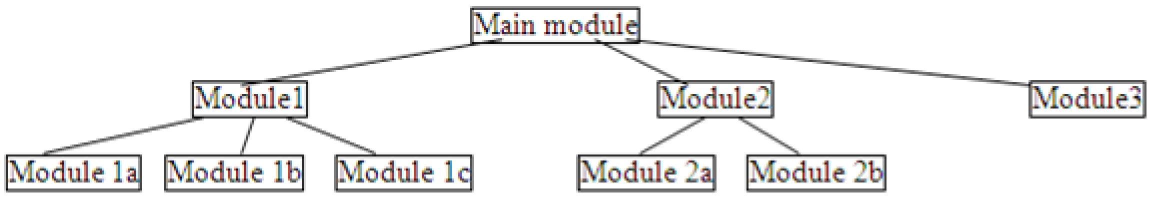
UNIT IV

Functions: Intro to Functions, Function Declaration and definition, Function call return Types, Recursion, and Arguments, Argument passing, Arrays as Parameters, Scope of variables, User Defined datatypes-Structures, unions.

Functions

Designing

- Breaking a complex problem into smaller parts is a common practice in developing software.
- These parts of a program are called as modules and the process of subdividing a problem into parts is called **modular programming**.
- In this approach a program is divided into a main module and its related modules. Each module can be subdivided into sub modules.
- This design can be represented using a structure chart.

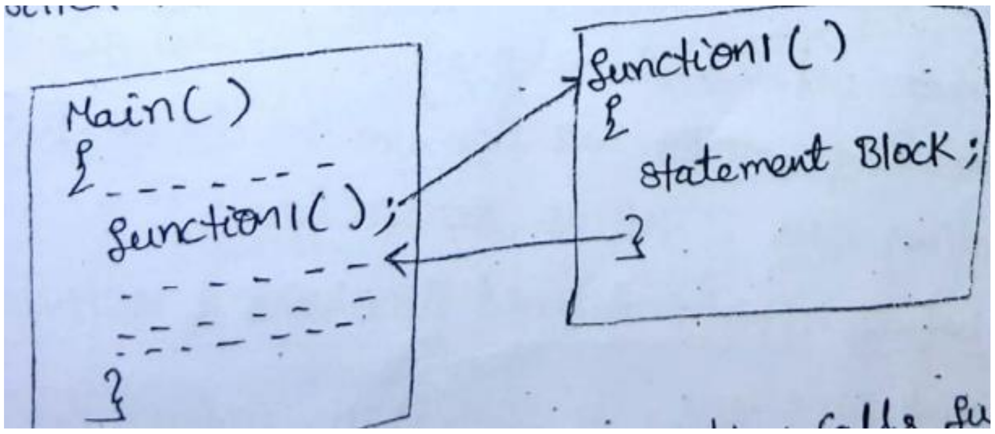


Function in C

- In C the idea of modular programming is implemented using **functions.**
- A C program is made of one or more functions one of which must be named as main.
- The execution of the program always starts from main. But it can call other functions to do special tasks.

Function – Defintion

- A function is a self – contained block or a sub program which contains a group of statements that performs a required task when called.
- A called function receives control from a calling function. When the called function completes its task, it returns control to the calling function.



- From the above figure we can see that main() calls function named function1(). Main() is known as calling function and function1() is known as called funciton.

Advantages of functions

- Functions provide a way to **reuse the code**. Function once written can be used many times.
- Using functions data can be **protected** since data used in the function cannot be used by outside of the function.
- Using functions, large programs can be reduced to smaller ones and it is **easy to find any errors** in the program.

Types of functions

- Functions are classified into two types they are:
  1. User defined functions
  2. Library or built in functions

User defined functions

- User defined functions are defined by the user.
- To use a function in C it must be both **declared** and **defined.** A function name is used three times:

For declaration: A function **declaration** tells the compiler about a function's name, return type, and parameters.

In a function call: Call any function from main.

For definition: A function **definition** provides the actual body of the function

For Function declaration

- The function declaration is done before the function call, gives the whole picture of the function. It contains only function header.

Syntax: **returntype functionname(type1,type2,…..) ;**

Example: **void greeting(void);**

- Above function declaration specifies that program contains a function with name **greeting** and it returns nothing and it has no arguments. (void means nothing).

int add(int, int);

- Above function declaration specifies add is the name of function and it returns an int value and accepts 2 int values also known as arguments.
- Variable names are not required in the function declaration.

The function definition

- The function definition contains the code needed to complete the task. It is made up of two parts. They are
  - The function header
  - The function body
- The general form of a function definition is
-

```
return_type function_name( parameter list )
{
  body of the function
}
```

- **A function header** contains three parts: the return type, the name of function, and the formal parameter list.
- A **semicolon is not used** at the end of function definition header.
- The **return_type** is the data type of the value the function returns.
- Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name** is the actual name of the function.
- **Parameters** are like a placeholder. When a function is invoked, you pass a value to the parameter. They are of two types
    1. Formal Parameters
    2. Actual Parameters

Formal and actual parameters

- **Formal parameters** are the variables that are declared in the header of the function definition.
- These are also called as dummy parameters since they don't have existence until the function is called.
- **Actual parameters** are used in the calling function.
- Formal and actual parameters must match exactly in type, order, and number of parameters
- The **function body** contains a collection of statements that defines what the function does. It contains the local declarations and statements.

The function call

- **A function can be called** once it is defined. Function call has highest precedence in C.
- A function call contains the name of function following the list of arguments enclosed between pair of parenthesize ().
- The parameters in function call are called as actual parameters and the values of these parameters are sent to the called function.
- There are many different ways to call a function.
    - greeting(); // no arguments
    - c = add(a,b) + 10 ; // function call is used in an expression

```
#include<stdio.h>
double average(int, int); //function declaration
main()
{
	int a=10,b=20;
	float res;
```

```
            res = average(a,b); // x,y are actual parameters //function call
            printf("result is %f",res);
}
double average(int x, int y)  // x,y are formal parameters // called funciton
{
            float sum;
             sum=x+y;                        //function body
            return (sum/2.0);
}
```

Output:

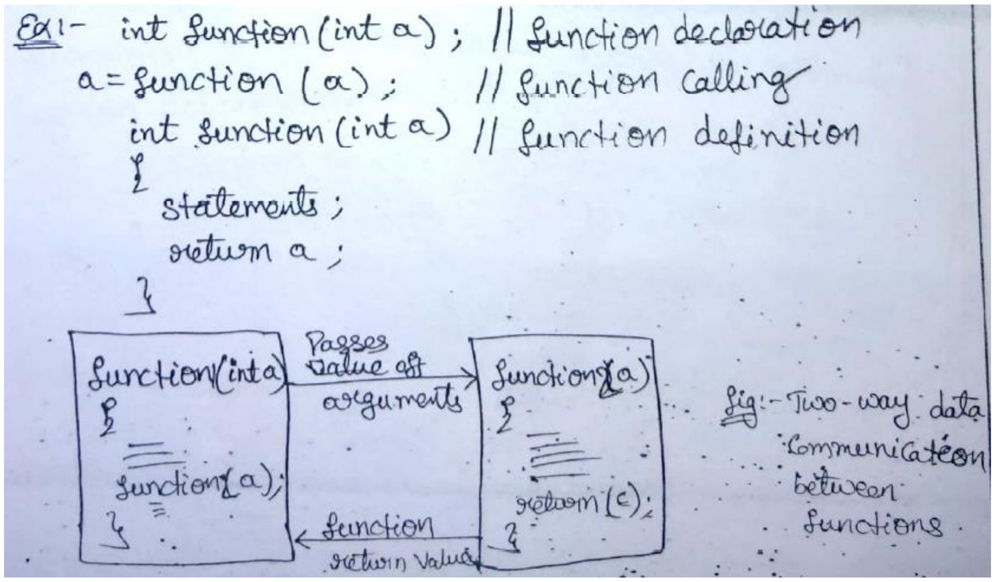
```
result is 15.000000
```

Categories of functions

- All C functions can be called a either with argument or without arguments in a C program. These functions may or may not return values to the calling functions.
- Functions may belongs to one of the following categories:
    1. Function with arguments and with return value
    2. Function with arguments and without return value
    3. Function without arguments and without return value
    4. Function without argument and with return value

