UNIT-II

FUNCTION, POINTERS, STRUCTRURES AND UNION

Functions – Pass by value – Pass by reference – Recursion – Pointers - Definition – Initialization – Pointers arithmetic. Structures and unions - definition – Structure within a structure - Union - Programs using structures and Unions – Storage classes, Pre-processor directives.

FUNCTION

A function is self-contained program segment that carries out some specific, well defined task.

Need for Functions:

Many programs require a set of instructions to be executed repeatedly from several places in aprogram, then the repeated code can be placed within a single function, which can be accessedwhenever it is required.

→Dividing the program into separate well-defined functions facilitates each function to be writtenand tested separately.

→Understanding, coding, and testing multiple separate functions is easier than doing the same for onebig function main function.

→ A big program is broken into comparatively smaller functions.

Advantages of Functions

-Improves readability of code.

-Divides a complex problem into simpler ones

- Improves reuseability of code.

-Debugging errors is easier.

-modifying a program is easier.

Terms

→A function *f* that uses another function *g* is known as the *calling function*, and *g* is known as the *called function.*

→The inputs that a function takes are known as *arguments*.

→When a called function returns some result back to the calling function, it issaid to *return* that result.

→The calling function may or may not function,which can theot pass *parameters* to the called function.

→***Function Declaration***

→***Function Definition***

→*Function Call*

Function Definition

When a function is defined, space is allocated for that function in the memory.

A functiondefinition comprises of two parts:

- Function header
- Function body

The syntax of a function definition can be given as:

```
return_data_type function_name(data_type variable1, data_type variable2,..)
{
statements
return(variable);
}
```

```
void sum(int a,int b)
{
Int c;
c=a+b;
return(c);
}
```

```
int sum(int a,int b)
{
Int c;
c=a+b;
return(c);
}
```

Note that the number of arguments and the order of arguments in the function header must be the same as that given in the function declaration statement.

Function Call

The function call statement invokes the function. When a function is invoked, the compiler jumps to the called function to execute the statements that are a part of that function. Once the called function is executed, the program control passes back to the calling function. A function call statement has the following

Syntax-1:

```
function_name(variable1, variable2, ...);
```

```
function_name(argument1, argument2, ...);
```

If the return type of the function is not void then the following syntax can be used.

Syntax-2

```
variable_name = function_name(variable1, variable2, ...);
```

```
variable_name = function_name(parameter1, parameter2, ...);
```


The following points are to be noted while calling a function:

→Function name and the number and the type of arguments in the function call must be same as that given in the function declaration and the function header of the function definition.

→Names (and not the types) of variables in function declaration, function call, and header of function definition may vary.

Function Declaration

It is also called as function prototype. A function prototype consists of 4 parts

(1) Return type

(2) function name

(3) parameter list

(4) terminating (;) semicolin

```
data_type function_name(data_type1 variable1, ...,data_type n variable n);
```

A function prototype tells the compiler that the function may later be used in theprogram. The function prototype is not needed if function is placed before main function code.

Example program

```
#include <stdio.h>
void sum(int a ,int b); //FUNCTION DECLARATION
int main()
{   int a,b;
   scanf("%d%d",&a,&b);  //get input values for a and
   bsum(a,b); //FUNCTION CALL
}
void sum(int x,int y) // FUNCTION DEFINITION
{
 int  c;
 c=x+y
 ;
 printf("c=%d",c);
}
```

o/p
2
3
c=5

void data type indicates that the function is returning nothing .(i.e) if the function is not returninganything then its datatype is as specified as void.

Example program without function prototype(function declaration)

```
#include <stdio.h>
void sum(int x,int y) // FUNCTION DEFINITION
{  int  c;
  c=x+y
  ;
 printf("c=%d",c);
}
int main()
{   int a,b;
   scanf("%d%d",&a,&b);  //get input values for a and
   bsum(a,b); //FUNCTION CALL
}
```

