→ These variables have lifetime till the end of program (throughout the program)

→ Static variable are two types
  → Internal static variable (local)
  → External static variable (global)

→ A static variable is declared inside a function is called internal static variable

→ A static variable is declared outside a function is called external static variable

Ex :-
```
void main()
{
Static int a; // static variable
Pf("%d\n", a);
}
```
O/P: zero (0)

Eg:

```
#include<stdio.h>
void main()
{
~~static~~ ~~int~~ void f1();  //function declaration
f1();
f1();
f1();
}
void f1()  //function definition.
{
~~static~~ ~~int~~ a=10;
Printf("%d\n", a);
a++;
}
```

O/P: 10
11
12

Scope :- The region of a program in which a variable is available for use

Visibility :- The program's ability to access a variable from the memory.

Lifetime :- The lifetime of a variable is the duration of time in which a variable exists in the memory during execution.

Storageclass	keyword	default value	declaration	Scope (Visibility)	Lifetime (Alive)
1) Automatic Storage class	auto	garbage	Inside a function/block	within the function/block	until the function/block completes
2) External Storage class	extern	zero	outside all functions	outside the block/function	until the program terminates
3). Register storage class	register	garbage	inside a function/block	within the function/block	until the function/block completes
4). Static storage (local) class	static	zero	inside a function/block	within the function/block	until the program terminates
Static storage (global) class	static	zero	outside a function	entire file in which it is declared	until the program terminates