Function with arguments and with return value

- These function having arguments and that argument are passed to the definition of a function such function will have two-way data communication i.e., here the data is transferred between calling and called function



```
Ex:-  int function (int a) ;   // function declaration
   a = function (a) ;         // function calling
      int function (int a)   // function definition
      {
        statements ;
        return a ;
      }
```

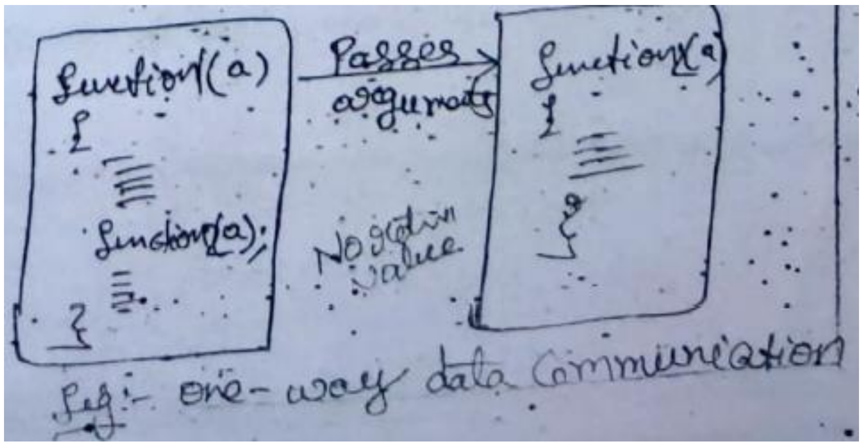
function(int a)
{
function(a);
}
Passes value of arguments
function(a)
{
return (c);
}
function return value
fig:- Two-way data communication between functions.

```
int  add(int,int); // function declaration
main()
{
     int a,b, ans;
    printf("enter 2 values  ");
    scanf("%d%d",&a,&b);
    ans = add(a,b);  // function is called in an expression, a,b are actual parameters
 }
int add(int x,int y) // function definition, x,y are formal parameters
{
   return x+y;   // x+y will be returned to called function
  }
```

Function with arguments and without return value

- Arguments are passed to the calling function the called function operates on those values but no result is sent back to the calling function such a functions are partially dependent on the calling function.

```
void  add(int,int); // function declaration
main()
{
    int a,b;
    printf("enter 2 values  ");
    scanf("%d%d",&a,&b);
    add(a,b);  // function call, a,b are actual parameters
 }
void add(int x,int y) // function definition, x,y are formal parameters
{
   int c; //local variable
  c=x+y;
  printf("\n sum = %d ",c);
}
```

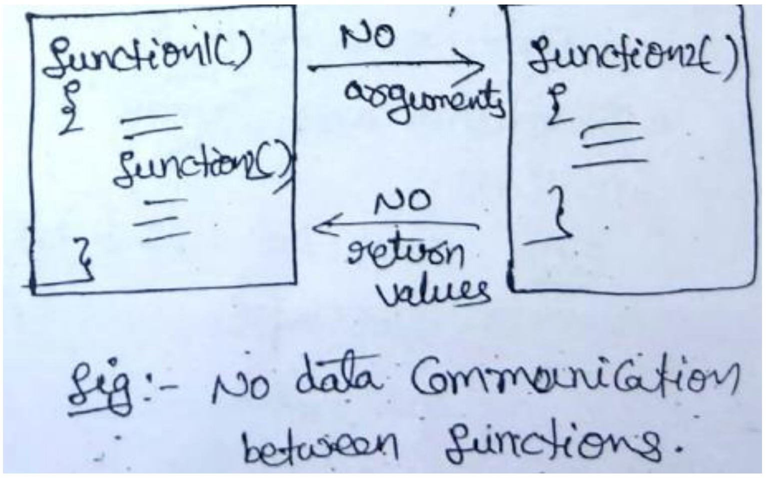


Function without arguments and without return value

- When a function has no arguments it does not receive any data from calling function. Similarly, when it does not return a value the calling function does not receive any data from the called function.
- There is no data transfer between the calling function and they called function

```
      #include<stdio.h>
      sum();
      main()
      {
            sum();
      }
sum()
{
     int a,b;
```

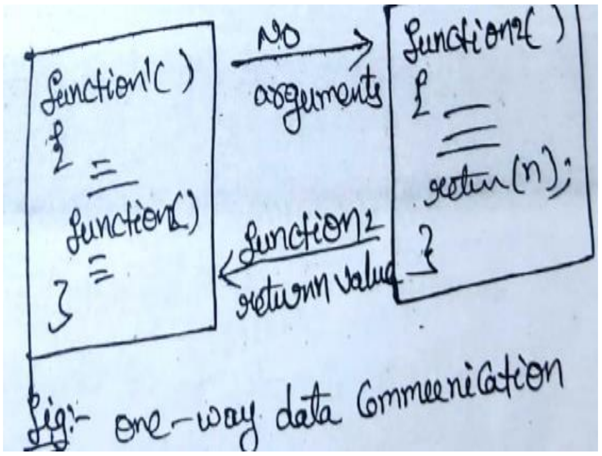
```
    printf(“enter 2 values  “);
    scanf(“%d%d”,&a,&b);
   c=a+b;
   printf(“%d”,c);
}
```



Function without argument and with return value

- The function having no arguments but return a value to the calling function.

```
int  add(); // function declaration
main()
{
   int  ans;
   ans = add();  // function is called in an expression, no parameters
   printf(“%d”,ans);
 }
int add() // function definition
{   int a,b; // local variables
    printf(“enter 2 values  “);
    scanf(“%d%d”,&a,&b);
   return (a+b);
 }
```



Standard library functions

- Library functions or predefined set of functions, their task is limited.

- A user cannot understand the internal working of these functions. The user can only use the functions but cannot change or modify them.

Example: sqrt(81) - 9

- Pre-defined functions that are provide to perform such as input output functions are printf(), scanf(), puts(), gets().
- Mathematical functions are pow(), sqrt() etc.

Eg libraries are

stdio.h which contains standard input and output functions like printf(), scanf() etc.

stdlib.h which contains standard library functions like rand(), exit() etc.

math.h which contains mathematical functions like pow(), sin(), cos() etc.

string.h which contains string manipulation functions like strlen(), strcat() etc.