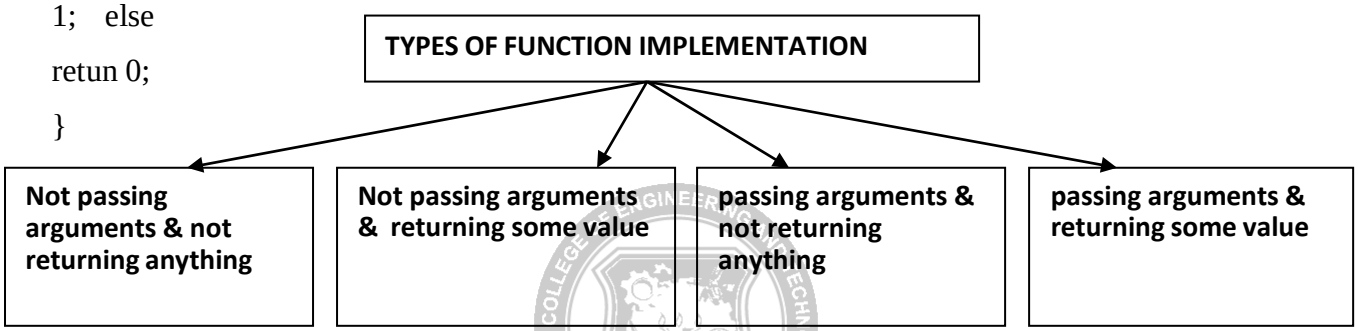
o/p
2
3
c=5

Eg-Write a program to find whether a number is even or odd using functions.

```
#include <stdio.h>
int evenodd(int); //FUNCTION DECLARATION
int main()
{
   int num, flag;
   printf("\n Enter the number :
```

Output
Enter the number : 78
78 is EVEN

```
");scanf("%d", &num);
flag = evenodd(num); //FUNCTION CALL
 if (flag == 1)
 printf("\n  %d  is  EVEN",
num);else
printf("\n  %d  is  ODD",
num);return 0;
}
int evenodd(int a)  // FUNCTION DEFINITION
{
 if(a%2 == 0)
return
1;   else
retun 0;
}
```



CASE-1 NOT PASSING ARGUMENTS & NOT RETURNING SOME VALUE

```
#include <stdio.h>
int sum( );  //FUNCTION DECLARATION
int main( )
 int a,b,c;
 scanf("%d%d",&a,&b); // get input values for a and
 bc=x+y;
 printf("c=%d",c);
}
```

CASE-2 NOT PASSING ARGUMENTS & RETURNING SOME VALUE

```
#include <stdio.h>
int sum( );  //FUNCTION DECLARATION
int main( )
{   sum( );   //FUNCTION CALL
   printf("c=%d",c);
}
int sum( )  // FUNCTION DEFINITION
{
 int a,b,c;
scanf("%d%d",&a,&b); // get input values for a and
 bc=x+y;
return c;
 }
```

o/p
2
3
c=5

CASE-3 PASSING ARGUMENTS & NOT RETURNING ANYTHING

```
#include <stdio.h>
int sum( int a,int b); //FUNCTION DECLARATION
int main( )
{   int a,b;
  scanf("%d%d",&a,&b); // get input values for a and b
    sum(int a,int b );  //FUNCTION CALL
   }
int sum( int x,int y) // FUNCTION DEFINITION
{
  int   c;
  c=x+;
  printf("c=%d",c);
}
```

o/p
2
3
c=5

CASE-4 NOT PASSING ARGUMENTS & RETURNING SOME VALUE

```
#include <stdio.h>
int sum( int a,int b); //FUNCTION DECLARATION
int main( )
{   int a,b,c;
    scanf("%d%d",&a,&b); // get input values for a and
  bc=sum(int a,int b );     //FUNCTION     CALL
  printf("c=%d",c);
return 0;
}
int sum( int x,int y) // FUNCTION DEFINITION
{           int
 result
 result=x+y;
 return  result;
 }
```

o/p
2
3
c=5

PASSING ARGUMENTS(PARAMETERS) TO FUNCTION

In programming function argument is commonly referred as **actual parameter** and function parameter is referred as **formal parameter**.

```
                    // Formal parameters
void add(int num1, int num2) // Function definition
{
    // Function body
}


int main()
{
            // Actual parameters
    add(10, 20); // Function call

    return 0;
}
```

There are two ways by which parameters can be passed to a function

1. Call by Value
2. Call by Reference

Call by value

In Call by value, during function call actual parameter value is copied and passed to formal parameter. Changes made to the formal parameters does notaffect the actual parameter.

Eg Program - C program to swap two numbers using call by value

```
#include<stdio.h>
int main()
{
   int n1, n2;
printf("Enter two numbers: ");
scanf("%d%d", &n1, &n2);
printf("In Main values before swapping: %d %d\n\n", n1, n2);
swap(n1, n2);
```

```
printf("In Main values after swapping: %d %d", n1, n2);
return 0;
    }
void swap(int num1, int num2)
    {
int temp;
printf("In Function values before swapping: %d %d\n", num1, num2);
temp = num1;
num1 = num2; num2 =
temp;
printf("In Function values after swapping: %d %d\n\n", num1, num2);
    }
```

Output -

Enter two numbers: 10 20
In Main values before swapping: 10 20
In Function values before swapping: 10 20
In Function values after swapping: 20 10 In
Main values after swapping: 10 20

NOTE : In the above program swap() function does not alter actual parameter value. Before passing the value of *n1* and *n2* to the swap() function, the C runtime copies the value of actual parameter *n1* and *n2* to a temporary variable and passes copy of actual parameter. Therefore inside the swap() function values has been swapped, however original value of *n1* and *n2* in main() function remains unchanged.

