

### MODULE-5

Structures, union and Pointers: Structure definition, giving values to members, structure initialization, comparison of structure variables, arrays of structures, arrays within structures, Structure and functions, structures within structures. Unions. Pointers: Understanding pointers, accessing the address of a variable, declaring & initializing pointers, accessing a variable through its pointer, pointer expression, pointer increments & scale factor, passing pointer variables as function arguments.

## Structures

It is the collection of dissimilar data types or heterogenous data types grouped together. It means the data types may or may not be of same type.

A structure can be defined to be a group of logically related data items, which may be of different types, stored in contiguous memory locations, sharing a common name, but distinguished by its members.

- A **structure** in C is a derived or user-defined data type. We use the keyword **struct** to define a custom data type that groups together the elements of different types. The difference between an array and a structure is that an [array](#) is a homogenous collection of similar types, whereas a structure can have elements of different types stored adjacently and identified by a name.
- We are often required to work with values of different [data types](#) having certain relationships its **title** (string), **author** (string), **price** (double), **number of pages** (integer), using four different variables, these values can be stored in a single **struct** variable.

### Structure declaration:

The format or prototype of the user-defined data type allows us to logically relate a group of data items which may be of different types.

Struct agname

{

Data type member1;

Data type member2;

Data type member3;

.....

.....

Data type member n;

};

OR

struct

{

Data type member1;

Data type member2;

Data type member3;

.....

.....

Data type member n;

};

Where struct is a keyword. Tag-name is a user-defined name, it is the name of the structure, datatype refers to any valid data type supported by C and members are the members of structure.

#### **Note:**

Members are not variables by themselves. So no memory locations get allocated here. Memory locations get allocated only when variables of the structure are declared.

#### **Declaration of structure variables:**

##### **Syntax:**

Struct tag-name variable-name;

OR

Structtagname

{

Data type member1;

Data type member2;

Data type member3;

.....

.....

Data type member n;

}variable name;

Example:

struct student s;

### Initialization of structure variable

A structure can be initialized in the same way as other data types are initialized. Initializing a structure means assigning some constants to the members of the structure. When the user does not explicitly initialize the structure, then C automatically does that. For int and float members, the values are initialized to zero and character and string members are initialized to '\0' by default (in the absence of any initialization done by the user). The initializers are enclosed in braces and are separated by commas. However, care must be taken to see that the initializers match their corresponding types in the structure definition.

Like primary variables structure variables can also be initialized when they are declared.

Structure templates can be defined locally or globally.

If it is local it can be used within that function. If it is global it can be used by all other functions of the program. We can't initialize structure members while defining the structure.

struct student

{

int age, roll; char name[20];

} struct student s1={16,101,"sona"};

struct student s2={17,102,"rupa"};

If initialiser is less than no.of structure variable, automatically rest values are taken as zero.

### C Program to illustrate how to initialize a structure

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

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

```
struct Student {  
    int id;  
    char name[50];  
    float percentage;  
};  
  
int main()  
{  
    // Initialize a structure  
    struct Student student1 = { 1, "John Doe", 85.5 };  
    // Print the initialized structure  
    printf("ID: %d\n", student1.id);  
    printf("Name: %s\n", student1.name);  
    printf("Percentage: %.2f\n", student1.percentage);  
    return 0;  
}
```

### Accessing the Members of a Structure

Each member of a structure can be used just like a normal variable, but its name will be a bit longer. A structure member variable is generally accessed using a '.' (dot) operator.

The syntax of accessing a structure or a member of a structure can be given as follows:

struct\_var.member\_name

The dot operator is used to select a particular member of the structure.

For example, to assign value to the individual data members of the structure variable stud1, we may write

```
stud1.r_no = 01;  
stud1.name = "Rahul";  
stud1.course = "BCA";  
stud1.fees = 45000;
```

To input values for data members of the structure variable stud1, we may write

```
scanf("%d", &stud1.r_no);
```

```
scanf("%s", stud1.name);
```

Similarly, to print the values of structure variable stud1, we may write

```
printf("%s", stud1.course);
```

```
printf("%f", stud1.fees);
```

### Comparison of structure variables

We can assign a structure to another structure of the same type. For example, if we have two structure variables stud1 and stud2 of type struct student given as

```
struct student stud1 = {f1, "Rahul", "BCA", 45000};
```

```
struct student stud2;
```

Then to assign one structure variable to another we will write, stud2 = stud1; This statement initializes the members of stud2 with the values of members of stud1.

