# SNS COLLEGE OF TECHNOLOGY

**Coimbatore – 35**

**An Autonomous Institution**

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A++' Grade

Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

## DEPARTMENT OF INFORMATION TECHNOLOGY

## PROGRAMMING IN C AND DATA STRUCTURES

**I YEAR - II SEM**

## UNIT III – ARRAYS AND INTRODUCTION TO DATA STRUCTURES

## TOPIC – STRUCTURE – Definition, Declaration, Initialization

# Introduction

- In C language, to store the integers, characters, and decimal values, we have int, char, float, or double data types already defined(also known as the primitive data types).
- Also, we have some derived data types such as arrays and strings, to store similar types of data types elements together.
- Still, the problem with arrays or strings is that they can only store variables of similar data types, and the string can store only characters.
- What if we need to store two different data types together in C for many objects? Like, there is a student variable that may have its name, class, section, etc.
- So if we want to store all of its information, We can create different variables for every variable like a character array to store the name, an integer variable to store the class, and a character variable to store the section.
- But this solution is a little messy, C provides us with a better neat and clean solution, i.e., Structure.

# Structure

- ✓ Structure in C is a **User-Defined data type**.
- ✓ It is used to bind two or more similar or different data types or data structures together into a single type.
- ✓ The structure is created using the struct keyword, and a structure variable is created using the struct keyword and the structure tag name.
- ✓ A data type created using structure in C can be treated as other primitive data types of C to define a pointer for structure, pass structure as a function argument or a function can have structure as a return type.
- ✓ For storing the details of a student, we can create a structure for a student that has the following data types: a character array for storing name, an integer for storing roll number, and a character for storing section, etc.
- ✓ Structures don't take up any space in the memory unless and until we define some variables for it. When we define its variables, they take up some memory space which depends upon the type of the data member and alignment

# Why do we use Structures in C?

- Structure in C programming is very helpful in cases where we need to store similar data of multiple entities. Let us understand the need for structures with a real-life example.
  - Suppose you need to manage the record of books in a library. Now a book can have properties like book_name, author_name, and genre. So, for any book you need three variables to store its records. Now, there are two ways to achieve this goal.
  - The first and the naive one is to create separate variables for each book. But creating so many variables and assigning values to each of them is impractical. So what would be the optimized and ideal approach? Here, comes the structure in the picture.
  - We can define a structure where we can declare the data members of different data types according to our needs. In this case, a structure named BOOK can be created having three members book_name, author_name, and genre. Multiple variables of the type BOOK can be created such as book1, book2, and so on (each will have its own copy of the three members book_name, author_name, and genre).

# How to Create a Structure?

To create a structure in C, the struct keyword is used followed by the tag name of the structure. Then the body of the structure is defined, in which the required data members (primitive or user-defined data types) are added.

Syntax:

```
struct structure_name
{
    Data_member_type data_member_defination;
    Data_member_type data_member_defination;
    Data_member_type data_member_defination;
    ...
    ...
}(structure_variables);
```

Example:

```
struct Student
{
    char name[50];
    int class;
    int roll_no;
} student1;
```

In the above syntax, the data_members can be of any data type like int, char, double, array or even any other user-defined data type. The data_member_definition for the data types like character array, int, and double is just a variable name like name, class, and roll_no. We have also declared a variable, i.e., student1 of the Student structure.

# Declaration of Structure Variables

A structure variable can either be declared with structure declaration or as a separate declaration like basic types.

Method 1:

```
struct Student
{
    char name[50];
    int class;
    int roll_no;
} student1;
```

If we declare the structure variables with the structure definition, they work as global variables(means they can be accessed in the whole program). If we need global variables, we can declare variables with the structure otherwise declaring it using the second approach is the best way as it is easy to maintain or initialize variables.

Method 2:

```
struct Student
{
    char name[50];
    int class;
    int roll_no;
};

int main()
{
    //struct structure_name variable_name;

    struct Student a; // here a is the variable of type Student
    return 0;
}
```

# Initialization of Structure Members

There are three ways to initialize structure members:

- ❖ Using **dot '.' operator**
- ❖ Using **curly braces '{}'**
- ❖ **Designated initializers**

**1. Using Dot '.' operator**

Using the dot (.) operator, we can access any structure member and then initialize or assign its value according to its data type.

```
struct structure_name variable_name;

variable_name.member = value;
```

In the above syntax first, we created a structure variable, then with the help of the dot operator accessed its member to initialize them.

# Dot operator - Example

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

struct Student
{
 char name[50];
 int class;
 char section;
};

int main()
{
  // created variable student1 for structure Student
  struct Student student1;

  // accessing student1 member and initializing them
 strcpy(student1.name,"Student_name");
 student1.class = 1;
 student1.section = 'A';

 // printing values
 printf( "Student Name : %s\n", student1.name);
 printf( "Student Class : %d\n", student1.class);
 printf( "Student Section : %c\n", student1.section);

 return 0;
}
```

1. In the given Example, we have created a structure, Student and declared some members in it.
2. After that, we created an instance (variable or object of structure Student) for it to access the structure members using the dot operator and assigned value to them.
3. Also, we used strcpy method of the string, this is used to assign the value of one string to another.
4. In the end, we print the values of structure members with the help of the dot operator.

