UNIT-IV

FUNCTIONS IN C:

1. A function is a self contained block of statements that can used to perform a specific task.
2. Generally a function is an independent set of statements that carries out some specific well defined task.
3. It is written after or before the function main().
4. A function has two components, function prototype and function definition.
5. Function prototype is used to describe the function properties or characteristics to the compiler.
   Return_type function_name(argument/parameter type)
6. Return type specifies return value of the function.
7. Function name is any user defined name except keywords.
8. Argument/parameter type specifies what type of values you are giving input to the function.

Syntax:

```
Return_type function_name(argument list)
{
        ---------
        Statements
        ---------
}
```

Advantages of functions:

1. Functions provide code reusability in the program. By calling the same function several times in the main program, we can achieve reusability of code.
2. Functions provide modular approach means code separation.
3. Function implementation in program will make debugging easy.

```
#include<stdio.h>
#include<conio.h>
int addition(int,int);
main()
{
        int a,b,c;
        clrscr();
        printf("Enter a,b values:");
        scanf("%d%d",&a,&b);
        a=addition(a,b);
        printf("Sum=%d",c);
        getch();
}
int addition(int x,int y)
{
        int z;
        z=x+y;
        return z;
}
```

ACTUAL ARGUMENTS/PARAMETERS:

The arguments in the function parenthesis at the place of function call are called as actual arguments/parameters.

Ex: addition(a,b); // a,b are called actual arguments/parameters

FORMAL ARGUMENTS/PARAMETERS:

The arguments/parameters in the function parenthesis at the place of function definition are called as formal arguments/parameters.

Ex: addition(int a,int b); // a,b are called as formal arguments/parameters

RETURN STATEMENT:

Return Statement is used to send values to main function from the function definition.

CATEGORIES OF FUNCTIONS:

Based on function return type and arguments list, functions are categorized into following categories.

1) Function with No Arguments and No Return type
2) Function with Arguments and No Return type
3) Function with No Arguments and Return type
4) Function with Arguments and Return type

a) Function with No arguments and No Return type:

In this category of the function, main program will not send any arguments to the function and function will not send any values to the main.

Syntax:

```
main()
{
    function();
}
    function()
    {
        Statement 1;
        Statement 2;
        …
    }
```

```
#include<stdio.h>
#include<conio.h>
void addition();
main()
{
    clrscr();
    addition();
}
void addition()
{
    int a,b,c;
    printf("Enter a,b values:");
    scanf("%d%d",&a,&b);
    c=a+b;
    printf("Additoin result is %d",c);
}
```

b) Function with Arguments and No Return type:

In this category of the function, the main program will send argument values to the function but the function will not return any values.

Syntax:
```
main()
{
    function(argument1,argument2);
}
    function(int arg1,int arg2)
    {
        Statements;
    }
```

Here actual argument values are copies into formal parameter variables.

```
#include<stdio.h>
#include<conio.h>
void area(int);
main()
{
    int r;
    clrscr();
    printf("Enter radius:");
    scanf("%d",&r);
    area(r);
    getch();
}
void area(int radius)
{
    float a;
    a=3.141*radius*radius;
    printf("Area=%f",a);
}
```

c) Function with No Arguments and Return type:

In this category of function the calling program will not send any arguments but the called function will return value to the calling program.

Syntax:
```
main()
{
    return_type variable_name=function_name();
}
function_name()
{
        Statements;
        return value;
}
```

Example:
```
int add();
main()
{
    c=add();
    printf("%d",c);
}
```

```
int add()
{
        int x=10,y=20;
        return(x+y);
 }
```

d) Function with Arguments and Return Type:

In this category of function the main function will send argument values to the function and the function will return some value to the main function.

Syntax:

```
main()
{
        return_type  variable=function_name(arguments);
}
return_type function_name(arguments)
{
        Statements;
        return value;
}
```

Example:

```
main()
{
     int a=10,b=20,c;
     c=add(a,b);
     printf("%d",c);
 }
    int add(int x,int y)
    {
        return(x+y);
    }
```

PARAMETER PASSING TECHNIQUES:

The values are passed to the function through the arguments. In C language we can pass values from calling program/function to called program/function in two types.

1. Call by value
2. Call by reference

CALL BY VALUE:

In call by value mechanism the values of actual arguments at the function call are copied into the formal arguments of the function definition.