### Compare the structure variable when Both have the same value

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

```
struct person
```

```
{
```

```
    int pid;
```

```
    char G;
```

```
};
```

```
int main()
```

```
{
```

```
    struct person p1, p2;
```

```
    /*Assigning value to the p1*/
```

```
    p1.pid = 1;
```

```
    p1.G = 'M';
```

```
    /*Assigning value to the p2*/
```

```
    p2.pid = 1;
```

```
    p2.G = 'M';
```

```
    //Comparing the structure variable
```

```
    if((p1.pid == p2.pid) && (p1.G == p2.G))
```

```
    {
```

```
        printf("Struct variables are equal\n");
```

```
    }
```

```
    return 0;
```

```
}
```

**Output:** Struct variables are equal

#### Array of Structures

- An array is a data type that stores a collection of elements.
- It is usually specified as a variable or array value.
- Array can be vectors and matrices, often called vector and matrix types, respectively.
- The most common use of structure in C programming is an array of structures.  
to declare an array of structure first the structure must be defined and then an array variable of that type should be defined.
- For Example, if we need to create an array of structures of containing 10 elements you can do so, as – **struct book b[10];**

#### Array of Structures in C

An array of structures in C can be defined as the collection of multiple structures variables where each variable contains information about different entities. The array of structures in C are used to store information about multiple entities of different data types. The array of structures is also known as the collection of structures.

##### Example:

```
#include<stdio.h>

#include <string.h>

struct student{
int rollno;
char name[10];
};

int main(){
int i;
struct student st[5];
printf("Enter Records of 5 students");
for(i=0;i<5;i++){
```

```
printf("\nEnter Rollno:");  
scanf("%d",&st[i].rollno);  
printf("\nEnter Name:");  
scanf("%s",&st[i].name);  
  
}  
  
printf("\nStudent Information List:");  
  
for(i=0;i<5;i++){  
printf("\nRollno:%d, Name:%s",st[i].rollno,st[i].name);  
  
}  
  
return 0;  
  
}
```

#### **Output**

Enter Records of 5 students

Enter Rollno:1

Enter Name: Sonoo

Enter Rollno:2

Enter Name: Ratan

Enter Rollno:3

Enter Name: Vimal

Enter Rollno:4

Enter Name: James

Enter Rollno:5

Enter Name: Sarfraz

Student Information List:

Rollno:1, Name: Sonoo

Rollno:2, Name: Ratan

Rollno:3, Name: Vimal

Rollno:4, Name: James

Rollno:5, Name: Sarfraz

### Array within the structure

In C programming, we can have **structure members** of **type arrays**. Any structure having an array as a structure member is known as an **array within the structure**. We can have as many members of the type array as we want in C structure.

To access each element from an array within a structure, we write **structure variables followed by dot (.) and then array name along with index**.

#### Example program:

```
struct student
{
    int roll;

    float marks [5]; /* This is array within structure */
};
```

In above example structure named student contains a member element mark which is of type array. In this example marks is array of floating-point numbers.

#### Example program:

```
#include<stdio.h>

int main()
{
    struct student s;

    int i;

    float sum=0, p;

    printf("Enter name and roll number of students:\n");

    scanf("%s%d",s.name,&s.roll);
```



## MODULE-05

### PROBLEM SOLVING USING C PROGRAMMING (BCA103)

---

```
printf("\nEnter marks obtained in five different subject\n");  
  
for(i=0;i< 5;i++) {  
    printf("Enter marks:\n");  
    scanf("%f",&s.marks[i]);  
    sum = sum + s.marks[i];  
}  
  
p = sum/5;  
  
printf("Student records and percentage is:\n");  
  
printf("Name : %s\n", s.name);  
printf("Roll Number : %d\n", s.roll);  
printf("Percentage obtained is: %f", p);  
  
return 0;  
}
```

#### **Output:**

Enter name and roll number of the student:

Alice 202

Enter marks obtained in five different subjects:

Enter marks for subject 1:

85

Enter marks for subject 2:

90

Enter marks for subject 3:

78

Enter marks for subject 4:

88

Enter marks for subject 5:

92

Student records and percentage:

Name: Alice

Roll Number: 202

Percentage obtained is: 86.60%

### Structures within structures

A structure within a structure is known as **nested structure in C**. When one of the elements in the definition of a struct type is of another struct type, then we call it a nested structure in C. Nested structures are defined when one of the elements of a struct type is itself a composite representation of one or more types.

When a structure is within another structure, it is called Nested structure. A structure variable can be a member of another structure and it is represented as

#### Syntax:

```
struct struct1
{
    type var1;
    type var2;
    struct struct2 strvar;
}

struct student
{
    element 1;
    element 2;
    .....
    .....

    struct student1
    {
        member 1;
```

member 2;

}variable1;

.....

.....

element n;

}variable 2;

It is possible to define structure outside & declare its variable inside other structure.

