PROGRAMMING IN C

NOTES

By

Prof. Prabhu Kichadi

9880437187

UNIT – IV

User Defined Functions: Need for user defined functions; Format of C user defined functions; Components of user defined functions - return type, name, parameter list, function body, return statement and function call; Categories of user defined functions - With and without parameters and return type. User defined data types: Structures - Structure Definition, Advantages of Structure, declaring structure variables, accessing structure members, Structure members initialization, comparing structure variables, Array of Structures; Unions - Union definition; difference between Structures and Unions.

User Defined Functions: *"Function is a re-usable block of code that performs a specific task, it is executed when it is invoked".*

- Functions which are created for programmer application requirement are called user defined functions.

Need for user defined functions:

User-defined functions allow programmers to create their own routines and procedures that the computer can follow;

Need or Advantages of user defined functions:

1. Code Becomes modular.
2. Code reusability
3. Code maintenance becomes easy.
4. Code length can be reduced.

Function: *"Function is a re-usable block of code that performs a specific task, it is executed when it is invoked".*

Remember these points when we use Functions,

- Generally, every function takes some inputs and processes it and returns some outputs to the caller.
- Inputs are supplied to the function in the form of function arguments(parameters) (Variable Declarations).
- Every function returns maximum one value to the caller,
- Every function has its own unique name(identifier), optional parameters, and a return type (valid data type).

Components of user defined functions

return type, name, parameter list, function body, return statement and function call;

Every function has the following components:

1. return type: Every function in C, returns zero or one value to the caller after its execution. Using **return** keyword, the function has to return zero or one value to the caller along with control.

Ex: int addNums(int x, int y);

-In this example the function addNums( ) returns one int value to the caller using return statement.

2. functionName : This is the name of the function we are creating; it must be a valid identifier.

3. Parameters OR Arguments: These are the inputs to the functions, which are nothing list of variable declarations, so that whenever we are calling a function, we must send/pass values/variables to functions. A copy of data set will be sent.

Ex:

```
        void add(int x, int y);
```

- In this example x and y are the arguments to add( ) function. So that whenever we are calling this function, we must pass two int values/variables to this function, otherwise compiler will throw an error.

4. Function definition: The actual task of the function must be written inside { } . It is also called as Function implementation, task, Function body.

Ex:

```
int addNums(int x, int y)
{
        int res;
        res = x + y;
        return res;
}
```

- In this example, addNums( ) is a function, the task of the function is to receive two integral numbers as input and it is going to find out the addition of two numbers and returns the result.

5. return statement: return keyword is used to transfer the control along with any value based on returntype of the function.

- It indicates the termination of function; it is used to return control back to the caller.

6. Function Call/Invoke:

The statement which calls a function for execution is nothing but a function call. Without calling a function, the function body will not execute.

Ex:

```
#include<stdio.h>
int addNums(int x, int y);   //Function Declaration or Prototype

int addNums(int x, int y)   //Function Definition or Body
{
   int res;
   res = x + y;
   return res;
}
```

```
int main()
{
  int sum;

  sum = addNums(10,20);       //Function Call
  printf("\nAddition Result = %d",sum);

  sum = addNums(30,40);       //Function Call
  printf("\nAddition Result = %d",sum);
}
```

- In this example addNums( ) is a function it must be called to execute.

Every function has 3 major components:

1. **Function Declaration/prototype** – First Line of the function to indicate compiler.
2. **Function Definition/Body/Implementation** – Actual work of the function is defined inside Curly Braces { }.
3. **Function Call/Invoke (Optional)** – To execute the function body, we need to call the function.

Parameters: Refers to any declaration within the parentheses following the function name in a function declaration & definition.
Arguments: Refers to any expression or variable names or values within the parentheses of a function call.

1. Function Declaration/prototype – First Line of the function to indicate compiler the signature of the function.
Syntax:

returntype functionname (optional parameters);

