Module-3

Functions (part 1) :-

Introduction
Function prototype
Types
Function definition
Function call
Function declaration
Categories (design) of function
Actual & Formal parameter
call by value & call by reference

Function :- It is a subprogram that carries out some specific & well-defined tasks.

uses of functions

1. It reduces the code size
2. Readability of the program can be increased
3. Easier debugging
4. Function can be shared by many programmers.

If the program is very big, there are many disadvantages.

- Difficult to write large program
- very difficult to understand.

So these large programs can be divided into a series of individual related programs called module. These modules are called functions.

* Types of functions :-

1) Library (Built in) function
2) user defined function (UDF)

1) Library function :-

The C library is a collection of various types of functions which perform some standard & pre-defined tasks. These functions are called as library function

Eg :- sqrt(); - To find square root
pow(); - To find power of number
printf(); - Send data to o/p unit
scanf(); - Accept data from keyboard

Advantages

These functions can be used whenever required.
The programmer's job is easy because the functions are already available.

Disadvantages

Since the library functions are limited, programmers can't completely rely on the library functions.

2) User Defined Function (UDF):

In most of the cases, the programmers need other than library functions to achieve some specific tasks.

The functions which are written by the user to do some specific tasks are called as User Defined Functions.

Eg:- If we want to add 2 matrices a & b, there is no library function in C to add 2 matrices. So we can write a function add_matrix(). So this function is written by user. So it is called user-defined function.

Eg:- program to add 2 numbers using functions.

```
#include<stdio.h>
int add(int a, int b);
void main()
{
    int a,b, c=0;
    a=10; b=20;
    c = add(a,b); // Function call
    printf("c is %d\n", c);
getch();
int add(int a, int b)
{   int result;
```

```
    result = a+b;
    return result;
}
```

In this program the function which is present inside the main() is called as calling Function.

The function which is called by the calling function is called as called function.

When the function is called, control is transfered from main() to add(). Each statement in add() is executed & addition 2 numbers is computed.

After executing last statement i.e., return result, the control will be transfered to the main() along with result from add().

The result obtained from add() is copied to variable c. Finally result will be displayed on screen.

Elements of UDF :-

1) Function declaration
2) Function definition
3) Function call.

1) Function declaration (Function prototype)

The general syntax of function declaration is

type function_name(type a1, type a2, ---, type an);

It gives the information about return type, name of function, number of parameters.

Eg:- int add (int a, int b);

Return type is integer

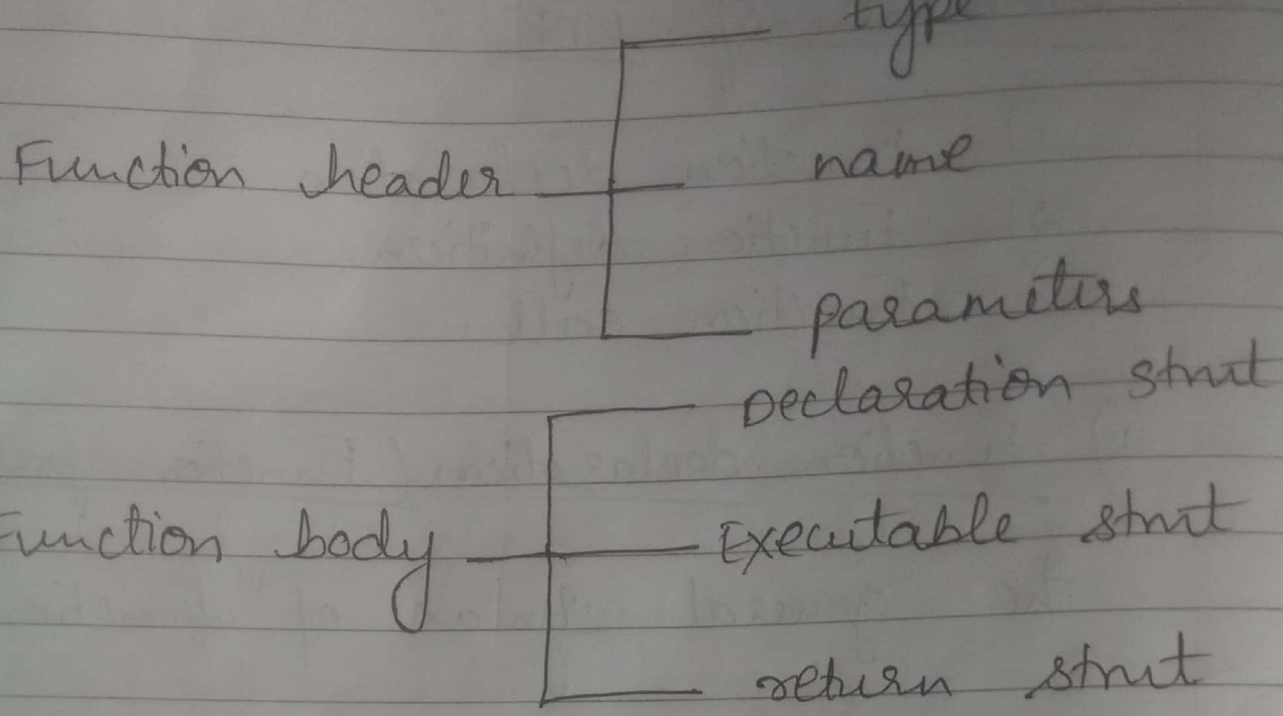
Function name is add

parameters are a & b which are of type integer.

