**VISION INSTITUTE OF MANAGEMENT**

**C PROGRAMMING**

**BCA 1st YEAR/2nd SEM**

**UNIT-4**

# **C Structures**

Structure is a user-defined datatype in C language which allows us to combine data of different types together. Structure helps to construct a complex data type which is more meaningful. It is somewhat similar to an Array, but an array holds data of similar type only. But structure on the other hand, can store data of any type, which is practical more useful.

**For example:** If I have to write a program to store Student information, which will have Student's name, age, branch, permanent address, father's name etc., which included string values, integer values etc., how can I use arrays for this problem, I will require something which can hold data of different types together.

In structure, data is stored in form of **records**

## **Defining a structure**

struct keyword is used to define a structure. struct defines a new data type which is a collection of primary and derived datatypes.

**Syntax:**

struct [structure\_tag]

{

 //member variable 1

 //member variable 2

 //member variable 3

 ...

}[structure\_variables];

As you can see in the syntax above, we start with the struct keyword, then it's optional to provide your structure a name, we suggest you to give it a name, then inside the curly braces, we have to mention all the member variables, which are nothing but normal C language variables of different types like int, float, array etc.

After the closing curly brace, we can specify one or more structure variables, again this is optional.

**Note:** The closing curly brace in the structure type declaration must be followed by a semicolon (;).

### Example of Structure

struct Student

{

 char name[25];

 int age;

 char branch[10];

 // F for female and M for male

 char gender;

};

Here struct Student declares a structure to hold the details of a student which consists of 4 data fields, namely name, age, branch and gender. These fields are called **structure elements or members**.

Each member can have different datatype, like in this case, name is an array of char type and age is of int type etc. **Student** is the name of the structure and is called as the **structure tag.**

### Declaring Structure Variables

It is possible to declare variables of a **structure**, either along with structure definition or after the structure is defined. **Structure** variable declaration is similar to the declaration of any normal variable of any other datatype. Structure variables can be declared in following two ways:

#### **1) Declaring Structure variables separately-**

struct Student

{

 char name[25];

 int age;

 char branch[10];

 //F for female and M for male

 char gender;

};

struct Student S1, S2; //declaring variables of struct Student

#### **2) Declaring Structure variables with structure definition**

struct Student

{

 char name[25];

 int age;

 char branch[10];

 //F for female and M for male

 char gender;

}S1, S2;

Here S1 and S2 are variables of structure Student. However this approach is not much recommended.

### Accessing Structure Members

Structure members can be accessed and assigned values in a number of ways. Structure members have no meaning individually without the structure. In order to assign a value to any structure member, the member name must be linked with the **structure** variable using a dot . operator also called **period** or **member access** operator.

**For example:**

#include<stdio.h>

#include<string.h>

struct Student

{

 char name[25];

 int age;

 char branch[10];

 //F for female and M for male

 char gender;

};

int main()

{

struct Student s1;

 /\*

 s1 is a variable of Student type and

 age is a member of Student

 \*/

 s1.age = 18;

 /\*

 using string function to add name

 \*/

 strcpy(s1.name, "Viraaj");

 /\*

 displaying the stored values

 \*/

 printf("Name of Student 1: %s\n", s1.name);

 printf("Age of Student 1: %d\n", s1.age);

 return 0;

}

**OUTPUT:**

Name of Student 1: Viraaj

Age of Student 1: 18

We can also use scanf() to give values to structure members through terminal.

scanf(" %s ", s1.name);

scanf(" %d ", &s1.age);

### Structure Initialization

Like a variable of any other datatype, structure variable can also be initialized at compile time.

struct Patient

{

 float height;

 int weight;

 int age;

};

struct Patient p1 = { 180.75 , 73, 23 }; //initialization

**OR**

struct Patient p1;

p1.height = 180.75; //initialization of each member separately

p1.weight = 73;

p1.age = 23;

### Nested Structures

Nesting of structures, is also permitted in C language. Nested structures means, that one structure has another structure as member variable.