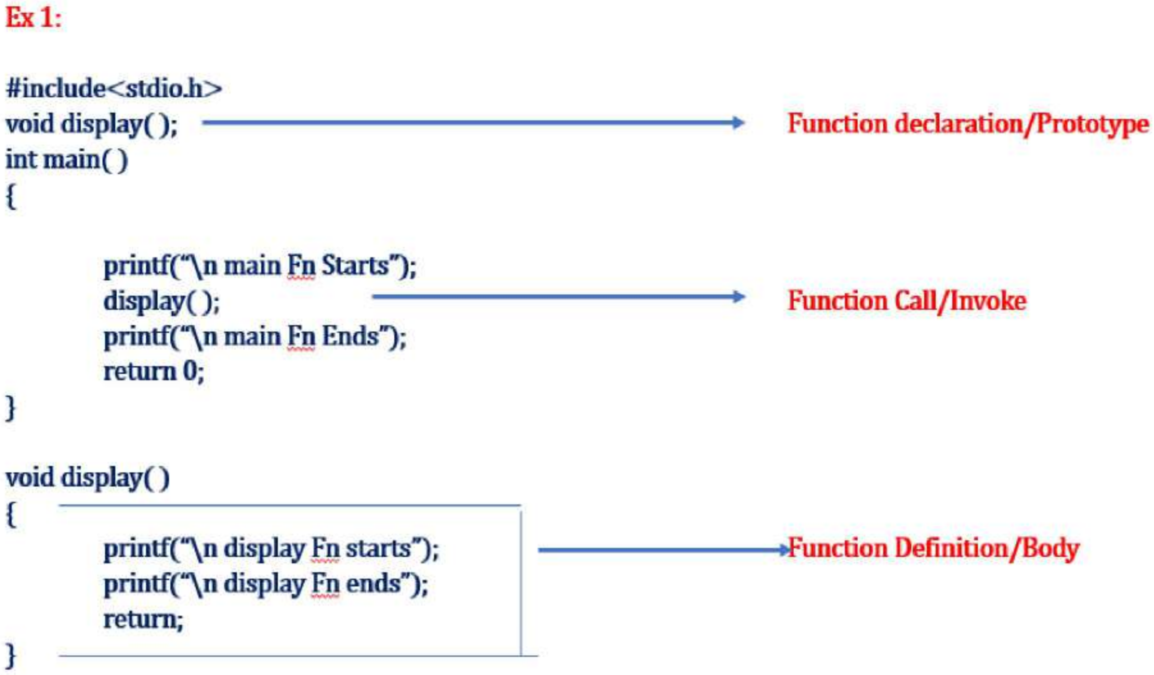
1. **returntype** – it must be a valid data type related keywords, it tells what type of a value is going to return by the function when it completes its execution. Ex: void, int, float, char, double, Student, Employee etc.
2. **Functionname** – It is the name of the function provided by the programmer; it must be a valid identifier.

3. **Optional parameters** – These are inputs to the function; list of variables declarations is written here.
It must be written, generally outside all functions body and after preprocessor directives, it must be global.

2. Function Definition/Body/Implementation – The actual task of the function is defined inside { }

Syntax:

```
returntype functionname (optional parameters)
{
        //Actual task of the function
        //Statements representing function implementation
        // return statement
}
```



Ex 1:

```
#include<stdio.h>
void display( );
int main( )
{

        printf("\n main Fn Starts");
        display( );
        printf("\n main Fn Ends");
        return 0;
}

void display( )
{
        printf("\n display Fn starts");
        printf("\n display Fn ends");
        return;
}
```

Categories/Types of User Defined Functions

User Defined Functions*: Functions which are created by the programmer as per his/her application requirements.*

- These are not standard functions which are not available in any library.

Classification of user defined functions,

1. without returntype and without parameters.
2. without returntype and with parameters.
3. with returntype and without parameters.
4. with returntype and with parameters.

1. without returntype and without parameters.

Defining (Creating) a function which is having no return type(void) and no parameters (no inputs).

Ex:

```
#include<stdio.h>
void disp( );   //Function prototype
int main( )   //Calling Function
{
        disp( );   //Function Call
        return 0;
}
void disp( )   //Called Function
{
        printf("\nHello BCA BCA");
        return;
}
```

Output:

```
Hello BCA BCA
```

2. without returntype and with parameters.

Defining (Creating) a function which is having no return type(void) and any no parameters (no inputs, means any no of variable declarations).

- At the time function call we must pass arguments to the function.

Ex:

```
#include<stdio.h>
void disp(int x);   //Function prototype, parameter is x of int type
int main()   //Calling Function
{
        int a = 199;
        disp(a);   //Function Call, passing argument a to disp( )
        return 0;
}
void disp(int x)   //Called Function
{
        printf("\nx = %d",x);
        return;
}
```

Output:

```
x = 199
```

3. with returntype and without parameters.

Defining (Creating) a function which is having a return type and no parameters (no inputs).