Returning a value

- Functions having two different types to returns
  - With return value
  - Without return value
- The function may or maynot send back any value to the calling function, if it is does it is done through the 'return' keyword.

Syntax: return(expression); (or) return expression;

- The expression should palced in between paranthesis.

Example: return(a+b); or returnc;

Parameter Passing Techiques

- There are two ways in which arguments or parameters can be passed to the called function. They are:
  - Call by value / pass by value
  - Call by reference / pass by reference

Call by value / pass by value

- In this call by value techique the value of each of the actual arguments in the calling function is copied into corresponding formal arguments of the called function.
- With this technique the changes made to the formal arguments in the called function have no effect on the values of actual argument in the calling function.

```
#include<stdio.h>
void swap (int ,int); // declaration
void main()
{
  int a,b;
printf("enter 2 values\n"  );
scanf("%d%d",&a,&b);
printf("\nvalues before swap are\t" );
printf("a = %d b= %d ",a,b);
swap(a,b);  // function call, values of a,b are passed, call by value
printf("\nvalues after swap are\t");
printf("a = %d b= %d",a,b);
}
void swap(int a, int b)
{
  int   t;
 t=a;
a=b;
b=t;
printf("\nvalues after swap are\t");
printf("a = %d b= %d",a,b);
}
```

Output:

```
enter 2 values
10 20

values before swap are        a = 10 b= 20
values after swap are         a = 20 b= 10
values after swap are         a = 10 b= 20
```

In the above program function swap contains 2 arguments. Even though inside the function these arguments are swapped it is not affected in the main program.

Call by reference / pass by reference

- In this type of parameter passing technique address of variables are passed to the function. To hold these addresses the formal parameters must be pointer variables.
- If any change is taking place for an argument in this type that change will be affected in the called function also.

```
#include<stdio.h>
void swap (int *,int *); // declaration
void main()
{
                                                                 int a,b;
printf("enter 2 values\n"  );
scanf("%d%d”,&a,&b);
printf("\nvalues before swap are\t" );
printf("a = %d b= %d ",a,b);
swap(&a,&b);  // function call, values of a,b are passed, call by value
printf("\nvalues after swap are\t");
printf("a = %d b= %d",a,b);
}
void swap(int *a, int *b)
{
                                                              int   t;
 t=*a;
*a=*b;
*b=t;

}
```

Output:

```
enter 2 values
10 20

values before swap are        a = 10 b= 20
values after swap are         a = 20 b= 10
```

- In the above program function swap contains 2 arguments and these are pointer variables.
- Inside the function these arguments are swapped, since pointers are used it is affected in the main program also. That is they will swap in the main.

Recursion

- Recursion is the process in which a function repeatedly calls itself to perform calculations.

Syntax:

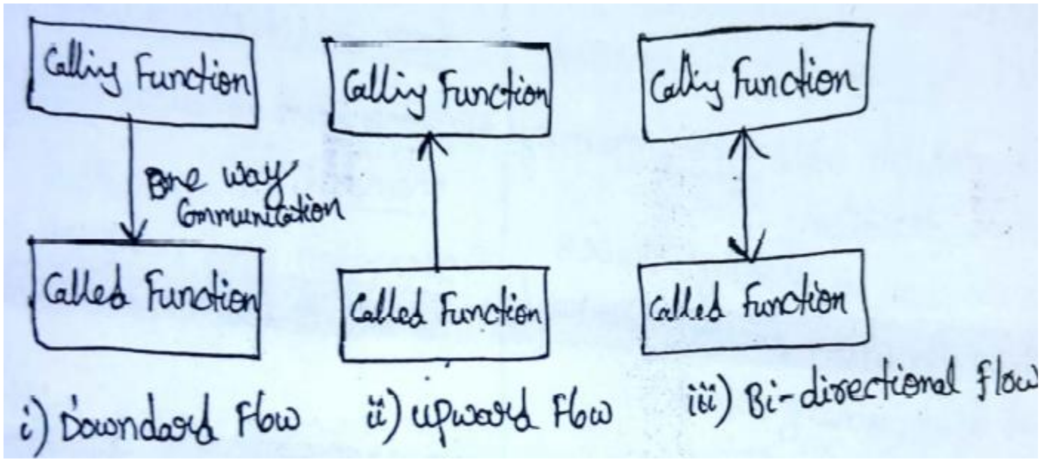
```
recursion()
        {
                recursion();
        }
        int main()
        {
                ……….
                recursion();
                …………
                return 0;
        }
```

```
#include <stdio.h>
int fib(int); // function declaration
```

```
main()
{
    int n,i;
    printf("\nEnter n ");
    scanf("%d",&n);
    for (i=0;i<n;i++)
        printf("%d\t",fib(i));
 }
  int fib(int n)
  {
    if (n==0 || n==1)
      return n;
    return fib(n-1)+fib(n-2);
  }
```

Inter- Function Communication

- The calling and called function are two separate entities, they need to communicate to exchange data.
- The data flow between calling and called functions can be divided into three strategies:
    1. Downward flow
    2. Upward flow
    3. Bi-directional flow



Downward flow

- In downward flow communication, the calling function sends data to the called function.
- No data flow in the opposite direction.

Example:

```
sum();
main()
{
        sum();
}
sum()
{
        int a=10,b=20,c;
        c=a+b;
        printf("%d\n",c);
}
```

Upward flow

- Upward flow communication occurs when called function sends data back to the called function without receiving any data from it.

```
void swap (int ,int); // declaration
main()
{
        int a,b;
printf("enter 2 values  ");
scanf("%d%d",&a,&b);
printf("\nvalues before swap are ");
printf("a = %d b= %d ",a,b);
swap(&a,&b);  // function call, values of a,b are passed, call by value
printf("\nvalues after swap are ");
printf("a = %d b= %d ",a,b);
}
void swap(int *a, int *b)
{
         int   t;
 t=*a;
*a=*b;
*b=t;
}
```

Bi-directional flow

- Bi-directional flow occurs when the calling function sends data down to the called function.

```
#include<stdio.h>
int sum(int a, int b);
main()
{
        int a=10,b=20,p;
        p=sum();
        printf("%d\n",p);
}
sum()
{
        int c;
        c=a+b;
        return c;
}
```