```
Student Name : Student_name
Student Class : 1
Student Section : A
```

# Using Curly braces { }

If we want to initialize all the members during the structure variable declaration, we can declare using curly braces.

```
struct stucture_name v1 = {value, value, value, ..};
```

To initialize the data members by this method, the comma-separated values should be provided in the same order as the members declared in the structure. Also, this method is beneficial to use when we have to initialize all the data members.

**Example**

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

struct Student
{
  char name[50];
  int class;
  char section;
};

int main()
{
   // created variable student1 for structure Student
   struct Student student1 = {"Student_name" , 1, 'A'};

  // printing values
  printf( "Student Name : %s\n", student1.name);
  printf( "Student Class : %d\n", student1.class);
  printf( "Student Section : %c\n", student1.section);

  return 0;
}
```

```
Student Name : Student_name
Student Class : 1
Student Section: A
```

# Designated Initializers

Designated initialization is simple initialization of the structure members and is normally used when we want to initialize only a few structure members, not all of them.

```
struct strucutre_name structure_varialbe = {.structure_member = value,.structure_member = value};
```

From syntax, we can see that we use curly braces, and in between them, with the help of the dot operator, data members are accessed and initialized. There can be any number of structure members from a single structure that we can initialize, and all of them are separated using commas. But the most important thing is that we can initialize members in any order. It is not compulsory to maintain the same order as the members are declared in the structure.

```c
#include <stdio.h>

// creating a structure
struct Student
{
     char name[50];
     int class;
     char section;
};

int main ()
{
    // creating a structure variable and initialzing some of its members
    struct Student student1 = {.section = 'B', .class = 6};

    // printing values
    printf("Student1 Class is: %d\n",student1.class);
    printf("Student1 Section is: %c",student1.section);
}
```

```
Student1 Class is: 6
Student1 Section is: B
```

# Structure as Function Argument

While passing structure as a function argument, structure variables are treated the same as variables of primitive data types.

The basic syntax for passing structure as a function argument is

```
// passing by value
returnTypeOfFunction functionName (struct sturcture_name variable_name);
functionName(vaiable_name);

// passing by reference
returnTypeOfFunction functionName (struct structure_name* varible_name);
functionName(&variable_name);
```

# Example

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

struct Student
{
  char name[50];
  char section;
  int class;
};

// pass by value
void printStudent(struct Student var) {

  printf("Student name : %s\n", var.name);
  printf("Student section : %c\n", var.section);
  printf("Student class : %d\n", var.class);
}

// pass by reference
void changeStudent(struct Student* var)
{

    var->class = 6;
    var->section = 'B';
}
```

```c
int main(){
    struct Student student1 = {"student_name", 'A', 5};  // initialising the object


    // passing by value
    printStudent(student1);


    // passing by reference
    changeStudent(&student1);


  return 0;
}
```

```
Student name : student_name
Student section : A
Student class : 5
```

# Array of Structure

As a structure in C is a user-defined data type, we can also create an array of it, same as other data types. The syntax is

```
struct structure_name array_name[size_of_array];
```

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

 struct Student
{
    char name[50];
    char section;
    int class;
};

int main()
{
    // creating an array of structures

    struct Student arr[5];

    // initializing every student with similar class and section

    for(int i=0;i<5;i++)
    {
        scanf("%s",arr[i].name);
        arr[i].section = 'A'+i;
        arr[i].class = i+1;
        printf("name: %s section: %c class: %d\n",arr[i].name,arr[i]
                                                  .section,arr[i].class);
    } return 0;  }
```

**Input:**

```
student1
student2
student3
student4
student5
```

**Output:**

```
name: student1 section: A class: 1
name: student2 section: B class: 2
name: student3 section: C class: 3
name: student4 section: D class: 4
name: student5 section: E class: 5
```

# Structure Pointer

Like primitive types, we can have a pointer to a structure. If we have a pointer to structure, members are accessed using arrow ( -> ) operator.

```c
#include <stdio.h>

struct Point {
    int x, y;
};

int main()
{
    struct Point p1 = { 1, 2 };

    // p2 is a pointer to structure p1
    struct Point* p2 = &p1;

    // Accessing structure members using structure pointer
    printf("%d %d", p2->x, p2->y);
    return 0;
}
```

```
1 2
```

# Limitations of C Structures

In C language, Structures provide a method for packing together data of different types. A Structure is a helpful tool to handle a group of logically related data items. However, C structures have some limitations.

1. The C structure does not allow the struct data type to be treated like built-in data types.
2. We cannot use operators like +,- etc. on Structure variables.
3. No Data Hiding: C Structures do not permit data hiding. Structure members can be accessed by any function, anywhere in the scope of the Structure
4. Functions inside Structure: C structures do not permit functions inside Structure
5. Static Members: C Structures cannot have static members inside their body
6. Access Modifiers: C Programming language does not support access modifiers. So they cannot be used in C Structures.
7. Construction creation in Structure: Structures in C cannot have a constructor inside Structures.