Ex:

```
#include<stdio.h>
int getNum( );   //Function prototype
int main()   //Calling Function
{
        int a;
        a = getNum( );   //Function Call
        printf("\na = %d",a);
        return 0;
}
int disp( )   //Called Function
{

        Return 33;
}
```

Output:

```
a = 33
```

4. with returntype and with parameters. (Recommended for use)

Defining (Creating) a function which is having a return type and any no parameters (arguments must be passed).

Ex:

```
#include<stdio.h>
int getSquare(int x );   //Function prototype
int main()   //Calling Function
{
        int a, b;
        a = 4;
        b = getSquare(a );   //Function Call
        printf("\nSquare = %d",b);
        return 0;
}

int getSquare(int x )   //Called Function
{
        int a;
        a = x * x;
        return a;
}
```

Output:

```
Square = 16
```

User Defined Data Types:

These data types are created by the programmer as per application requirement and need. These data types also inherit the properties of basic data types only.

Examples:

1. Structures - struct
2. Unions – union

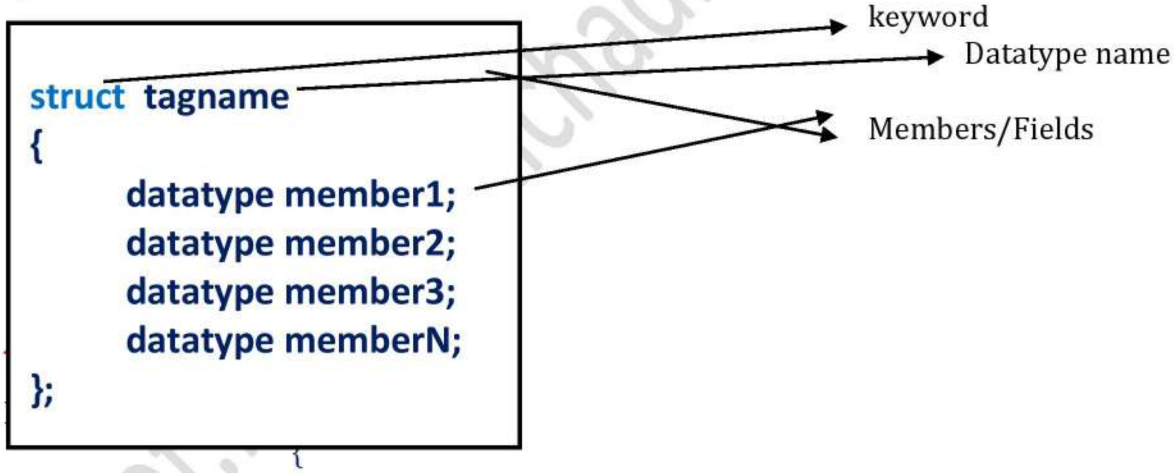
Structures – Definition:

Structure in c is a user-defined data type that enables us to store the collection of different data types. Each element of a structure is called a member. Structure is a user defined data type; it represents both homogenous and heterogeneous elements in it.

Structure type creation/Structure Definition/ Structure Data Type creation.

Using **struct** keyword, we can create structure data type in C.

Syntax:



```
struct  tagname
{
        datatype member1;
        datatype member2;
        datatype member3;
        datatype memberN;
};
```

```
                {
                        int rollno;
                        char name[25];
                };
```

- In this example Student is structure data type and rollno & name are the members of Student type.
- Structure type must be defined outside function bodies and after preprocessor directives.
- When we create a structure variable then members of the structure will be allocated its memory.

Declaring Structure Variables:

- After creating a structure data type then we can create structure variables.
- Using below syntax we can create structure variables in C.

```
struct tagname variablename;
```

We can declare a variable for the structure so that we can access the member of the structure easily. There are two ways to declare structure variable:

1. By struct keyword within main() function
2. By declaring a variable at the time of defining the structure.

1. within main:

```
struct employee
{   int id;
    char name[50];
    float salary;
};

int main( )
{
        struct employee e1, e2;
}
```

The variables e1 and e2 can be used to access the values stored in the structure. Here, e1 and e2 has all the members of employee structure.