Passing Array to Functions/Arrays as parameters

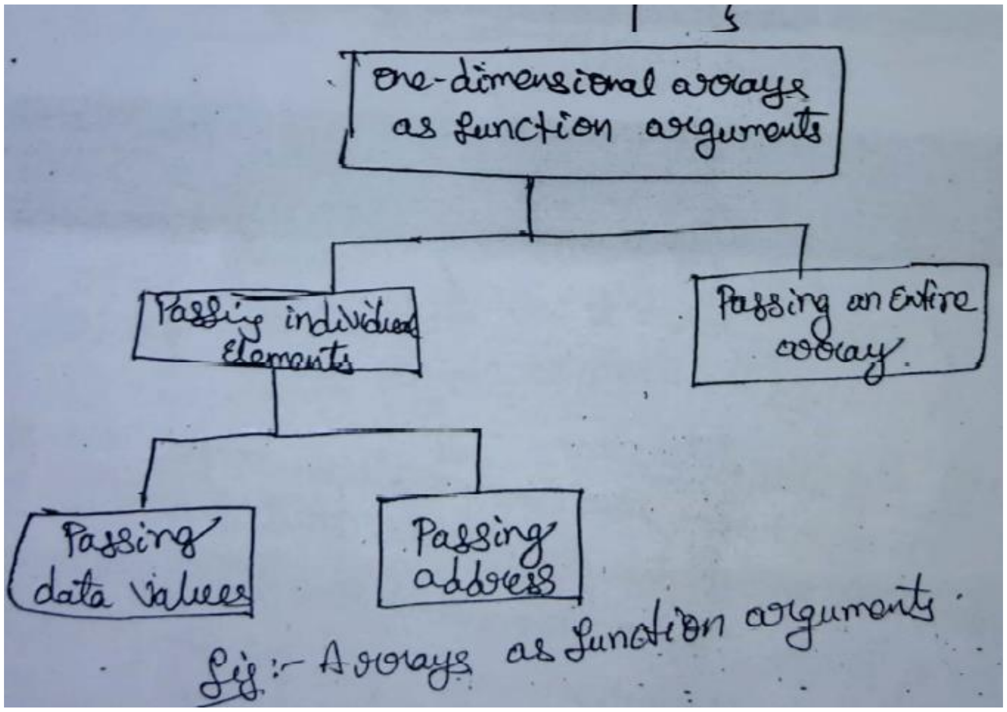
- Array can pass through calling function, where as we don't return an array from function.
- Rather we return a pointer holding the base address of the array to be returned. But, we must make sure that the array exits after the function ends.

1. Formal parameter as a pointer as follows:

```
void myfunction(int *p)
{
        ……
        …..
}
```

2. Formal parameter as a sized array as follows:

```
myfunctio(int p[10])                    myfunctio(int p[ ])
{                                       {
        …….                                     ……..
        …….                                     …….
}                                       }
```



One-dimensional arrays as function arguments
Passing individual Elements
Passing an Entire array.
Passing data values
Passing address

fig :- Arrays as function arguments

Passing Individual Elements

- Individual elements are passed to functions by simply including them as subscripted vairables in the function call argument list.

Passing data values

```
 #include<stdio.h>
fun(int arr[]);
main()
{
        int arr[5]={1,2,3,4,5};
        fun(arr[3]);
}
fun(int num)
{
        printf("%d",num);
}
```

Passing address

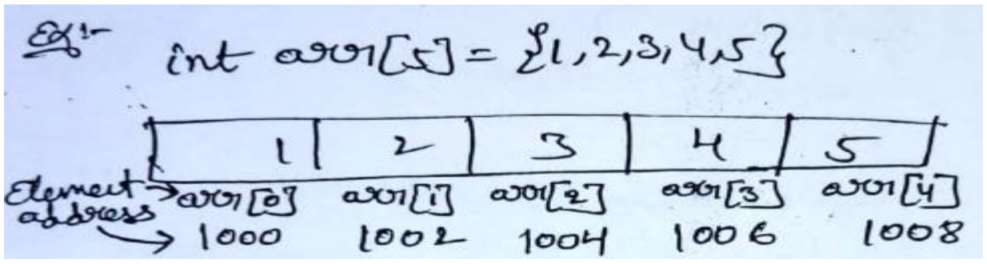
```
#include<stdio.h>
fun(int *n);
main()
{
        int arr[5]={1,2,3,4,5};
        fun(&arr[3]);
}
fun(int *n)
{
        printf("%d",*n);
}
```

Passing the Entire array

```
#include<stdio.h>
fun(int arr[5]);
main()
{
        int arr[5]={1,2,3,4,5};
        fun(arr);
}
fun(int arr[5])
{
        int i;
        for(i=0;i<5;i++)
                printf("%d",arr[i]);
}
```

Passing pointer to function

- Function in a program occupies memory. The name of the function is a pointer constant to its first byte of memory.



int *p;

p=&arr; or p=&arr[0];

```
#include<stdio.h>
main()
{
        int i;
        int a[5]={1,2,3,4,5};
        int *p=&a;
        for(i=0;i<5;i++)
                printf("%d",*p);
}
```

Scope of Variables:

Scope of variables in C language refers to the region or part of the code where a variable can be accessed or used. It defines the visibility of a variable within different parts of a program.

For example:

- If a variable is declared inside a function, it can only be used within that function, and this is called local scope.
- If a variable is declared outside of all functions, it can be accessed from anywhere in the program, and this is known as global scope.
- Simply put, scope determines where in the program a variable is available for use.

Types of Scope of Variables in C

Variables in C programming have different scopes depending on where and how they are declared. Learning these scopes is essential for managing variable visibility and lifetime throughout the program. Hence, let’s discuss various scope of variables in C language:

1. Local Scope

A variable has local scope when it is declared inside a function or block (within {} braces). This means that the variable is accessible only within that function or block and is not visible to other parts of the program.

- **Function-Level Scope**: Variables declared inside a function are accessible only within that function.
- **Block-Level Scope:** Variables declared inside a block (such as within an if statement or a loop) are accessible only within that block.

Example:

```
void exampleFunction() {
   int localVar = 10;  // Function-level local variable
   if (localVar == 10) {
       int blockVar = 20;  // Block-level local variable
       printf("blockVar: %d\n", blockVar);  // Accessible here
   }
   // printf("%d", blockVar);  // Error: blockVar is not accessible here
   printf("localVar: %d\n", localVar);  // Accessible within the whole function
}
```
Here:

- localVar has function-level scope and is accessible anywhere inside exampleFunction().
- blockVar has block-level scope and is only accessible inside the if block.

2. Global Scope

A variable has global scope when it is declared outside of all functions. This means the variable is accessible from any function in the program.

Global variables remain in memory for the entire duration of the program and can be accessed by any function after their declaration.

Example:

```
int globalVar = 100;  // Global variable with global scope

void function1() {
   printf("Global variable in function1: %d\n", globalVar);
}
```

```
void function2() {
    globalVar = 200;  // Modifying global variable
    printf("Global variable in function2: %d\n", globalVar);
}

int main() {
    function1();  // Accesses globalVar
    function2();  // Modifies and accesses globalVar
    return 0;
}
```

Here:

- globalVar is a global variable and can be accessed or modified by any function in the program.

3. Function Scope (Function Parameters)

Function parameters have function scope, meaning they are only accessible within the body of that function. These parameters are treated like local variables but are initialized with values passed when the function is called.

Example

```
void multiply(int a, int b) {  // Function parameters a and b
    int result = a * b;
    printf("Result: %d\n", result);
}

int main() {
    multiply(5, 3);  // Passing values to function parameters
    return 0;
}
```

Here:

- a and b are parameters with function scope and can only be accessed within the multiply() function.

4. File Scope

Variables with file scope are those declared at the global level, but their visibility can be restricted to the file in which they are declared. This is typically done using the static keyword.

Additionally, the extern keyword can be used to make global variables accessible across multiple files.

- **Static Variables (File Scope):** Variables declared as static outside of all functions have file scope. This means they are only visible within the file where they are defined and cannot be accessed by functions in other files.

- **Extern Variables:** The extern keyword allows a global variable to be shared across multiple files. The variable is defined in one file and declared using extern in other files to access the same variable.

Lifetime of Variables in C

Variable lifetime in C refers to the period during which a variable exists in memory and retains its value. It defines how long the variable's allocated memory is valid and available for use in the program. The lifetime of a variable depends on where and how the variable is declared.