2) Function definition :-

It consists of 2 elements -

1. Function header
2. Function body



Syntax :- type name(parameters)

```
{
    declaration part;
    executable part;
    return part;
}
```

Eg:-

```
int add(int a, int b)
{
    int sum; // Declaration
    sum = a+b; // Executable strut
    return sum; // return strut
}
```

3) Function call :-

The invokation of a function inorder to do specific task is called as function call.

```
Eg:- void main()
     {
       int m, n, res;
       m = 10;
       n = 20;
       res = max(m, n); // Function call
       printf("max = %d \n", res);
       getch();
     }
```

* Actual parameter & Formal parameter

Actual parameter	Formal parameter
1) They are used in calling function. Eg:- c = sum(a, b); Here a & b are actual parameters.	1) They are used in function header of called function. Eg:- int sum(int a, int b) Here a & b are formal parameter.
2) Actual parameter can be constants, variable or expression. Eg:- c = sum(a+4, b);	2) It should be only variable. Eg:- int sum(int a, int b)

3) It sends value to Formal paramitus.	3) It recieve values from actual paramitu
Eg:- c = sum(4,5);	Here, m will receive value 4 & n will take 5 valu
4) Addresses of this can be sent to formal paramitus	4) If formal paramitu contains addresses, they should be declared as pointers.

*** categories (Design) of function

1) Function with no paramitu & no return type
2) Function with no paramitu & return type.
3) Function with paramitu & no returntype
4) Function with paramitu & return type

1) Function with no paramitu & no return type

* In this type, there is no data transfer between calling function & called function
* The function doesnot contain paramitu
* The function doesnot return any value.

Eg:-
```
#include<stdio.h>
void add();
void main()
{
    add();
}
void add()
{
    int a, b, c;
    a=10;
    b=20;
    c=a+b;
    printf("sum=%d",c);
}
```

In this case the function add() do not receive any values from main() & doesnot return any value to main().

2) Function with no parameter & return type

* In this type, the function doesnot contain parameter, but function returns value.

Eg:-
```
#include<stdio.h>
int add();
void main()
{
    int c;
    c=add();
    printf("c=%d", c);
```

```
    getch();
}
int add()
{
    int a, b, c;
    a=10;
    b=20;
    c=a+b;
    return c;
}
```

3) Function with parameter & no return type

In this type, the function contains parameter. But it doesnot return any value.

Eg:-
```
#include<stdio.h>
void add(int a, int b);
void main()
{
    int a, b;
    a=10;
    b=20;
    add(a,b);
}
void add(int a, int b)
{
    int c;
```

```
    c = a+b;
    printf("c=%d", c);
}
```

4) Function with paramiter & return type

* In this type, function contains both paramiter & return type.

Eg:-
```
#include<stdio.h>
int add(int a, int b);
void main()
{
    int a, b, c;
    a=10;
    b=20;
    c= add(a,b);
    printf("c=%d", c);
}
int add(int a, int b)
{
    int c;
    c= a+b;
    return c;
}
```

(Passing parameters to function)

** call by value & call by reference

1) call by value :- (pass by value)

In this type, function is called with actual parameter, where values of actual parameter will be copied to formal parameter.

Eg:- program to add 2 numbers

```
#include<stdio.h>
void add(int m, int n);
void main()
{  int a, b;
   a=10; b=20;
   add(a,b);
   getch();
}
void add(int m, int n)
{  int c;
   c= m+n;
   printf("c=%d", c);
}
```

2) call by reference :- (pass by reference)

In this type, function is called with address of actual parameter, where addresses of actual parameters are passed

to formal parameter

Eg:-

```
#include<stdio.h>
void add(int *m, int *n);
void main()
{
    int a,b;
    a=10; b=20;
    add(&a,&b);
}
void add(int *m, int *n)
{
    int c;
    c = *m + *n;
    printf("c=%d", c);
}
```

pass by value	pass by reference
1) when a function is called, the values of variables are passed	1) Here, addresses of variables are passed
2) Execution is slower since all the values have to be copied to Formal parameter	2) Execution is faster since only addresses are copied
3) change of formal parameter will not affect actual parameter	3) Actual parameter are changed since formal parameter indirectly manipulates actual parameter

passing array to function

In this, array is passed as a parameter to function.

* WAP to read & print n array elements using function by passing array as a parameter.

```
#include<stdio.h>
void input(int a[10]);
void output(int a[10]);
void main()
{
  int x[10];
  clrscr();
  printf("Enter n\n");
  scanf("%d", &n);
  input(x);
  output(x);
  getch();
}
void input(int a[10])
{
  printf("Enter n elements\n");
  for(i=0; i<n; i++)
  scanf("%d", &a[i]);
}
void output(int a[10])
{
  printf("output array is\n");
```

Prof. Chaithra H.E

```
for(i=0; i<n; i++)
printf("%d\t", a[i]);
}
```

Note :- similarly try to write bubble sort program by passing array as a parameter to function.