After manipulations in the function definitions changes done in formal arguments will not affect the actual arguments.

```
#include<stdio.h>
#include<conio.h>
void  swap(int,int);
main()
{
        int x,y;
        clrscr();
        printf("Enter x,y values:");
        scanf("%d%d",&x,&y);
        swap(x,y);
        printf("After swapping in main x=%d y=%d",x,y);
        getch();
}
```

```
void swap(int x,int y)
{
        int z;
        z=x;
        x=y;
        y=z;
        printf("after swapping in function x=%d  y=%d",x,y);
}
```

CALL BY REFERENCE:

In this mechanism instead of sending argument values to function definition, address of actual arguments will be send.

The changes made in the formal arguments will affect on actual arguments.

```
#include<stdio.h>
#include<conio.h>
void swap(int*,int*);
main()
{
        int x,y;
        clrscr();
        printf("Enter x,y values:");
        scanf("%d%d",&x,&y);
        swap(&x,&y);
        printf("After swapping in main x=%d y=%d",x,y);
        getch();
}
void swap(int *x,int *y)
{
        int z;
        z=*x;
        *x=*y;
        *y=z;
        printf("after swapping in function x=%d  y=%d",x,y);
}
```

RECURSIVE FUNCTION:

1. A function which calls itself is called as recursive function.
2. It is the process of calling a function by itself, until some condition is satisfied.
3. It is used for repetitive computations in which each action is stated in terms of previous results.
4. A recursion procedure has to satisfy the following conditions.
    a. A recursive function should have a specific condition
    b. Changes must be made in the argument values passed to the function

In recursive function there should be a condition to terminate the function.

Example:

```
main()
{
        int n,f;
        printf("Enter a number:");
        scanf("%d",&n);
```

```
        f=factorial(n);
        printf("Factorial = %d",f);
}
int factorial(int x)
{
        if(x==1)
                return 1;
        else
                return (x*factorial(x-1));
}
```

DIFFERENCES BETWEEN CALL BY VALUE AND CALL BY REFERENCE

Call by Value	Call by Reference
➔ Values of actual arguments will be copied into formal arguments.	➔ The address of actual arguments will be copies into formal arguments.
➔ Manipulations done on values.	➔ Manipulations done on addresses.
➔ Changes in formal arguments will not affect actual arguments.	➔ Changes in formal arguments will affect actual arguments.

STORAGE CLASSES:

To fully define a variable one needs to mention not only its type but also its "storage class". In other words, not only do all variables have a data type, they also have a "storage class".

A variable"s storage class tells us:

1. Where the variable would be stored.
2. What will be the initial value of the variable, if initial value is not specifically assigned; i.e., the default initial value.
3. What is the scope of the variable; i.e., in which functions the value of the variable would be available.
4. What is the life of the variable; i.e., how long would the variable exist.

The following are the storage classes in C.

- ➔ Automatic storage class
- ➔ Register storage class
- ➔ Static storage class
- ➔ External storage class

AUTOMATIC STORAGE CLASS-(auto)

Storage : Memory
Default initial value : Garbage value
Scope : Local to the block in which the variable is defined.
Life : Till the control remains within the block in which the variable is defined.

Example:

```
main()
{
        auto int i=1;
        clrscr();
        {
                auto int i=2;
                {
                        auto int i=3;
                        printf("\n i=%d",i);
                }
                printf("\n i=%d",i);
        }
        printf("\n i=%d",i);
        getch();
}
```

OUTPUT:

```
i=3
i=2
i=1
```

REGISTER STORAGE CLASS-(register)

Storage	: CPU Registers
Default initial value	: Garbage value
Scope	: Local to the block in which the variable is defined.
Life	: Till the control remains within the block in which the variable is defined.

The value stored in a CPU register can always be accessed faster than the one that is stored in memory.

Example:
```
main()
{
        register int i;
        for(i=1;i<=10;i++)
                printf("\n%d",i+);
}
```
We cannot use register storage class for all types of variables. This is because the CPU registers in a microprocessor are usually 16 bit registers and therefore cannot hold a float or double or long value.

STATIC STORAGE CLASS-(static)

Storage	: Memory
Default initial value	: Zero
Scope	: Local to the block in which the variable is defined.
Life	: Value of the variable persists between different function calls.

```
main()
{
        increment();
        increment();
        increment();
}
increment()
{
        auto int i=1;
        printf("\n%d",i);
        i=i+1;
}
```
OUTPUT:
1
1
1

```
main()
{
        increment();
        increment();
        increment();
}
increment()
{
        auto int i=1;
        printf("\n%d",i);
        i=i+1;
}
```
OUTPUT:
1
2
3

Static variables don"t disappear when the function is no longer active. Their values persist. If the control comes back to the same function again the static variables have the same values they had last time around.

