UNIT II - Managing simple Input and Output operations - Operators and Expressions - Decision Making: Branching statements, looping statements - Function: Declaration, Definition - Passing arguments by value - Recursion - Storage classes.

2.4 INTRODUCTION TO FUNCTION

Function is defined as the block of organized, reusable code that is used to perform the specific action. A function is a subprogram of one or more statements that performs a specific task when called.

Advantages of Functions:

1. Code reusability
2. Better readability
3. Reduction in code redundancy
4. Easy to debug & test.

Classification of functions:

a) Based on who develops the function

b) Based on the function prototype

a) Based on the function prototype

Function prototype is a declaration statement that identify function with function name, data type, a list of a arguments

b) Based on who develops the function

There are two types.

- Library functions
- User-defined functions

Library(Pre-defined) function	User defined function
Contains Pre-defined set of functions	The user defined the functions
User cannot understand the internal working	User can understand internal working
Source code is not visible	Source code is visible
User cannot modify the function	User can modify the function
Example: printf( ), scanf( ).	Example: sum() , square( )

Elements of user defined function (or) Steps in writing a function in a program

1. Function Declaration (Prototype declaration)
2. Function Call

3. Function Definition

1. FUNCTION PROTOTYPE

Function prototype is a declaration statement that identifies function with function name, data type, a list of a arguments. All the function need to be declared before they are used. (i.e. called)

Syntax: returntype functionname (parameter list);

- Return type – data type of return value. It can be int, float, double, char, void etc.
- Function name – name of the function
- Parameter type list –It is a comma separated list of parameter types.

Example:

int add(int a, int b);

Function declaration must be terminated with a semicolon(;).

Types of function prototypes:

1. Function with no arguments and no return values
2. Function with arguments and no return values
3. Function with arguments and one return values
4. Function with no arguments and with return values

<u>Prototype 1</u>: Function with no arguments and no return values

- This function doesn‟t accept any input and doesn‟t return any result.
- These are not flexible.

Program

```
#include<stdio.h>
#include<conio.h>
void show(); //function prototype
void main()
{
    show( );        //function call
    getch();
}
```

```
void show( ) //function definition
{
printf("Hai \n");
}
```

Output:

Hai

Prototype 2: Function with arguments and no return values

Arguments are passed through the calling function. The called function operates on the values but no result is sent back.

Program

```
#include<stdio.h>
#include<conio.h>
void show(int);
void main()
{
int a;
printf("Enter the value for a \n");
scanf("%d", &a);
show(a);
getch();
}
void show(int x)
{
printf("Value =%d", x);
}
```

Output:

Enter the value for a: 10

Value = 10

Prototype 3: Function with arguments and return values

- Arguments are passed through the calling function
- The called function operates on the values.
- The result is returned back to the calling function.

Program

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

```
#include<conio.h>
float circlearea(int);
void main()
{
int r;
float area;
printf("Enter the radius \n");
scanf("%d",&r);
area=circlearea(r);
printf("Area of a circle =%d\n", area);
getch();
}
int circlearea(int r1)
{
return 3.14 * r1 * r1;
}
```

Output:

Enter the radius

2

Area of circle = 12.000

Prototype 4: Function with no arguments and with return values

- This function doesn‟t accept any input and doesn‟t return any result.
- The result is returned back to the calling function.

Program

```
#include<stdio.h>
#include<conio.h>
float circlearea();
void main()
{
float area;
area=circlearea();
printf("Area of a circle =%d\n", area);
```

```
getch();
}
int circlearea()
{
int r=2;
return 3.14 * r * r;
}
```

Output:

Enter the radius

2

Area of circle = 12.000

2. FUNCTION DEFINITION

It is also known as function implementation. When the function is defined, space is allocated for that function in memory.

Syntax

```
returntype functionname (parameter list)
{
statements;
return (value);
}
```

Example

```
int abc(int, int, int) // Function declaration
void main()
{
int x,y,z;
abc(x,y,z) // Function Call
 …
…
}
int abc(int i, int j, int k) // Function definition
{
…….
….
return (value);
}
```

Every function definition consists of 2 parts:

a) Header of the function

b) Body of the function

a) Header of the function

The header of the function is not terminated with a semicolon. ***The return type and the number & types of parameters must be same*** in both function header & function declaration.

Syntax:

returntype functionname (parameter list)

Where,

- Return type – data type of return value. It can be int, float, double, char, void etc.
- Function name – name of the function
- Parameter type list –It is a comma separated list of parameter types.

b) Body of the function

- It consists of a set of statements enclosed within curly braces.
- The return statement is used to return the result of the called function to the calling function.

Program:

```
#include<stdio.h>
#include<conio.h>
float circlearea(int); //function prototype
void main()
{
int r;
float area;
printf("Enter the radius \n");
scanf("%d",&r);
area=circlearea(r); //function call
printf("Area of circle =%f\n", area);
getch();
```

```
}
float circlearea(int r1)
{
return 3.14 * r1 * r1; //function definition
}
```

Output:

Enter the radius

2

Area of circle = 12.000

3. FUNCTION CALL

Function call is used to invoke the function. So the program control is transferred to that function. A function can be called by using its name & actual parameters.

Function call should be terminated by a semicolon (;).

Syntax:

Functionname(argumentlist);

Example

```
int abc(int, int, int) // Function declaration
void main()
{
int x,y,z;
abc(x,y,z) // Function Call
…
…
}
int abc(int i, int j, int k) // Function definition
{
…….
….
return (value);
}
```

Calling function – The function that calls a function is known as a calling function.