Types of Variable Lifetime in C Language

Variables in C have different lifetimes depending on how they are declared. Below are the main types of variable lifetime in C:

1. Automatic (Local) Variables

Automatic variables are local variables that are automatically created when a function is called and destroyed when the function returns. These variables are created using the auto keyword (though it's implicit and rarely used explicitly).

Automatic variables only exist during the function's execution, and their values do not persist between function calls.

Characteristics:

- Created when the function starts.
- Destroyed when the function ends.
- They do not retain their values between function calls.
- Stored in the stack.

Example:

```
#include <stdio.h>

void exampleFunction() {
    int localVar = 10;  // Automatic variable
    printf("Local Variable: %d\n", localVar);
}  // localVar is destroyed here

int main() {
    exampleFunction();  // localVar is created and destroyed during the function call
    return 0;
}
```

Explanation:

In the example, localVar is an automatic variable that is created when exampleFunction() is called and destroyed when the function returns.

2. Static Variables

Static variables are variables that retain their value between multiple function calls. If a static variable is declared inside a function, it is initialized only once, and its value persists across function calls.

Static variables can be declared at the global level as well, in which case they are only visible within the file they are declared in.

Characteristics:

- Created when the program starts.
- Retain their value across multiple function calls.
- Destroyed when the program ends.
- Stored in the data segment of memory.

Example

```
#include <stdio.h>


void countCalls() {
    static int count = 0;  // Static variable initialized once
    count++;
    printf("Function called %d times\n", count);
}


int main() {
    countCalls();  // Function called 1st time
    countCalls();  // Function called 2nd time
    countCalls();  // Function called 3rd time
    return 0;
}
```

Explanation:

In the example, the count variable is a static variable. It is initialized only once, and its value is retained across multiple calls to countCalls(). Each time the function is called, count is incremented, and its value persists.

3. Global Variables

Global variables are variables declared outside all functions, making them accessible from any function in the program. They have a lifetime that spans the entire duration of the program.

Global variables are created when the program starts and destroyed when the program ends. Since they are available throughout the program, they should be used carefully to avoid unintended modifications.

Characteristics:

- Created when the program starts.

- Accessible from any function.
- Destroyed when the program ends.
- Stored in the data segment.

Example

```
#include <stdio.h>


int globalVar = 100;  // Global variable


void function1() {
    printf("Global Variable in function1: %d\n", globalVar);
}


void function2() {
    globalVar = 200;  // Modify global variable
    printf("Global Variable in function2: %d\n", globalVar);
}


int main() {
    function1();  // Access globalVar
    function2();  // Modify and access globalVar
    return 0;
}
```

Explanation:

In this example, globalVar is a global variable. It is accessible in both function1() and function2(), and its value can be modified by any function. The variable exists for the entire duration of the program.

4. Dynamic Variables

Dynamic variables are variables that are created and destroyed at runtime using dynamic memory allocation functions like malloc(), calloc(), and free(). These variables do not have a fixed lifetime; instead, they exist for as long as they are explicitly managed by the programmer.

Dynamic memory is stored on the heap, and the programmer is responsible for releasing the memory using free() once the variable is no longer needed.

Characteristics:

- Created dynamically at runtime using malloc() or calloc().
- Persist until manually deallocated with free().
- Stored in the heap memory.

Example:

```
#include <stdio.h>

#include <stdlib.h>  // For malloc and free



int main() {

    int* dynamicVar = (int*)malloc(sizeof(int));  // Dynamically allocated variable

    if (dynamicVar == NULL) {

        printf("Memory allocation failed\n");

        return 1;

    }

    *dynamicVar = 50;  // Assign value to dynamically allocated memory

    printf("Dynamic Variable: %d\n", *dynamicVar);



    free(dynamicVar);  // Deallocate memory

    return 0;

}
```

Explanation:

In this example, memory is allocated dynamically for dynamicVar using malloc(). The memory persists until free(dynamicVar) is called, at which point the allocated memory is released. If the memory is not freed, it can lead to memory leaks.

Difference Between Scope and Lifetime of Variables in C

Below is a comparison of the difference between scope visibility and lifetime of variables in C:

Aspect	**Scope of a Variable**	**Lifetime of a Variable**

Definition	Scope refers to the region of the program where the variable is visible and can be accessed.	Lifetime refers to the duration for which the variable exists in memory during program execution.
Focus	Focuses on where the variable can be accessed.	Focuses on how long the variable occupies memory.
Types	Local scope (function/block), global scope, function scope, file scope.	Automatic (local), static, global, dynamic (heap).
Start	The scope of a variable starts when it is declared.	The lifetime of a variable starts when it is allocated in memory.
End	The scope ends when the block or function where the variable is declared ends.	The lifetime ends when the variable is deallocated or the program terminates.
Example (Local)	A local variable in a function is only accessible inside that function.	A local variable is created when the function is called and destroyed when the function ends.
Example (Global)	A global variable is accessible throughout the entire program.	A global variable persists for the entire program’s duration.
Influence on Access	Scope determines which part of the program can access the variable.	Lifetime determines when the variable is available in memory and can be accessed

STRUCTURES

Arrays are used for storing a group of SIMILAR data items. In order to store a group of data items, we need structures. Structure is a constructed data type for packing different types of data that are logically related. The structure is analogous to the “record” of a database. Structures are used for organizing complex data in a simple and meaningful way.

Example for structures:

Student	:	regno, student_name, age, address
Book	:	bookid, bookname, author, price, edition, publisher, year Employee
	:	employeeid, employee_name, age, sex, dateofbirth, basicpay
Customer	:	custid, cust_name, cust_address, cust_phone

Structure Definition

Structures are defined first and then it is used for declaring structure variables. Let us see how to define a structure using simple example given below:

```
struct book
{
        int bookid;
        char bookname[20];char
        author[20]; float price;
        int year; int
        pages;
        char publisher[25];
};
```

The keyword “struct” is used for declaring a structure. In this example, book is the name of the structure or the structure tag that is defined by the struct keyword. The book structure has six fields and they are known as structure elements or structure members. Remember each structure member may be of a different data type. The structure tag name or the structure name can be used to declare variables of the structure data type.

The syntax for structure definition is given below:

```
struct tagname

{

        Data_type   member1;

        Data_type   member2;

        …………….

        ……………

};
```

Note:

1. To mark the completion of the template, semicolon is used at the end of thetemplate.
2. Each structure member is declared in a separate line.

Declaring Structure Variables

First, the structure format is defined. Then the variables can be declared of that structure type. A structure can be declared in the same way as the variables are declared. There are two ways for declaring a structure variable.

1) Declaration of structure variable at the time of defining the structure (i.e structure definition and structure variable declaration are combined)

```
struct book
{
                int bookid;
                char bookname[20];char
                author[20]; float price;
                int year; int
                pages;
                char publisher[25];
```

```
        }   b1,b2,b3;
```

The b1, b2, and b3 are structure variables of type struct book.

2) Declaration of structure variable after defining the structure

```
struct book
{
            int bookid;
            char bookname[20];char
            author[20]; float price;
            int year;int
            pages;
            char publisher[25];
};

struct book b1, b2, b3;
```

NOTE:

- Structure tag name is optional.

