Lecture 3: Structures

In this lecture, the following topics are covered:

- ❑ The concept of structure in C
- ❑ Arrays of Structures
- ❑ Structures and Functions
- ❑ Self-Referential Structures

1. The Concept of Structure in C

- Structure is basically a user-defined data type that can store related information (even of different data types) together.
- The major difference between a structure and an array is that an array can store only information of same data type.
- A structure is therefore a collection of variables under a single name.
- The variables within a structure are of different data types and each has a name that is used to select it from the structure

1.1 Structure Declaration

- A structure is declared using the keyword **struct** followed by the structure name.
- All the variables of the structure are declared within the structure.
- A structure type is generally declared by using the following syntax:

```
struct struct–name {
     data_type var–name;
    data_type var–name;
     ...............

  };
```

For example, if we have to define a structure for a student, then the related information for a student probably would be: **id_num**, **name**, **course**, and **fees**. This structure can be declared as:

```
struct student {
    int id_num;
    char name[20];
    char course[20];
    float fees;
};
```

- Now the structure has become a user-defined data type.
- Each variable name declared within a structure is called a member of the structure.
- The structure declaration, however, does not allocate any memory or consume storage space. It just gives a template that conveys to the C compiler how the structure would be laid out in the memory and also gives the details of member names.
- Like any other data type, **memory is allocated for the structure when we declare a variable of the structure**.

  For example, we can define a variable of student by writing:

  ```
  struct student stud1;
  ```

  Here, **struct student** is a **data type** and **stud1** is a variable.
- Look at another way of declaring variables. In the following syntax, the variables are declared at the time of structure declaration.

  ```
  struct student {
          int id_num;
          char name[20];
          char course[20];
          float fees;
  } stud1, stud2;
  ```

  In this declaration we declare two variables stud1 and stud2 of the structure student. So if you want to declare more than one variable of the structure, then separate the variables using a comma.

  When we declare variables of the structure, separate memory is allocated for each variable.

*Structure type and variable declaration of a structure can be either local or global depending on their placement in the code

1.2 Typedef Declarations

- The **typedef** (derived from type definition) keyword enables the programmer to create a new data type name by using an existing data type.
- By using typedef, no new data is created, rather an alternate name is given to a known data type.
- The general syntax of using the **typedef** keyword is given as:

  typedef **existing_data_type new_data_type;**
- Note that typedef statement does not occupy any memory; it simply defines a new type.
- For example, if we write typedef int INTEGER; then INTEGER is the new name of data type int.
- To declare variables using the new data type name, precede the variable name with the data type name (new). Therefore, to define an integer variable, we may now write INTEGER num=5;
- When we precede a **struct name** with the **typedef** keyword, then the **struct** becomes a **new type**. It is used to make the construct shorter with more meaningful names for types already defined by C or for types that you have declared. For example, consider the following declaration:

```
typedef struct student {
        int id_num;
        char name[20];
        char course[20];
        float fees;
};
```

Now that you have preceded the structure's name with the **typedef** keyword, **student becomes a new data type**. Therefore, now you can straightaway declare the variables of this new data type as you declare the variables of type int, float, char, double, etc. To declare a variable of structure student, you may write

student stud1;

Note that we have not written **struct student** stud1.

1.3 Initialization of Structures

- 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 it. For int and float members, the values are initialized to zero, and char and string members are initialized to '\0' by default.
- The initializers are enclosed in braces and are separated by commas. However, care must be taken to ensure that the initializers match their corresponding types in the structure definition.
- The general syntax to initialize a structure variable is as follows:

```
    struct struct_name
    {
            data_type member_name1;
            data_type member_name2;
            data_type member_name3;
            .......................
    }struct_var = {constant1, constant2, constant3,...};
```

or

```
    struct struct_name
    {
            data_type member_name1;
            data_type member_name2;
            data_type member_name3;
            .......................
    };
struct struct_name struct_var = {constant1, constant2, constant 3,...};
```