Called function – The function that has been called is known as a called function.

Actual arguments – The arguments of the calling function are called as actual arguments.

Formal arguments – The arguments of called function are called as formal arguments.

Steps for function Execution:

1. After the execution of the function call statement, the program control is transferred to the called function.

2. The execution of the calling function is suspended and the called function starts execution.

3. After the execution of the called function, the program control returns to the calling function and the calling function resumes its execution.

Program:

```
#include<stdio.h>
#include<conio.h>
float circlearea(int); //function prototype
void main()
{
int r;
float area;
printf("Enter the radius \n");
scanf("%d",&r);
area=circlearea(r); //function call
printf("Area of a circle =%f\n", area);
getch();
}

float circlearea(int r1)
{
return 3.14 * r1 * r1; //function definition
}
```

Output:

```
Enter the radius
2
Area of circle = 12.000
```

ARGUMENTS

Argument is the value, supply to function call. This value is assigned to corresponding parameter in the function definition. Arguments are specified within a pair of paranthesis, separated by commas.

Types of arguments

i) Required arguments

ii) Keyword arguments

iii) Default arguments

iv) Variable-Length arguments

i) Required arguments

Required arguments are the arguments passed to a function in correct positional order. Here number of arguments in function call should match exactly with function definition.

Example Program:

```
def sum(a,b):
c=a+b
return c
print("The sum is:",sum(6,4))
```

Output:

The sum is:10

(ii) Keyword arguments

The keyword arguments are related to function call. Here caller identifies arguments by parameter name.

Example Program:

```
def sum(a,b):
c=a+b
return c
print("The sum is:",sum(a=5,b=10))
```

Output:

The sum is:15

iii) Default arguments

Default argument is an argument that assume default value if value is not provided in function call.

Example Program:

```
def sum(a,b):
c=a+b
return c
print("The sum is:",sum(a=5))
```

Output:

The sum is:15

iv) Variable-Length arguments

Variable-Length arguments are arguments that makes function call with any number of arguments. Here (*) is placed before the name of the variable.

Example Program:

```
def greeting(*name):
print("Hai",name)
greeting("Welcome")
greeting("Welcome","Hello")
```

Output:

Hai Welcome

Hai Welcome Hello

2.1 PASSING ARGUMENTS BY VALUE

(i) Call by Value (Pass by Value)

While calling a function, the values of actual parameters are passed to the formal parameters. Therefore the called function works on the copy and not on original values of actual parameters.

When arguments are passed by value, C allocates separate memory for formal arguments and copy the actual argument value in that location. Therefore the changes made on formal parameters will not affect the actual parameter.

Syntax for call by value:

function_name(arguments list);

Program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b;
void swap(int ,int);
a=10;
b=20;
printf("\n Before swapping: a = %d and b = %d",a,b);
swap(a, b);
printf("\n After swapping: a= %d and b= %d",a,b);

getch();
}
void swap(int x,int y)
{
int temp;
temp=x;
x=y;
y=temp;
printf("\n In Swap, x=%d and y=%d", x,y);
}
```

Output:

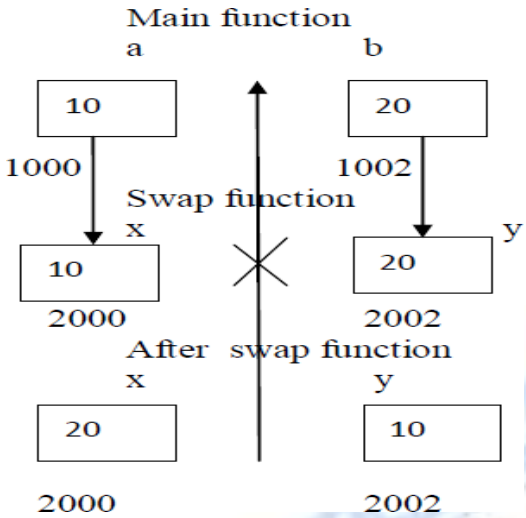
Before swapping: a =10 and b =20

In Swap, x=20 and y=10

After swapping: a =10 and b = 20

Explanation:

In the above program, before calling the swap function, the value of a is 10 and the value of b is 20. During the function call, the values of a and b are passed to x and y. In the swap function, the values of x and y are swapped. Now, x has the value 20 and y has the value 10. But this change does not get reflected in the value of a and b.



(ii) Call by Reference (Pass by Reference)

In this method, the address of the actual argument, in the calling function are copied into the formal arguments of the called function. That is the actual & formal arguments refer same memory location. Therefore the changes made on formal parameters will affect the actual parameter of the calling function.

Example:

```
#include<stdio.h>
#include<conio.h>
void swap(int *, int *)
void main()
{
int a = 10, b = 20;
printf("Before swap a = %d, b = %d \n", a,b);
swap (&a, &b);
printf("After swap a = %d, b = %d", a, b);
getch();
}
void swap (int *x, int *y)
{
int temp;
temp = *x;
```

```
*x = *y;
*y = temp;
printf("\n In Swap, x=%d and y=%d", *x,*y);
}
```

Output:

Before swap a = 10, b = 20

In Swap, x=20 and y=10

After swap a = 20, b = 10

Explanation:

In the above program, before calling the swap function, the value of a is 10 and the value of b is 20. During the function call, the references of a and b are passed to x and y. In the swap function, the values of x and y are swapped. Now, x has the value 20 and y has the value 10. This change gets reflected in the value of a and b.