**Example:**

struct Student

{

 char[30] name;

 int age;

 /\* here Address is a structure \*/

 struct Address

 {

 char[50] locality;

 char[50] city;

 int pincode;

 }addr;

};

**Example of Structure:**

### Add two distances-

// Program to add two distances (feet-inch)

#include <stdio.h>

struct Distance

{

 int feet;

 float inch;

} dist1, dist2, sum;

int main()

{

 printf("1st distance\n");

 printf("Enter feet: ");

 scanf("%d", &dist1.feet);

 printf("Enter inch: ");

 scanf("%f", &dist1.inch);

 printf("2nd distance\n");

 printf("Enter feet: ");

 scanf("%d", &dist2.feet);

 printf("Enter inch: ");

 scanf("%f", &dist2.inch);

 // adding feet

 sum.feet = dist1.feet + dist2.feet;

 // adding inches

 sum.inch = dist1.inch + dist2.inch;

 // changing to feet if inch is greater than 12

 while (sum.inch >= 12)

 {

 ++sum.feet;

 sum.inch = sum.inch - 12;

 }

 printf("Sum of distances = %d\'-%.1f\"", sum.feet, sum.inch);

 return 0;

}

**Output**

1st distance

Enter feet: 12

Enter inch: 7.9

2nd distance

Enter feet: 2

Enter inch: 9.8

Sum of distances = 15'-5.7"

# **C – Unions**

A **union** is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

## **Defining a Union**

To define a union, you must use the **union** statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The format of the union statement is as follows −

union [union tag]

{

 member definition;

 member definition;

 ...

 member definition;

} [one or more union variables];

The **union tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional. Here is the way you would define a union type named Data having three members i, f, and str –

union Data

{

 int i;

 float f;

 char str[20];

} data;

Now, a variable of **Data** type can store an integer, a floating-point number, or a string of characters. It means a single variable, i.e., same memory location, can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in the above example, Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by a character string. The following example displays the total memory size occupied by the above union −

#include <stdio.h>

#include <string.h>

union Data

{

 int i;

 float f;

 char str[20];

};

int main( )

{

 union Data data;

 printf( "Memory size occupied by data : %d\n", sizeof(data));

 return 0;

}

When the above code is compiled and executed, it produces the following result −

Memory size occupied by data: 20

## **Accessing Union Members**

To access any member of a union, we use the **member access operator (.)**. The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use the keyword **union** to define variables of union type. The following example shows how to use unions in a program −

#include <stdio.h>

#include <string.h>

union Data

{

 int i;

 float f;

 char str[20];

};

int main( )

{

 union Data data;

 data.i = 10;

 data.f = 220.5;

 strcpy( data.str, "C Programming");

 printf( "data.i : %d\n", data.i);

 printf( "data.f : %f\n", data.f);

 printf( "data.str : %s\n", data.str);

 return 0;

}

When the above code is compiled and executed, it produces the following result −

data.i: 1917853763

data.f: 4122360580327794860452759994368.000000

data.str: C Programming

Here, we can see that the values of **i** and **f** members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of **str** member is getting printed very well.

Now let's look into the same example once again where we will use one variable at a time which is the main purpose of having unions −

#include <stdio.h>

#include <string.h>

union Data

{

 int i;

 float f;

 char str[20];

};

int main( )

{

 union Data data;

 data.i = 10;

 printf( "data.i : %d\n", data.i);

 data.f = 220.5;

 printf( "data.f : %f\n", data.f);

 strcpy( data.str, "C Programming");

 printf( "data.str : %s\n", data.str);

 return 0;

}

When the above code is compiled and executed, it produces the following result −

data.i : 10

data.f : 220.500000

data.str : C Programming

Here, all the members are getting printed very well because one member is being used at a time.

# **Difference between Structure and Union in C**

[**Structures in C**](https://www.geeksforgeeks.org/structures-c/)

A structure is a user-defined data type available in C that allows to combining data items of different kinds. Structures are used to represent a record.

**Defining a structure:** To define a structure, you must use the **struct** statement. The struct statement defines a new data type, with more than or equal to one member. The format of the struct statement is as follows:

 struct [structure name]

 {

 member definition;

 member definition;

 ...

 member definition;

 };

[**Union**](https://www.geeksforgeeks.org/union-c/) **in C**

A union is a special data type available in C that allows storing different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple purposes.

**Defining a Union:** To define a union, you must use the **union** statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The format of the union statement is as follows:

 union [union name]

 {

 member definition;

 member definition;

 ...

 member definition;

 };

**Similarities between Structure and Union**

1. Both are user-defined data types used to store data of different types as a single unit.
2. Their members can be objects of any type, including other structures and unions or arrays. A member can also consist of a bit field.
3. Both structures and unions support only assignment = and sizeof operators. The two structures or unions in the assignment must have the same members and member types.
4. A structure or a union can be passed by value to functions and returned by value by functions. The argument must have the same type as the function parameter. A structure or union is passed by value just like a scalar variable as a corresponding parameter.
5. **‘.’** operator is used for accessing members.

**Differences**

****