**Example program:**

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

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

```
struct employee
```

```
{
```

```
char name [10];
```

```
float salary;
```

```
struct dob
```

```
{
```

```
int d, m, y;
```

```
} d1;
```

```
};
```

```
int main()
```

```
{
```

```
struct employee e1 = {"Kiran", 25000, {12, 5, 1990}};
```

```
printf("Name: %s\n", e1.name);
```

```
printf("Salary: %f\n", e1.salary);
```

```
printf("Date of Birth: %d-%d-%d\n", e1.d1.d, e1.d1.m, e1.d1.y);
```

```
return 0;
```

```
}
```

**Output:**

Name: Kiran

Salary: 25000.000000

Date of Birth: 12-5-1990

**Union**

**Union** is derived data type contains collection of different data type or dissimilar elements. All definition declaration of union variable and accessing member is similar to structure, but instead of keyword struct the keyword union is used.

The main difference between union and structure is Each member of structure occupy the memory location, but in the unions members share memory. Union is used for saving memory and concept is useful when it is not necessary to use all members of union at a time.

**Syntax of union:**

```
union student {  
    datatype member1;  
    datatype member2;  
};
```

Like structure variable, union variable can be declared with definition or separately such as

```
union union name  
{  
    Datatype member1;  
}var1;
```

Example:- union student s;

	STRUCTURE	UNION
<b>Keyword</b>	The keyword <b>struct</b> is used to define a structure	The keyword <b>union</b> is used to define a union.
<b>Size</b>	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is <b>greater than or equal to the sum of sizes of its members.</b>	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of <b>union is equal to the size of largest member.</b>
<b>Memory</b>	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
<b>Value Altering</b>	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
<b>Accessing members</b>	Individual member can be accessed at a time.	Only one member can be accessed at a time.
<b>Initialization of Members</b>	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

## POINTER

A pointer is a variable that store memory address or that contains address of another variable where addresses are the location number always contains whole number. So, pointer contain always the whole number. It is called pointer because it points to a particular location in memory by storing address of that location.

Syntax-

**Data type \*pointer name;**

Here \* before pointer indicate the compiler that variable declared as a pointer.

**e.g.**

int \*p1; //pointer to integer type

float \*p2; //pointer to float type

char \*p3; //pointer to character type

#### Program to Demonstrate Accessing Array Elements Using Pointers

```
#include <stdio.h>

int main() {

    int arr[3] = { 10, 20, 30};

    int *ptr = arr; // Pointer to the first element of the array

    printf("Accessing array elements using pointers:\n");

    for (int i = 0; i < 3; i++) {

        printf("Element %d: %d\n", i, *(ptr + i)); // Accessing via pointer arithmetic

    }

    printf("\nAccessing array elements using array indexing:\n");

    for (int i = 0; i < 3; i++) {

        printf("Element %d: %d\n", i, arr[i]); // Accessing via array indexing

    }

    return 0;

}
```

#### Output:

Accessing array elements using pointers:

Element 0: 10

Element 1: 20

Element 2: 30

Accessing array elements using array indexing:

Element 0: 10

Element 1: 20

Element 2: 30

### Accessing the address of a variable

The actual location of a variable in the memory is system dependent and therefore, the address of a variable is not known to us immediately. How can we then determine the address of a variable? This can be done with the help of the operator & available in C. we have already seen the use of this address operator in the scanf function. The operator & immediately preceding a variable returns the address of the variable associated with it. For example, the statement

```
P=&quantity;
```

Would assign the address 5000(the location of quantity) to the variable p. The & operator can be remembered as ‘address of’. The & operator can be used only with a simple variable or an array element. The following are illegal use of address operator:

1. &125 (pointing at constants).
2. Int x[10];  
    &x (pointing at array names).
3. &(x+y)(pointing at expressions).  
    If x is an array, then expressions such as  
    &x[0] and &x[i+3]  
    Are valid represent the addresses of 0<sup>th</sup> and (i+3)th elements of x.

### Example:

```
#include<stdio.h>

int main()
{
char a;

int x;

float p,q;

a='A';

x=125;

p=10.25,q=18.76;

printf(“%c is stored at addr %u.\n”,a,&a);

printf(“%d is stored at addr %u.\n”,x,&x);

printf(“%f is stored at addr %u.\n”,p,&p);
```

```
printf("%f is stored at addr %u.\n",q,&q);  
}
```

Output:

A is stored at addr 4436.

125 is stored at addr 4434.

10.250000 is stored at addr 4442.

18.760000 is stored at addr 4438.

### Accessing a variable through its pointer

Accessing a variable through its pointer involves dereferencing the pointer, which means accessing the value stored at the memory address pointed to by the pointer.