Call by reference

In Call by reference we pass memory location (reference) of actual parameter to formal parameter. It uses pointers to pass reference of an actual parameter to formal parameter. Changes made to the formal parameter immediately reflects to actual parameter.

Eg Program - C program to swap two numbers using call by reference

```
#include <stdio.h>
int main()
{    int n1, n2;
   printf("Enter two numbers: ");
   scanf("%d%d", &n1, &n2);
   printf("In Main values before swapping: %d %d\n\n", n1, n2);
   swap(&n1, &n2);
   printf("In Main values after swapping: %d %d", n1, n2);
   return 0;
}
void swap(int * num1, int * num2)
{
   int temp;
   printf("In Function values before swapping: %d %d\n", *num1, *num2);
   temp = *num1;
```

```
    *num1 = *num2;
    *num2 = temp;
    printf("In Function values after swapping: %d %d\n\n", *num1, *num2);
}
```

Output :-

Enter two numbers: 10 20

In Main values before swapping: 10 20

In Function values before swapping: 10 20

In Function values after swapping: 20 10 In

Main values after swapping: 20 10

In above example instead of passing a copy of *n1* and *n2*, to swap() function. Operations performed on formal parameter is reflected to actual parameter (original value). Hence, actual swapping is performed inside swap() as well as main() function.

WRITE THESE SWAP PROGRAM(S) FOR YOUR EXAMS

Call by value

```
#include<stdio.h>
Void swap(int x,int y);
int main( )
{
   int a = 10, b = 20 ;
   swap (a,b) ; // calling by value
   printf ( "\n Before swapping x and y) ;
   printf ( "\na = %d b = %d", a, b ) ;
   return 0;
}
void swap( int x, int y )
{
   int t  ;
   t = x ;
   x = y ;
   y = t ;
   printf ( "\n After swapping x and y) ;
   printf( "\nx = %d y = %d", *x,*y);
}
```

```
Before swapping x and y
10    20
After swapping x and y
20    10
```

Call by reference

```
#include<stdio.h>
Void swap(int *x,int *y);
int main( )
{
   int a = 10, b = 20 ;
   swap ( &a, &b ) ; // calling by reference
   printf ( "\n Before swapping x and y) ;
   printf ( "\na = %d b = %d", a, b ) ;
   return 0;
}
void swap( int *x, int *y )
{
   int t ;
   t = *x ;
   *x = *y ;
```

```
Before swapping x and y
10    20
After swapping x and y
20    10
```

```
    *y = t ;
    printf ( "\n After swapping x and y) ;
    printf( "\nx = %d y = %d", *x,*y);
}
```

Call by value	**Call by reference**
When a function is called the actual values are passed	When a function is called the address of values(arguments) are passed
The parameters passed are normal variables	The parameters passed are pointer variables
In call by value, actual arguments cannot be modified.	Modification to actual arguments is possible within from called function.
Actual arguments remain preserved and no chance of modification accidentally.	Actual arguments will not be preserved.
Calling Parameters: swap(a,b)	Calling Parameters: swap(&a,&b)
Receiving parameters: void swap(int x, int y)	Receiving parameters: void swap(int *x, int *y)

PASSING ARRAYS TO A FUNCTION

Passing Array Element to a Function

```
#include <stdio.h>
void display(int a);
int main()
{
int age[] = { 21, 31, 41 };
 display(age[2]);      //Passing          array
element age[2] only.return 0;
}
void display(int age)
 {
 printf("%d", age);
 }
```

output

41

Passing Entire Array(1-Dimentional array) Element to a Function

```
#include <stdio.h>
float average(float age[]);int main()
 {
float avg, age[] = { 30.5,40.5,60.5,80.5};
avg=average(age)
printf("Averageage=%.2
f", avg);
return 0;
 }
float average(float age[])
 {
int i;float avg, sum = 0.0;
for (i = 0; i < 4; ++i)
        {
```

Output

53.000000

```
sum += age[i];
            }
avg = (sum / 4);
return avg;
     }
```

Passing Entire Array(2-Dimentional array) Element to a Function

```
     #include <stdio.h>
     void displaynumbers(int num[2][2]);
int main( )
{
Int num[2][2],i,j;
Printf("enter numbers");
for(i=0;i<2;i++)
for(i=0;i<2;i++)
     scanf("%d",&num[i][j]);
     displaynumbers(num) //
     calling functionreturn 0;
     }
     void displaynumbers(int num[2][2])  / / function declaration
     {
int i,j;
Printf("display numbers");
for(i=0;i<2;i++)
 for(i=0;i<2;i++)
printf("%d",num[i][j]);
     }
```

Output

```
Enter numbers
10
20
30
40
Display numbers
10
20
30
40
```