EXTERNAL STORAGE CLASS-(extern)

Storage	: Memory
Default initial value	: Zero
Scope	: Global
Life	: As long as the program‟s execution doesn‟t come to an end.

External variables are declared outside all functions, they are available to all functions that use extern variable.

```
int i=1;
main()
{
        printf(“\n i=%d”,i);
        increment();
        increment();
        decrement();
        decrement();
}
increment()
{
        i=i+1;
        printf(“\n On incrementing i=%d”,i);
}
decrement()
{
        i=i-1;
        printf(“\n On decrementing i=%d”,i);
}
```

OUTPUT:

```
On incrementing i=1
On incrementing i=2
On decrementing i=1
On decrementing i=0
```

STRUCTURES

A structure is a user defined datatype. A structure is heterogeneous collection of related fields. Here fields are called structure elements or structure members.

A structure can be considered as a template used for defining a collection of variables under a single name. Structures help programmers to group elements of different datatypes into a single unit.

Syntax:

```
struct structure-name
{
        Datatype-1 structure-element1;
        Datatype-2 structure-element2
        -  -  -
        -  -  -
        Datatype-2 structure-elementn;
};
```

→ The keyword "struct" is used to define structure.
→ Variables inside the structure are called structure elements.
→ By using structure variable we can access structure elements.
→ When we declare structure variable, memory allocation takes place.
→ Memory allocated to the structure variable is sum of individual elements in the structure.

Syntax for Structure Variable:

```
struct structure-name variable-name;
```

Example:

```
struct student
{
        int rno;
        char name[20];
        float fees;
};
main( )
{
        struct student s;
}
```

Here rno, name, fees are called as structure elements and "s" is called as structure variable.

INITIALISING STRUCTURE:

The structure elements are accessed using dot (•) operator. It is also called as membership operator. The individual elements are initialized as follows.

```
s.rno=10;
s.name="Aditya";
s.fees=1000;
```

We can also initialize structure element values inside curly braces, with each value separated by (,).

```
struct student s={10,"Aditya",10000};
```

```
struct student
{
```

```
        int rno;
        char name[50];
        float fees;
};
main( )
{
        struct student s;
        clrscr( );
        printf("enter student rno, name, fees :");
        scanf("%d%s%f",&s.rno,s.name,&s.fees);
        printf("STUDENT DETAILS");
        printf("\n\tRNO\t\tNAME\t\tFEES\n");
        printf("%d\t\t%s\t\t%f",s.rno,s.name,s.fees);
        getch( );
}
```

ARRAYS OF STRUCTURES:

C does not limit a programmer to storing simple datatypes in an array. User can define structures as an element of an array.

The general syntax used for declaration of array of structures can be

```
struct structure-name variable-name[size];

struct student s[10];
```

This defines an array called s that has 10 elements. Each element inside the array will be of type struct student. Referencing an element in the array is as follows.

S[0] • rno=10;
S[0] • name=”Aditya”;
S[0] • fees=10000;

```
struct student
{
        int rno;
        char name[20];
        float fees;
};
main( )
{
        struct student s[10];
        int i,n;
        clrscr( );
        printf(“How many students:”);
        scanf(“d”,&n);
        for(i=0;i<n;i++)
        {
                printf(“Enter student RNO,NAME,FEES:\n”);
                scanf(“%d%s%f”,&s[i].rno,s[i].name,&s[i].fees);
        }
        printf(“STUDENT DETAILS”);
        printf(“\n\tRNO\t\tNAME\t\tFEES\n”);
        for(i=0;i<n;i++)
        {
                printf(“%d\t\t%s\t\t%f”,s[i].rno,s[i].name,s[i].fees);
        }
        getch( );
}
```

NESTED STRUCTURES:

A structure can be embedded within another structure. In other words when a structure is declared and processed with in another structure then it is called nested structure or structures within structures.

Structures can be nested in two ways.