2. By declaring a variable at the time of defining the structure.

```
struct employee
{   int id;
    char name[50];
    float salary;
}e1,e2;
```

Ex1:

Structure data type creation:

```
struct Student
{
        int id;
        char name[25];
};
```

Structure variable creation:

```
struct Student s1, s2, s3;
```

Accessing Structure Members:

There are two ways to access structure members:

1. By . (Member or dot operator) : structurevariablename.membername
2. By -> (structure pointer operator): structpointervariable->membername

1. Example Program:

```
#include<stdio.h>
 struct Employee
{
        int id;
        double salary;
};

 int main()
{
        struct Employee e1,e2;

        e1.id = 101;
        e1.salary = 20000.00;

        e2.id = 102;
        e2.salary = 30000.00;

        printf("\nEmployee 1 details:\n");
        printf("\nID : %d",e1.id);
        printf("\nSalary : %f",e1.salary);

        printf("\n\nEmployee 2 details:\n");
        printf("\nID : %d",e2.id);
        printf("\nSalary : %f",e2.salary);
        return 0;
}
```

Output:

```
Employee 1 details:

ID : 101
ID : 20000.000000

Employee 2 details:

ID : 102
ID : 30000.000000
```

Structure members initialization:

1. Using Initializer Way:

- members cab be initialized at the time of structure variable creation.

Ex:

```
struct Employee
{
        int id;
        double salary;
};
int main( )
{
        struct Employee e1 = {121, 5000.00};
        printf("\nID : %d",e2.id);
        printf("\nSalary : %f",e2.salary);
        return 0;
}
```

2. After declaring structure variable and initializing members wise:

Ex1:

```
#include<stdio.h>
 struct Employee
{
        int id;
        double salary;
 };

 int main()
```

```
{
        struct Employee e1;

        e1.id = 101;
        e1.salary = 20000.00;

        printf("\nEmployee 1 details:\n");
        printf("\nID : %d",e1.id);
        printf("\nSalary : %f",e1.salary);

        return 0;
}
```

Output:

```
Employee 1 details:

ID : 101
Salary : 20000.000000
```

Comparing Structure Variables:

- When two structure variables are equal when both members-wise values are equal, then we can say both structure variables are same.
- We cannot compare two structure variables directly, instead we to compare member-wise.
- It is not possible to compare two structure variables using == and != operator. But we can compare individual members.
- When we assign a structure variable another structure then both occupies same values and both are equal.

Ex1:

```
#include<stdio.h>
 struct Employee
{
        int id;
        double salary;
};

 int main( )
{
        struct Employee e1,e2,e3;

        e1.id= 11;
        e2.id = 11;
        e1.salary = 12000;
        e2.salary = 12000;

        if(e1.id==e2.id && e1.salary==e2.salary)
        {
                printf("\nBoth structures are equal");
        }
        else
        {
                printf("\nBoth structures are not equal");
        }

        return 0;
}
```

Output:

```
Both structures are equal
```

Array of Structures:

An array of structures is simply an array in which each element is a structure of the same type.
Every element of the array is a structure of the same type.

Scenario: whenever we want to store one student information, we need to define a structure student with the following members, id, name.

```
struct Employee
{
        int id;
        char name[10];
        float salary;
};
```

Using above structure Student definition, we can create variables and arrays of Student.

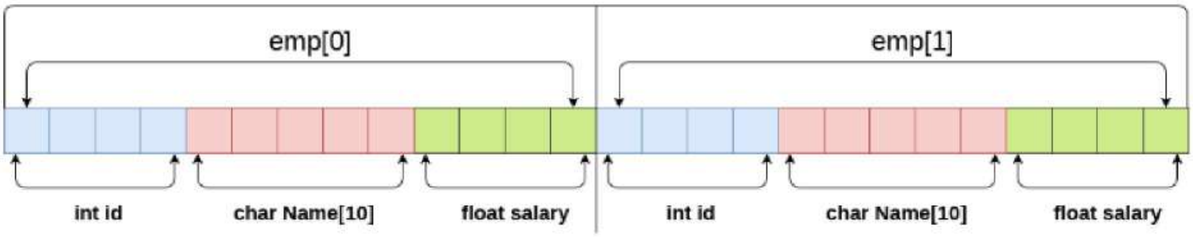
```
        struct Employee e1;
```

But using s1 variable at max we can store only one student's data, if we want to store n number of students data then we must go for array of Student.

```
        struct Employee e[2];
```

e is an array of employee of 2 size.
Every element of emp array is one Employee Data.

