Structures and Self-referential structures

Introduction

- Structure is a user-defined data type that can store related information (even of different data types) together.
- A structure is declared using the keyword struct followed by a structure name.
- All the variables of a structure are declared within the structure.
- A structure type is defined by using the following syntax:

```
struct struct-name
{     data_type var-name;
      data_type var-name;
      ...
};
```

Introduction

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

- The structure definition does not allocate any memory.
- It just gives a template that conveys to the compiler how the structure is laid out in memory and gives details of the members.
- 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;
```

Typedef Declaration

- When we precede a struct name with typedef keyword, then the struct becomes a new type.
- For example, consider the following declaration:

```
typedef struct stud {
    int r_no;
    char name[20];
} student;
```

- Now we can straightaway declare variables of this new data type as we declare variables of type int, float, char, double, etc.
- To declare a variable of structure student, we will just write:

```
student stud1;
```

Initializing Structures

- Initializing a structure means assigning some constants to the members of the structure.
- When the user does not explicitly initializes the structure then C automatically does that.
- 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.
- Note that initializers should match their corresponding types in the structure definition.

Initializing Structures

- The general syntax to initialize a structure variable is given as follows:

```
struct struct_name
{
 data_type member_name 1;
 data_type member_name 2;
 data_type member_name 3;
……..
}struct_var= {constant1, constant2, constant3, ….};
```

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 member of a structure is:

  struct_var.member_name

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

  stud1.r_no = 01;

Accessing the Members of a Structure

- 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

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

  struct student stud2;
- Then to assign one structure variable to another, we will write:

  stud2 = stud1;

Nested Structures

- A structure can be placed within another structure.
- Such a structure that contains another structure as its member is called a nested 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;
   DATE  DOB;
}student;
```

To assign values to the structure fields, we will write:

```
struct student stud1;
stud1.DOB.dd = 15;
stud1.DOB.mm = 03;
stud1.DOB.yy= 1990;
```

Arrays of Structures

- The general syntax for declaring an array of structure can be given as:

  struct struct_name struct_var[index];

  struct student stud[30];

- Now, to assign values to the i^{th} student of the class, we will write:

  stud[i].r_no = 09;

  stud[i].name = “RASHI”;

  stud[i].course = “MCA”;

  stud[i].fees = 60000;

Passing Individual Structure Members to a Function

- To pass any individual member of the structure to a function we must use the direct selection operator to refer to the individual members for the actual parameters.
- The called program does not know if the two variables are ordinary variables or structure members.

```
typedef struct
{    int x;
     int y;
}POINT;
POINT p1={2,3};
display(p1.x, p1.y); //passing members of p1 to function display()
```

Passing a Structure to a Function

- When a structure is passed as an argument, it is passed using call by value method. That is 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);

```
typedef struct
{    int x;
     int y;
}POINT;
POINT p1={2,3};
display(p1); //passing entire structure p1 to function display()
```

Access Structures through Pointers

- C allows to create a pointer to a structure.
- 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;
    ....................................
}*ptr;
```

OR

```
struct struct_name *ptr;
```

Access Structures through Pointers

- For our student structure we can declare a pointer variable (ptr_stud) and a normal data variable(stud) by writing:

  struct student *ptr_stud, stud;

- The next step is to assign the address of stud to the pointer using the address operator (&). So to assign the address, we will write:

  ptr_stud = &stud;

- To access the members of the structure, one way is to write:

  (*ptr_stud).roll_no;

- An alternative to the above statement can be used by using 'pointing-to' operator (->):

  ptr_stud->roll_no = 01;

Self-referential Structures

- Self-referential structures are those structures that contain a reference to data of its same type.
- That is, a self-referential structure contains a pointer to a data that is of the same type as that of the structure.

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

- Here the structure node contains two types of data: an integer *val* and *next* that is a pointer to a node. You must be wondering why do we need such a structure? Actually, self-referential structure is the foundation of other data structures linked lists.