As we know that a pointer is a special type of variable that is used to store the memory address of another variable. A normal variable contains the value of any type like **int**, **char**, **float** etc, while a pointer variable contains the memory address of another variable.

1. **Declare a Variable:** Create a variable of a specific type.
2. **Declare a Pointer:** Create a pointer of the same type and assign it the address of the variable using the address-of operator (&).
3. **Dereference the Pointer:** Use the dereference operator (\*) to access or modify the value stored at the memory address the pointer points to.

**Here's how you can access a variable through its pointer in C:**

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

```
int main()
```

```
{
```

```
    int x = 10;      // Define an integer variable 'x'
```

```
    int *ptr = &x;   // Define a pointer 'ptr' and initialize it with the address of 'x'
```

```
    // Accessing the value of 'x' through its pointer 'ptr'
```

```
    printf("Value of x: %d\n", *ptr);
```



```
    return 0;  
  
}
```

In this example:

- We define an integer variable **x** and initialize it with the value **10**.
- We define a pointer variable **ptr** and initialize it with the address of the variable **x** using the “address-of” operator (**&**).
- To access the value of **x** through its pointer **ptr**, we use the dereference operator (**\*ptr**). This retrieves the value stored at the memory address pointed to by **ptr**.
- We then print the value of **x** using **\*ptr**.

When you run this program, it will print the value of **x**, which is **10**, obtained through its pointer **ptr**.

It's important to note that dereferencing a pointer when it's not pointing to a valid memory address (e.g., when it's uninitialized or pointing to **NULL**) leads to undefined behaviour, which may result in a segmentation fault or other runtime errors. Always ensure that the pointer is properly initialized and points to a valid memory location before dereferencing it.

### Pointer Expressions

Like other variables, pointer variables can also be used in expressions. For example, if **ptr1** and **ptr2** are pointers, then the following statements are valid.

```
int num1=2, num2= 3, sum=0, mul=0, div=1;
```

```
int *ptr1, *ptr2;
```

```
ptr1 = &num1;
```

```
ptr2 = &num2;
```

```
sum = *ptr1 + *ptr2;
```

```
mul = sum * *ptr1;
```

```
*ptr2 +=1;
```

```
div = 9 + *ptr1/*ptr2 - 30;
```

In C, the programmer may add or subtract integers from pointers. We can also subtract one pointer from the other. We can also use short hand operators with the pointer variables as we use with other variables. C also allows to compare pointers by using relational operators in the expressions.

For example,  $p1 > p2$ ,  $p1 == p2$ , and  $p1 != p2$  are all valid in C.

### Pointer Increments and Scale Factor

When using pointers, unary increment (++) and decrement (--) operators have greater precedence than the dereference operator (\*). Both these operators have a special behaviour when used as suffix. In that case the expression is evaluated with the value it had before being increased. Therefore, the expression `*ptr++` is equivalent to `*(ptr++)` as ++ has greater operator precedence than \*. Therefore, the expression will increase the value of ptr so that it now points to the next memory location. This means the statement `*ptr++` does not perform the intended task. Therefore, to increment the value of the variable whose address is stored in ptr, you should write `(*ptr)++`

**Let us now summarize the rules for pointer operations:**

- A pointer variable can be assigned the address of another variable
- A pointer variable can be assigned the value of another pointer variable
- A pointer variable can be initialized with a null value.
- Prefix or postfix increment and decrement operators can be applied on a pointer variable.
- An integer value can be added or subtracted from a pointer variable.
- A pointer variable can be compared with another pointer variable of the same type using relational operators.
- A pointer variable cannot be multiplied by a constant.
- A pointer variable cannot be added to another pointer variable.

### PASSING ARGUMENTS TO FUNCTION USING POINTERS

Pointers provide a mechanism to modify data declared in one function using code written in another function. In other words: If data is declared in `func1()` and we want to write code in `func2()` that modifies the data in `func1()`, then we must pass the addresses of the variables to `func2()`. The calling function sends the addresses of the variables and the called function declares those incoming arguments as pointers. In order to modify the variables sent by the calling function, the called function must dereference the pointers that were passed to it. Thus, passing pointers to a function avoids the overhead of copying data from one function to another.

**Write a program to add two integers using functions**

```
#include<stdio.h>

#include<conio.h>

void sum (int *a, int *b, int *t);

int main()

{

int num1, num2, total;

printf("\n Enter the first number:");

scanf("%d", &num1);

printf("\n Enter the second number:");

scanf("%d", &num2);

sum(&num1, &num2, &total);

printf("\n Total = %d", total);

getch();

return 0;

}

void sum (int *a, int *b, int *t)

{

*t = *a + *b;

}
```

#### **Output**

Enter the first number: 2

Enter the second number: 3

Total = 5