For example, we can initialize a student structure by writing,

```
struct student {
        int id_num;
        char name[20];
        char course[20];
        float fees;
} stud1 = {120, "Ahmed Ali", "B.Sc in IT", 2500.500};
```

Or, by writing,

struct student stud1 = **{**120, "Ahmed Ali", "B.Sc in IT", 2500.500**}**;

1.4 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: **struct_var**.**member_name**
- The dot operator is used to select a particular member of the structure. For example, to assign values to the individual data members of the structure variable **stud1**, we may write:
  stud1**.id_num** = 125;
  stud1**.name** = "Mohammed Ali";
  stud1**.course** = " B.Sc in Web Tech";
  stud1**.fees** = 3500.500;
- To input values for data members of the structure variable stud1, we may write:

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

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

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

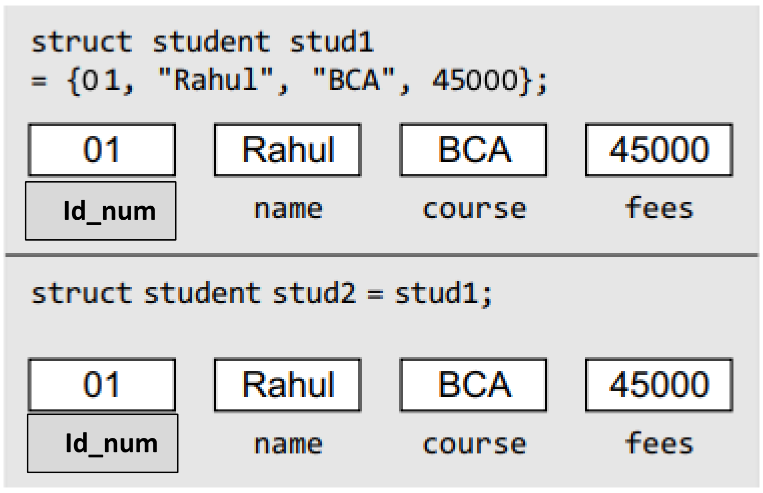
  ```
  printf ("%s", stud1.course);

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

1.5 Copying and Comparing Structures

- 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 = {**120, "Ahmed Ali", "B.Sc in IT", 2500.500**}**;
  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. Therefore, now the values of stud1 and stud2 can be given as shown in the following Figure:



```
struct student stud1
= {01, "Rahul", "BCA", 45000};
```

```
struct student stud2 = stud1;
```

- C does not permit comparison of one structure variable with another. However, individual members of one structure can be compared with individual members of another structure. When we compare one structure member with another structure's member, the comparison will behave like any other ordinary variable comparison. For example, to compare the fees of two students, we will write:

  If (stud1.fees > stud2.fees)

- **An error will be generated if you try to compare two structure variables.**

1.6 Nested structures

- A structure can be placed within another structure, i.e., a structure may contain another structure as its member.
- A structure that contains another structure as its member is called a nested structure.
- Although it is possible to declare a nested structure with one declaration, it is not recommended.
- The easier and clearer way is to declare the structures separately and then group them in the higher level structure. When you do this, take care to check that nesting must be done from inside out (from lowest level to the most inclusive level), i.e., declare the innermost structure,

then the next level structure, working towards the outer (most inclusive) structure.

```
typedef struct
{
      char first_name[20];
      char mid_name[20];
      char last_name[20];
}NAME;
typedef struct
{
      int dd;
      int mm;
      int yy;
}DATE;
typedef struct
{
      int r_no;
      NAME name;
      char course[20];
      DATE DOB;
      float fees;
} student;
```

In this example, we see that the structure student contains two other structures, NAME and DATE. Both these structures have their own fields. The structure NAME has three fields: first_name, mid_name, and last_name. The structure DATE also has three fields: dd, mm, and yy, which specify the day, month, and year of the date. Now, to assign values to the structure fields, we will write:

```
student stud1;
stud1.r_no = 01;
stud1.name.first_name = "Janak";
stud1.name.mid_name = "Raj";
stud1.name.last_name = "Thareja";
stud1.course = "BCA";
stud1.DOB.dd = 15;
stud1.DOB.mm = 09;
stud1.DOB.yy = 1990;
stud1.fees = 45000;
```

- In case of nested structures, we use the dot operator in conjunction with the structure variables to access the members of the innermost as well as the outermost structures.

- The use of nested structures is illustrated in the next program.

The following program is used to read and display the information of a student using a nested structure.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        struct DOB
        {
                int day;
                int month;
                int year;
        };
        struct student
        {
                int roll_no;
                char name[100];
                float fees;
                struct DOB date;
        };
        struct student stud1;
        clrscr();
        printf("\n Enter the roll number : ");
        scanf("%d", &stud1.roll_no);
        printf("\n Enter the name : ");
        scanf("%s", stud1.name);
        printf("\n Enter the fees : ");
        scanf("%f", &stud1.fees);
        printf("\n Enter the DOB : ");
        scanf("%d %d %d", &stud1.date.day, &stud1.date.month, &stud1.date.year);
        printf("\n ********STUDENT'S DETAILS *******");
        printf("\n ROLL No. = %d", stud1.roll_no);
        printf("\n NAME = %s", stud1.name);
        printf("\n FEES = %f", stud1.fees);
        printf("\n DOB = %d - %d - %d", stud1.date.day, stud1.date.month, stud1.date.year);
        getch();
        return 0;
 }
```

The output:

```
Enter the roll number : 01
Enter the name : Rahul
Enter the fees : 45000
Enter the DOB : 25 09 1991
********STUDENT'S DETAILS *******
ROLL No. = 01
NAME = Rahul
FEES = 45000.00
DOB = 25 - 09 - 1991
```

2. Arrays of Structures

- Now, we will discuss how an array of structures is declared.
- For this purpose, let us first analyze where we would need an array of structures.
  - In a class, we do not have just one student. But there may be at least 30 students. So, the same definition of the structure can be used for all the 30 students. This would be possible when we make an array of structures. An array of structures is declared in the same way as we declare an array of a built-in data type.
  - Another example where an array of structures is desirable is in case of an organization. An organization has a number of employees. So, defining a separate structure for every employee is not a viable solution. So, here we can have a common structure definition for all the employees.
- The general syntax for declaring an array of structures can be given as:

```
struct struct_name
{
        data_type member_name1;
        data_type member_name2;
        data_type member_name3;
        .......................
};
```

Consider the given structure definition:

```
struct student
{
        int r_no;
        char name[20];
        char course[20];
        float fees;
};
```

A student array can be declared by writing:

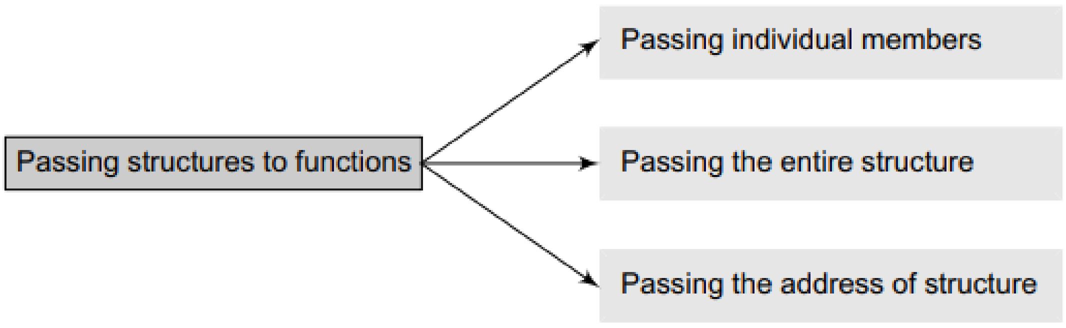
```
struct student stud[30];
```

Now, to assign values to the **ith** student of the class, we will write:
stud[i].id_num = 09;
stud[i].name = "Khalid Ali";
stud[i].course = "MBA";
stud[i].fees = 12000;