E.g.

```
struct
{
int bookid;
char bookname[20];char
author[20]; float price;
int year; int
pages;
char publisher[25];
}b1, b2, b3;
```

Declaration of structure variable at a later time is not possible with this type ofdeclaration. It is a drawback in this method. So the second method can be preferred.

- Structure members are not variables. They don"t occupy memory until theyare associated with a structure variable.

Accessing Structure Members

There are many ways for storing values into structure variables. The members of astructure can be accessed using a “dot operator” or “period operator”.

E.g. b1.author -> b1 is a structure variable and author is a structure member.

Syntax

STRUCTURE_Variable.STRUCTURE_Members

The different ways for storing values into structure variable is given below:

Method 1: Using Simple Assignment Statement

```
    b1.pages = 786;
  b1.price = 786.50;
```

Method 2: Using strcpy function

```
    strcpy(b1.title, ‚Programming in C‛);strcpy(b1.author,
    ‚John‛);
```

Method 3: Using scanf function scanf(‚%s \n‛,

```
    b1.title); scanf(‚%d \n‛, &b1.pages);
```

Example

```
    #include<stdio.h>
    #include<conio.h>  struct
    book
        {
        int bookid;
        char bookname[20];
        char author[20];float
        price;
        int year;int
```

```
pages;
char publisher[25];


};

struct book b1, b2, b3;main()
{
struct book b1;
clrscr();
printf("Enter the Book Id: "); scanf("%d",
&b1.bookid); printf("Enter the Book Name: ");
scanf("%s", b1.bookname); printf("Enter the Author
Name:   ");scanf("%s", b1.author); printf("Enter the
Price: "); scanf("%f", &b1.price); printf("Enter the
Year: "); scanf("%d", &b1.year);
printf("Enter the Total No. of Pages:   ");scanf("%d",
&b1.pages);
printf("Enter the Publisher Name:   ");scanf("%s",
b1.publisher);
printf("%d %s %d %f %d %d %s", b1.bookid, b1.bookname,b1.author, b1.price, b1.year,
b1.pages, b1.publisher);
getch();
}
```

Output

```
Enter the Book Id:   786
Enter the Book Name:   ProgrammingEnter the
Author Name:   John Enter the Price:   123.50
Enter the Year:   2015
Enter the Total No. of Pages: 649 Enter the Publisher
Name:   Tata McGraw
786    Programming                2118  123.500000                  2015  649    Tata
```

Structure Initialization

Like variables, structures can also be initialized at the compile time.

Example

```
main()
     {
       struct
       {
            int   rollno; int
            attendance;
       }
     s1={786, 98};
     }
```

The above example assigns 786 to the rollno and 98 to the attendance.Structure

variable can be initialized outside the function also.

Example

```
     main()
     {
       struct student
       {
            int   rollno; int
            attendance;
       };
       struct student s1={786, 98};struct student
       s2={123, 97};
     }
```

Note:

Individual structure members cannot be initialized within the template. Initialization ispossible only with the declaration of structure members.

Nested Structures or Structures within Structures

Structures can also be nested. i.e A structure can be defined inside another structure.

Example

```
     struct   employee
     {
       int empid;
```

```
        char empname[20];
            int basicpay;
        int da;
        int hra;
        int cca;
    } e1;
```

In the above structure, salary details can be grouped together and defined as aseparate structure.

Example

```
    struct   employee
    {
        int empid;
        char empname[20];
struct SAL
        {
                int basicpay;
                int da;
                int hra;
                int cca;
        } salary;
    } e1;
```

The structure employee contains a member named salary which itself is another structure that contains four structure members.   The members inside salary structure can be referred as below:

```
        e1.salary.basicpay
        e1.salary.da; e1.salary.hra;
        e1.salary.cca;
```

However, the inner structure member cannot be accessed without the inner structurevariable.

Example

```
        e1.basicpay
        e1.da e1.hra
        e1.cca
    are invalid statement
```

Moreover, when the inner structure variable is used, it must refer to its inner structure member. If it doesn‟t refer to the inner structure member then it will be considered asan error.

Example

e1.salary (salary is not referring to any inner structure member. Hence it is wrong)

Note: C permits 15 levels of nesting and C99 permits 63 levels of nesting.

Array of Structures

A Structure variable can hold information of one particular record. For example, single record of student or employee. Suppose, if multiple records are to be maintained, it is impractical to create multiple structure variables. It is like the relationship between a variable and an array. Why do we go for an array? Because we don"t want to declare multiple variables and it is practically impossible. Assume that you want to store 1000 values. Do you declare 1000 variables like a1, a2, a3…. Upto a1000? Is it easy to maintain such code ? Is it a good coding? No. It is not. Therefore, we go for Arrays. With a single name, with a single variable, we can store 1000 values. Similarly, to store 1000 records, we cannot declare 1000 structure variables. But we need "Array of Structures".

An array of structure is a group of structure elements under the same structurevariables.

```
struct student s1[1000];
```

The above code creates 1000 elements of structure type student. Each element will be structure data type called student. The values can be stored into the array of structures as follows:

```
s1[0].student_age = 19;
```

Example

```
#include<stdio.h>
#include<conio.h>  struct
book
{
        int bookid;
```

```
                char bookname[20];
                char author[20];
    };


   Struct b1[5];

    main()
    {
    int i; clrscr();
    for (i=0;i<5;i++)
    {
    printf("Enter the Book Id: ");
    scanf("%d", &b1[i].bookid);
    printf("Enter the Book Name: ");
    scanf("%s", b1[i].bookname);
    printf("Enter the Author Name:   ");
    scanf("%s", b1[i].author);
    }
    for (i=0;i<5;i++)
    {
  printf("%d \t %s \t %s \n", b1[i].bookid, b1[i].bookname,b1[i].author);
    }
    getch();
    }
```

Output:

```
          Enter the Book Id:   786
          Enter the Book Name:   Programming Enter the Author
          Name: Dennis RitchieEnter the Book Id:   101
          Enter the Book Name:   Java Complete ReferenceEnter the Author
          Name:   Herbert Schildt Enter the Book Id:   008
          Enter the Book Name:   Computer Graphics
```

```
Enter the Author Name:    Hearn and Baker

786     Programming        Dennis Ritchie
101     Java Complete Reference     Herbert Schildt008   Computer

Graphics                          Hearn and Baker
```

Structure as Function Argument

Example

```
struct sample
{
        int no; float
        avg;
} a;
void main( )
{
        a.no=75;
        a.avg=90.25;
        fun(a);
}

void fun(struct sample p)
{
        printf(,The no is=%d
        Average is %f‟,p.no , p.avg);
}
```

Output

```
The no is 75                Average is 90.25
```

Function that returns Structure

The members of a structure can be passed to a function. If a structure is to be passed to acalled function , we can use any one of the following method.

Method 1 :- Individual member of the structure is passed as an actual argument of the function call. The actual arguments are treated independently. This method is not suitable if a structure is very large structure.

Method 2:- Entire structure is passed to the called function. Since the structure declared as the argument of the function, it is local to the function only. The members are valid for the

function only. Hence if any modification done on any member of the structure , it is not reflected in the original structure.