Array of structures



Example Program:

Program to demonstrate student structure to read & display records of n students.

```
#include<stdio.h>
 struct Student
 {
        int id;
        char name[10];
 };

 int main()
 {
        struct Student s[10];
        int n,i;

        printf("\nEnter Number of Students : ");
        scanf("%d",&n);

        for(i=0; i<n; i++)
        {
                printf("\n\nEnter %d Student Details...",i+1);
                printf("\nEnter Id : ");
                scanf("%d",&s[i].id);
                printf("\nEnter Name : ");
                fflush(stdin);
                gets(s[i].name);
        }
        printf("\n****Students Data*****\n");
        for(i=0; i<n; i++)
        {
                printf("\n\nStudent %d Data....",i+1);
                printf("\nId = %d",s[i].id);
                printf("\nName = %s",s[i].name);
        }
        return 0;
}
```

Output:

```
Enter Number of Students : 2

Enter 1 Student Details...
Enter Id : 11
Enter Name : Raju

Enter 2 Student Details...
Enter Id : 12
Enter Name : Ravi

****Students Data*****

Student 1 Data....
Id = 11
Name = Raju

Student 2 Data....
Id = 12
Name = Ravi
```

Unions - Union definition:

Union can be defined as a user-defined data type which is a collection of different variables of different data types in the same memory location. The union can also be defined as many members, but only one member can contain a value at a particular point in time.

- At a time we can store only value to the member and that too store in larger memory member.

Note: Total size of the union is equals to the member having larger size.

Union type creation and union variable creation:

Union data type creation:

```
union Student
{
        int id;
        char name[25];
};
```

Union variable creation:

union Student s1;

Example Program:

```
#include<stdio.h>
union Employee
{
        int id;
        double salary;
};
int main()
{
        union Employee e1;
        e1.id = 101;
        printf("\nId = %d",e1.id);
        e1.salary = 25000;
        printf("\nSalary = %lf",e1.salary);
        printf("\nId = %d",e1.id);
        return 0;
}
```

Output:

```
Id = 101 Salary = 25000.000000 Id = 0
```

Difference Between Structures and Unions.

Parameters	**Structure**	**Union**
Keyword	struct	union
Member Access	All members at a time	Only one member at a time
Memory Sharing	All members share separate memory locations	All members share larger sized member memory location.
Total Memory Size	Total Size = Sum of size of all members	Total Size = A member having larger size
Affect after altering member value	No affect	Affects remaining members.
When to use	Memory is not a matter	Memory matters

Ex	struct emp
	{
	int emp_no;
	char emp_name[20];
	};
	
	struct emp e1;
		union emp
		{
		int emp_no;
		char emp_name[20];
		};
		
		union emp e1;

Write a C program to differentiate structure & union.

```
#include<stdio.h>
union Employee_un
{
        int id;
        double salary;
};
struct Employee_st
{
        int id;
        double salary;
};
int main()
{
        union Employee_un e1;
        struct Employee_st e2;

        printf("\nSize of Union : %d",sizeof(e1));
        printf("\nSize of Structure : %d",sizeof(e2));

        return 0;
}
```

Output:

```
Size of Union : 8
Size of Structure : 16
```

Advantages of Structures:

Heterogeneous collection of data items: structure allows us to create user defined data-type which can store items with different data types.

Reduced complexity:

Increased productivity: structure in C eliminates lots of burden while dealing with records which contain heterogeneous data items.

Maintainability of code: using structure, we represent complex records by using a single name, which makes code maintainability like a breeze.

Enhanced code readability:

Suitable for some mathematical operations:

UNIT-IV QUESTION BANK

2 MARKS

1. Define structure.
2. Name different types of functions.
3. What is string?.
4. Define structure & union.
5. Define function. Mention the types of functions.
6. Differentiate structure & union.
7. What are arguments?
8. Define function? State 2 advantages.
9. What is structure? How to define it.
10. What is union? How to define it.

4/5 MARKS

1. Explain syntax declarations of functions .
2. Write a C program to find the given number is Prime using function.
3. Write a C program using structure to store the record of n students.
4. Write the advantages of functions.
5. How do you call user defined function? Explain with an example.
6. Explain function with no-argument but return value.
7. Explain the general format of user defined functions.
8. How do you access members of the structure? Explain with example.
9. Differentiate array & structure.
10. Write a program to differentiate structure and union.
11. Write a C program to sort array elements.
12. Differentiate structure and union.
13. Explain types of user defined functions.
14. What is structure? How to declare it? Explain with an example.
15. How structures are different from union? Explain.

Note: Practice all Lab programs, 4 to 5 programs will be asked for main theory exams also.
Kindly refer text book and previous year question papers for exams.
Use Below link to find Previous Year Question Papers:

CP Question Papers:
https://drive.google.com/drive/folders/1ugmW0wmGYl76yuEn2vWXjgqDvEt_u1aO?usp=sharing

All BCA Question Papers:
https://drive.google.com/drive/folders/1F0CEAtoFmji9Gx9snSGZYqEksw4VkjSW