- In order to initialize the array of structure variables at the time of declaration, we can write as follows:
  struct student stud[3] = {{01, "Aman", "BCA", 45000},{02, "Aryan", "BCA", 60000}, {03, "John", "BCA", 45000}};

3. Structures and Functions

- For structures to be fully useful, we must have a mechanism to pass them to functions and return them.
- A function may access the members of a structure in three ways as shown in the following figure:


Passing structures to functions
Passing individual members
Passing the entire structure
Passing the address of structure

3.1 Passing Individual Members

- To pass any individual member of a structure to a function, we must use the direct selection operator to refer to the individual members.
- The called program does not know if a variable is an ordinary variable or a structured member.
- Look at the code given below which illustrates this concept.

```
#include <stdio.h>
typedef struct
{
      int x;
      int y;
}POINT;
void display(int, int);
int main()
{
      POINT p1 = {2, 3};
      display(p1.x, p1.y);
      return 0;
}
void display(int a, int b)
{
      printf(" The coordinates of the point are: %d %d", a, b);
}
```

3.2 Passing the Entire Structure

- Just like any other variable, we can pass an entire structure as a function argument.
- When a structure is passed as an argument, it is passed using the call by value method, i.e., a copy of each member of the structure is made.
- The general syntax for passing a structure to a function and returning a structure can be given as

  ```
  struct struct_name func_name(struct struct_name struct_var);
  ```

  The above syntax can vary as per the requirement. For example, in some situations, we may want a function to receive a structure but return a void or the value of some other data type.
- The code given below passes a structure to a function using the call by value method:

```
#include <stdio.h>
typedef struct
{
      int x;
      int y;
}POINT;
```

```
void display(POINT);
int main()
{
        POINT p1 = {2, 3};
        display(p1);
        return 0;
}
void display(POINT p)
{
        printf("The coordinates of the point are: %d %d", p.x, p.y);
}
```

Summary of some points that must be considered while passing a structure to the called function.

- If the called function is returning a copy of the entire structure, then it must be declared as struct followed by the structure name.
- The structure variable used as parameter in the function declaration must be the same as that of the actual argument in the called function (and that should be the name of the struct type).
- When a function returns a structure, then in the calling function the returned structure must be assigned to a structure variable of the same type.

3.3 Passing Structures through Pointers

- Passing large structures to functions using the call by value method is very inefficient. Therefore, it is preferred to pass structures through pointers.
- It is possible to create a pointer to almost any type in C, including the user-defined types.
- It is extremely common to create pointers to structures.
- Like in other cases, a pointer to a structure is never itself a structure, but merely a variable that holds the address of a structure.
- The syntax to declare a pointer to a structure can be given as,

```
struct struct_name
{
        data_type member_name1;
        data_type member_name2;
        data_type member_name3;
        .......................
}*ptr;
```

or **struct** struct_name ***ptr;**

For our student structure, we can declare a pointer variable by writing:

struct student ***ptr_stud, stud;**

The next thing to do is to assign the address of stud to the pointer using the address operator (&), as we would do in case of any other pointer. So to assign the address, we will write

ptr_stud = **&**stud;

- To access the members of a structure, we can write:

(***ptr_stud).id_num;

Since parentheses have a higher precedence than *, writing this statement would work well. But this statement is not easy to work with, especially for a beginner. So, C introduces a new operator to do the same task. This operator is known as 'pointing-to' operator (->). It can be used as:

ptr_stud **->** id_num = 120;

4. Self-Referential Structures

- Self-referential structures are those structures that contain a reference to the data of its same type.
- That is, a self-referential structure, in addition to other data, contains a pointer to a data that is of the same type as that of the structure. For example, consider the structure node given below:

```
struct node
    {
          int val;
          struct node *next;
    };
```

Here, the structure node will contain two types of data: an integer **val** and a pointer **next**. You must be wondering why we need such a structure. Actually, self-referential structure is the foundation of other data structures.