Method 3 :- Pointers can be used for passing the structure to a user defined function. When the pointers are used , the address of the structure is copied to the function. Hence if any modification done on any member of the structure , it is reflected in the original structure.

```
Return data type   function name ( structured variable )Structured Data type for the Structured variable;
    {
       Local Variable declaration;Statement 1;
       Statement 2;
       - -------------------
       - --------------------
     Statement n;
     }
```

Example :

```
#include <stdio.h>
struct st
{
char name[20];
int no;
int marks;
};
int   main( )
{
          struct st x ,y;int res;
          printf(‚\n Enter the First Record‛);
          scanf(‚%s%d%d‛,x.name,&x.no,&x.marks);printf(‚\n
          Enter the Second Record‛);
          scanf(‚%s%d%d‛,y.name,&y.no,&y.marks);
          res = compare ( x , y );
          if (res == 1)
                     printf(‚\n First student has got the Highest Marks‛);
          else
                     printf(‚\n Second   student has got the Highest Marks‛);
     }
compare ( struct st st1 , struct st st2)
{
    if (st1.marks > st2. marks   )
    return ( 1 );
else
```

```
        return ( 0 );
}
```

In the above example , x and y are the structures sent from the main ( ) function as theactual parameter to the formal parameters st1 and st2 of the function compare ( ).

Example program (1) – passing structure to function

```
#include<stdio.h>
#include<conio.h>
//-       ---------------------------------------------------- ----
struct Example
{
        int num1;int
        num2;
}s[3];
//-       ---------------------------------------------------- ----
void accept(struct Example *sptr)
{
        printf("\nEnter num1 : ");
        scanf("%d",&sptr->num1);
        printf("\nEnter num2 : ");
        scanf("%d",&sptr->num2);
}
//-       ---------------------------------------------------- ----
void print(struct Example *sptr)
{
        printf("\nNum1 : %d",sptr->num1);
        printf("\nNum2 : %d",sptr->num2);
}
//-       ---------------------------------------------------- ----
void main()
{
int i; clrscr();
for(i=0;i<3;i++)
accept(&s[i]);

for(i=0;i<3;i++)
print(&s[i]);

getch();
}
```

Output :

```
Enter num1 : 10Enter
num2 : 20
```

```
Enter     num1 :     30
Enter     num2 :     40
Enter     num1 :     50
Enter     num2 :     60
Num1 :    10
Num2 :    20
Num1 :    30
Num2 :    40
Num1 :    50
Num2 :    60
```

Example program (2) – passing structure to function in C by value:

In this program, the whole structure is passed to another function by value. It means the whole structure is passed to another function with all members and their values. So, this structure can be accessed from called function. This concept is very useful while writing very bigprograms in C.

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

struct student
{
                int id;
                char name[20]; float
                percentage;
};

void func(struct student record);

void   main()
{
                struct student record;

                record.id=1;
                strcpy(record.name, "Raju");
                record.percentage = 86.5;

                func(record);
                getch();
}

void func(struct student record)
{
                printf(" Id is: %d \n", record.id);
                printf(" Name is: %s \n", record.name);
```

```
                                printf(" Percentage is: %f \n", record.percentage);
}
```

```
Id is:    1 Name is:
Raju
Percentage is: 86.500000
```

Output:

Example program (3) Passing structure by value

A structure variable can be passed to the function as an argument as normal variable. If structure is passed by value, change made in structure variable in function definition does not reflect in original structure variable in calling function.

Write a C program to create a structure student, containing name and roll. Ask user the name and roll of a student in main function. Pass this structure to a function and display the information in that function.

```
#include <stdio.h>
struct student
{
        char name[50];int
        roll;
};
void Display(struct student stu);
/* function prototype should be below to the structure declarationotherwise compiler shows error */
          int main()
{
        struct student s1;
        printf("Enter student's name: ");
        scanf("%s",&s1.name);
         printf("Enter roll number:");
        scanf("%d",&s1.roll);
        Display(s1);            // passing structure variable s1 as argumentreturn 0;
}
   void Display(struct student stu){ printf("Output\nName:
      %s",stu.name);printf("\nRoll: %d",stu.roll);
}
```

Output

```
Enter student's name: Kevin AmlaEnter roll
number: 149
Output
```

Example program (4) – Passing structure to function in C by address:

In this program, the whole structure is passed to another function by address. It means only the address of the structure is passed to another function. The whole structure is not passed to another function with all members and their values. So, this structure can be accessed from called function by its address.

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

struct student
{
                int id;
                char name[20];
                float percentage;
};

void func(struct student *record);

void   main()
{
               struct student record;

               record.id=1; strcpy(record.name, "Raju");
               record.percentage = 86.5;

               func(&record);
               getch();
}

void func(struct student *record)
{
            printf(" Id is: %d \n", record->id);
             printf(" Name is: %s \n", record->name);
            printf(" Percentage is: %f \n", record->percentage);
}
```

Output:

```
Id is:   1 Name is:
Raju
Percentage is: 86.500000
```

Example program (5) Passing structure by reference

The address location of structure variable is passed to function while passing it by reference. If structure is passed by reference, change made in structure variable in function definition reflects in original structure variable in the calling function.

Write a C program to add two distances(feet-inch system) entered by user. To solve this program, make a structure. Pass two structure variable (containing distance in feet and inch) to add function by reference and display the result in main function without returning it.

```
#include <stdio.h>struct
distance
{
            int feet; float inch;
};
void Add(struct distance d1,struct distance d2, struct distance *d3);int main()
{
            struct distance dist1, dist2, dist3;printf("First
            distance\n"); printf("Enter feet: ");
            scanf("%d",&dist1.feet); printf("Enter inch: ");
            scanf("%f",&dist1.inch); printf("Second distance\n");
            printf("Enter feet: "); scanf("%d",&dist2.feet);
            printf("Enter inch: "); scanf("%f",&dist2.inch);
            Add(dist1, dist2, &dist3);

/*passing structure variables dist1 and dist2 by value whereas passingstructure variable dist3 by reference */
            printf("\nSum of distances = %d\'-%.1f\"",dist3.feet, dist3.inch);
            return 0;
}
void Add(struct distance d1,struct distance d2, struct distance *d3)
{
    /* Adding distances d1 and d2 and storing it in d3 */d3->feet=d1.feet+d2.feet;
              d3->inch=d1.inch+d2.inch;
         if (d3->inch>=12) {                    /* if inch is greater or equal to 12,converting it to feet.
*/
                    d3->inch-=12;
                    ++d3->feet;
            }
}
```

Output

```
First distanceEnter feet:
12
Enter inch: 6.8
```

```
Second distanceEnter
feet: 5
Enter inch: 7.5
Sum of distances = 18'-2.3"
```

Explanation

In this program, structure variables *dist1* and *dist2* are passed by value (because value of *dist1* and *dist2* does not need to be displayed in main function) and *dist3* is passed by reference ,i.e, address of *dist3* (&dist3) is passed as an argument. Thus, the structure pointer variable *d3* points to the address of *dist3*. If any change is made in *d3* variable, effect of it is seed in *dist3* variable in main function.

Example program(6) to declare a structure variable as global in C:

Structure variables also can be declared as global variables as we declare other variables in C. So, When a structure variable is declared as global, then it is visible to all the functions in a program. In this scenario, we don‟t need to pass the structure to any function separately.