1→ Using previously defined structure:

```
        struct structure-name1
        {
                structure element;
                -   -   -
                -   -   -
        };
        struct structure-name2
        {
                structure element;
                -   -   -
                -   -   -
                struct structure-name1 variable-name;
        };
```

Example:
```
struct dob
{
        int  day;
        int month;
        int year;
};
struct student
{
        int rno;
        char name[100];
        struct dob d;
};
```
Defining structure variable in another structure can be called as Nested Structure.

2→Declare new Structure within another structure

```
Struct structure-name1
{
        Structure element-1;
        -   -   -
        -   -   -
        Struct structure-name2
        {
                Structure element
                .       .       .
                .       .       .
        }var;
}var1;
```

Example:
```
struct student
{
        int rno;
        char name[20];
        struct dob
        {
                int  day;
                int month;
                int year;
        }d;
};
```
The inner structure elements can be linked with the outer structure variable by dot operator. This can be represented as

Outer-variable • inner-variable • inner-element;

UNION

A union is a user-defined datatype that allows you to store different datatypes in same memory location.

We can define a union by using "UNION" keyword. The union statement defines new datatype with more than one member for your program.

Syntax:

```
union union-name
{
        union element-1;
        union element-2;
        -   -   -
        -   -   -
        union element-n;
}union-variable;
```

Example:

```
union data
{
        int i;
        float f;
        char str[20];
}d;
```

Here union variable d can store an integer, a floating point number or a string. It means a single variable i.e., same memory location can be used to store multiple types of data.

The memory occupied by a union will be the largest member memory of the union. In the above example data d will occupy 20 bytes of memory.

ACCESSING UNION MEMBERS:

→To access any member of a union we use the member access operator •.

→By using keyword union we can define union variables.

Union union-name variable-name;

```
union student
{
        int rno;
        float fees;
        char name[20];
};
main( )
{
        union student s;
        clrscr( );
        printf("Enter student roll number, fee paid and name:");
        scanf("%d%f%s",&s.rno,&s.fees,s.name);
        printf("student roll number    :%d",s.rno);
        printf("student fee paid       :%f",s.fees);
        printf("student name           :%s",s.name);
        getch( );
}
```

OUTPUT:

```
Enter student roll number, fee paid and name:
44 10000 Aditya
Student roll number    :Garbage value
Student fee paid       :Garbage value
```

DIFFERENCES BETWEEN STRUCTURES AND UNIONS:

1. Keyword struct is used to define structure.	→ Keyword union is used to define union.
2. Within a structure all members get memory allocated.	→ For union compiler allocates the memory of the largest among all members.
3. The total size of the structure is the sum of size of all structure elements.	→ The total size of the union is the largest member size in the union.
4. Within a structure we can access any member at any time.	→ Within a union we can access most recently stored value.
struct structure-name { Datatype-1 structure-element1; Datatype-2 structure-element2 - - -	union structure-name { Datatype-1 union-element1; Datatype-2 union-element2 - - -

- - - Datatype-2 structure-elementn; }variable-name;	- - - Datatype-2 union-elementn; }variable-name;
5. Several members of a structure can initialize at once.	→ Only the first member of a union can be initialized.

ENUMERATED DATA TYPE:

An enumeration is user-defined data type consist of integer constants and each integer constant given a name. keyword "enum" is used to define enumerated datatype.

Syntax:

enum type-name { value-1,value-2, ... value-n };

Here type-name is the name of enumerated datatype and value-1, value-2 ..... value-n are the values in the enumeration.

By default value-1 will be equal to 0, value-2 will be 1 and so on, the programmer change the default value.

Example:

```
main( )
{
        enum month{jan=1,feb,mar,apr,may,june,july,aug,sept,oct,nov,dec};
        enum month m;
        clrscr( );
        printf("enter your month number:");
        scanf("%d",&m);
        switch(m)
        {
                case jan: printf("Number of Days 31");
                          break;
                case feb: printf("Number of Days 28");
                          break;
                case mar: printf("Number of Days 31");
                          break;
                case apr: printf("Number of Days 30");
                          break;
                case may: printf("Number of Days 31");
                          break;
                case june: printf("Number of Days 30");
                          break;
                case july: printf("Number of Days 31");
                          break;
                case aug: printf("Number of Days 31");
                          break;
                case sept: printf("Number of Days 30");
                          break;
                case oct: printf("Number of Days 31");
                          break;
                case nov: printf("Number of Days 30");
                          break;
```

```
                case dec: printf("Number of Days 31");
                        break;
        }
        getch( );
}
```