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

struct student
{
                int id;
                char name[20]; float
                percentage;
};
struct student record; // Global declaration of structurevoid structure_demo();

int main()
{
                record.id=1;
                strcpy(record.name, "Raju");
                record.percentage = 86.5;
                structure_demo();return
                0;

}

void structure_demo()
{
                printf(" Id is: %d \n", record.id);
                 printf(" Name is: %s \n", record.name);
                printf(" Percentage is: %f \n", record.percentage);
}
```

OUTPUT:

```
Id is:   1 Name is:
Raju
Percentage is: 86.500000
```

Example program(7)Passing Array of Structure to Function in C Programming

Array of Structure can be passed to function as a Parameter.function can also return Structureas return type.Structure can be passed as follow

Example :

```
#include<stdio.h>
#include<conio.h>
//-        ---------------------------------------------------- ----
struct Example
{
        int num1;int
        num2;
}s[3];
//-        ---------------------------------------------------- ~~void~~
accept(struct Example sptr[],int n)
{
        int i; for(i=0;i<n;i++)
        {
        printf("\nEnter       num1       :       ");
        scanf("%d",&sptr[i].num1);
        printf("\nEnter       num2       :       ");
        scanf("%d",&sptr[i].num2);
        }
}
//-        ---------------------------------------------------- ----
void print(struct Example sptr[],int n)
{
        int i; for(i=0;i<n;i++)
        {
        printf("\nNum1 : %d",sptr[i].num1);
        printf("\nNum2 : %d",sptr[i].num2);
        }
}        ----------------------------------------------------  ----
void main()
{
int i;
clrscr();
accept(s,3);
print(s,3);
getch();
}
```

Output :

```
    Enter    num1 :    10
    Enter    num2 :    20
    Enter    num1 :    30
    Enter    num2 :    40
    Enter    num1 :    50
    Enter    num2 :    60
    Num1 :   10
    Num2 :   20
    Num1 :   30
    Num2 :   40
    Num1 :   50
    Num2 :   60
```

Explanation :

Inside main structure and size of structure array is passed. When reference (i.e ampersand) is not specified in main , so this passing is simple pass by value. Elements can be accessed by using dot [.] operator

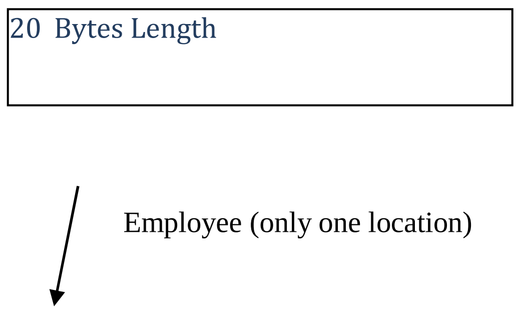
Union

The concept of Union is borrowed from structures and the formats are also same. The distinction between them is in terms of storage. In structures , each member is stored in its own location but in Union , all the members are sharing the same location. Though Union consists of more than one members , only one member can be used at a particular time. The size of the cell allocated for an Union variable depends upon the size of any member within Union occupying more no:- of bytes. The syntax is the same as structures but we use the keyword **union** instead of **struct.**

Example:- the employee record is declared and processed as follows

```
union   emp
        {
                char name[20];int
                eno; float salary;
          } employee;
```

where employee is the union variable which consists of the member name,noand salary. The compiler allocates only one cell for the union variable as



Location / Cell for name,no and salary

20 bytes cell can be shared by all the members because the member name is occupying the highest no:- of bytes. At a particular time we can handle only one member.To access the members of an union , we have to use the same format of structures.

Example program for C union:

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


union student
{
                    char name[20]; char
                    subject[20];float
                    percentage;
};

int main()
{
        union student record1;union
        student record2;

        // assigning values to record1 union variablestrcpy(record1.name,
            "Raju"); strcpy(record1.subject, "Maths"); record1.percentage =
            86.50;

            printf("Union record1 values example\n"); printf(" Name : %s \n",
            record1.name); printf(" Subject   : %s \n", record1.subject);
            printf(" Percentage : %f \n\n", record1.percentage);

        // assigning values to record2 union variableprintf("Union record2 values
            example\n");strcpy(record2.name, "Mani");
            printf(" Name                   : %s \n", record2.name);

            strcpy(record2.subject, "Physics");
            printf(" Subject                : %s \n", record2.subject);

             }
```

Output: ntf(" Percentage : %f \n", record2.percentage);return 0;
record2.per
centage =
99.50;
p
r
i

```
Union record1 values exampleName :
Subject :
Percentage : 86.500000; Union
record2 values exampleName : Mani
Subject : Physics Percentage :
99.500000
```

Explanation for above C union program:

There are 2 union variables declared in this program to understand the difference inaccessing values of union members.

Record1 union variable:

- “Raju” is assigned to union member “record1.name” . The memory location name is “record1.name” and the value stored in this location is “Raju”.
- Then, “Maths” is assigned to union member “record1.subject”. Now, memory location name is changed to “record1.subject” with the value “Maths” (Union can hold only one member at a time).
- Then, “86.50” is assigned to union member “record1.percentage”. Now, memory location name is changed to “record1.percentage” with value “86.50”.
- Like this, name and value of union member is replaced every time on the common storage space.
- So, we can always access only one union member for which value is assigned at last.

  We can‟t access other member values.
- So, only “record1.percentage” value is displayed in output. “record1.name” and “record1.percentage” are empty.

Record2 union variable:

- If we want to access all member values using union, we have to access the member before assigning values to other members as shown in record2 union variable in this program.
- Each union members are accessed in record2 example immediately after assigning values to them.
- If we don"t access them before assigning values to other member, member name and value will be over written by other member as all members are using same memory.
- We can"t access all members in union at same time but structure can do that.

Example program – Another way of declaring C union:

In this program, union variable "record" is declared while declaring union itself as shown inthe below program.

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


union student
{
                char name[20]; char
                subject[20];float
                percentage;
}record; int

main()
{


                strcpy(record.name, "Raju");
                strcpy(record.subject, "Maths");
                record.percentage = 86.50;

              printf(" Name                    : %s \n", record.name); printf(" Subject
                                       : %s \n", record.subject); printf("
              Percentage : %f \n", record.percentage);return 0;

}
```

Output:

```
Name :
Subject :
Percentage : 86.500000
```

Note:

We can access only one member of union at a time. We can‟t access all member values at the same time in union. But, structure can access all member values at the same time. This is because, Union allocates one common storage space for all its members. Wher**e** as Structure allocates storage space for all its members separately.

Difference between structure and union in C:

S.no	**C Structure**	**C Union**
1	Structure allocates storage space for all its members separately.	Union allocates one common storage space for all its members. Union finds that which of its member needs high storage space over other members and allocates that much space
2	Structure occupies higher memory space.	Union occupies lower memory space over structure.
3	We can access all membersof structure at a time.	We can access only one member of union at a time.

4	Structure example:
	struct student
	{
	int mark; char
	name[6];
	double average;
	};
		Union example:
		union student
		{
		int mark; char
		name[6];
		double average;
		};

5	For above structure, memory allocation will be like below. int mark – 2B char name[6] – 6B double average – 8B Total memory allocation = 2+6+8 = 16 Bytes	For above union, only 8 bytes of memory will be allocatedsince double data type will occupy maximum space of memory over other data types. Total memory allocation = 8 Bytes