STRUCTURE AND UNION IN C

Module – 1

Dear friends,

In this session, we will discuss two very interesting and useful features of C programming language named structure and union. So far we have been discussing the primitive data types and some of the derived data types and using them in C programs. In a larger program, sometimes it may be necessary to organize a group of data items of different data types referring to a single entity. However we cannot use an array if we want to represent a collection of data items of different types using a single name. A structure helps to define such a data types in C. Each data item is a member or field of the structure.

Structures are slightly different from the variable types you have been using till now. Structures are data types by themselves. When you define a structure or union, you are creating a custom data type. For example, A passenger who wants to reserve a railway ticket has to furnish the details like, Name, Sex, Age, train number, Boarding place, Destination, date of journey. It will be convenient to handle all these details of a passenger as a single entity in a program. This information may be maintained for every passenger for further processing. Arrays can hold only data of similar data types and hence they cannot hold information having different data types. The need for structure arises in such situations.

Structure Declaration

A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. Structures help to organize complicated data, particularly in large programs, because they permit a group of related variables to be treated as a unit instead of separate entities. Structures have declarations and definitions. The general format of a structure declaration is,

```
struct tag
{
        Declaration of member 1;
        Declaration of member 2;
        ------------
        -------------
        Declaration of member n;
};
```

The declaration begins with the keyword **struct**. The list of declarations of its members must be enclosed in braces. The **tag** is a name that identifies structures of this type. The individual members can be ordinary variables, pointers, arrays or other structures. The member names within a particular structure must be distinct from one another, though a member name can be same as the name of a variable defined outside the structure.
For example, suppose we want to store data about a passenger, which consist of passenger name (a string), age(an integer), train number(an integer), boarding place(a string) and destination (a string) so on. Let us see the structure declaration now.

```
struct passenger
{
        char name [20];
        int age;
        int train_no;
        char board_place[20];
        char destn[20];
};
```

This declaration uses passenger as a tag. Observe the semicolon after the closing brace. There are five members in this structure declaration. Tag name may be used as an ordinary variable name or a member variable name without any conflict in a program. After the declaration of a structure, a structure variable can be created by defining it as

```
struct tag var1, var2,…., varn;

 for example struct passenger pas1, pas2;     /* Structure variable definition */
```

defines two structures variables pas1 and pas2 of strcut data type. The structure definition reserves spaces for the structure variables. The members are stored in memory in the order they are declared in the structure declaration. There are different ways to define structure variables.

1.
   ```
   struct tag
   {
   Member declarations;                          /* structure declaration */
   };

   Struct tag var1, var2, …., varn;              /* structure definition */
   ```

2.
   ```
   struct tag
   {
   Member declarations;                          /* structure declaration */
   };

   typedef struct tag NEWNAME;                   /* renaming the structure  */
   NEWNAME var1, var2,…..,varn;                  /* structure definition */
   ```

3.
   ```
   typedef struct tag             /* structure declaration with renaming */
   {
   Member declarations;
   }  NEWNAME;

   NEWNAME var1, var2,…..,varn;                  /* structure definition */
   ```

4.
   ```
   typedef struct                 /* structure declaration with renaming */
   {
   Member declarations;
   }  NEWNAME;

   NEWNAME var1, var2,…..,varn;                  /* structure definition */
   ```

5.
   ```
   struct tag
   {
   Member declarations;                         /* structure declaration */
   } var1, var2,….., varn;                       /* structure definition */
   ```

6.
   ```
   struct
   {
   Member declarations;
   }var1, var2, …., varn;                       /* structure definition */
   ```

Observe the declarations and definitions carefully. There should be at least one blank space after the closing brace in the third and fourth methods.

It is very important to remember that

- Only after the definition, the structure variables are created and storage space are reserved.
- The structure declaration should be terminated with a semicolon.

Initializing structure elements

A structure element can be initialized in its definition itself with a list of initializes enclosed within the braces. Each initializer must be a constant expression and the order and type of each member must match the order and type of its declaration. For example:

```
struct student
{
        char name [20];
        int rollno;
        float fees;
        char division;
```

```
} ;
struct student s1 = {“arun”, 11, 3200, ‘A’};
struct student s2 = {“ram”, 27, 4000, ‘C};
```

A structure variable may be initilaised by means of assigning another structure variable of similar type. For example,

Struct student s1=s2;

Assigns the data of s1 to s2. But the structure variable at the right side of the assignment statement has to be defined and initialized or read before the assignment.

The initialization method shown is the screen is also a valid one.

```
struct student
{
        char name [20];
        int rollno;
        float fees;
        char division;
} s1 = {“arun”, 11, 3200, ‘A’};
```

Here the declaration and definition are combined. While declaring a structure, it is very important to remember that the member variables can not be initialized. This is because a structure declaration does not reserve nay memory space.

Accessing Structure Elements

Having declared the structure type and the structure variables, let us see how the elements of the structure can be accessed. In arrays we can access individual elements of an array using a subscript. Structures use a different scheme. They use a dot (.) operator. So to refer to **name** of the structure defined in our sample program we have to use,

s1.name

Similarly, to refer to **rollno** we would use,

s1.rollno

Please note that before the dot there must always be a structure variable and after the dot there must always be a structure element.

The corresponding C statements will be;

strcpy(s1.name, “Ravi”);

s1.roll_no = 13;

……….

How Structure Elements are Stored

Whatever be the elements of a structure, they are always stored in contiguous memory locations. Let me now illustrate the concept using a simple program.

```
/* Memory map of structure elements */

#include<stdio.h>
main( )
{
        struct book
        {
                char name[20] ;
                float price ;
                int pages ;
        } ;
struct book b1 = { “book1”, 75.00, 225 } ;
printf ( "\nAddress of name = %u", &b1.name ) ;
printf ( "\nAddress of price = %u", &b1.price ) ;
printf ( "\nAddress of pages = %u", &b1.pages ) ;
}
```

Here is the output of the program...

```
Address of name = 64318
Address of price = 64339
Address of pages = 64344
```

The memory map for the structure declared in the program will be as shown in the screen.

b1.name	b1.price	b1.pages
book1	75.00	225
64318	64339	64344

Copying and comparing structure variable

We have already discussed that two variables of same structure type can be copied the same way as ordinary variables. If *student1* and *student2* belong to the same structure, then the following statements are valid:

student1 = student2;

student2 = student1;

However, the statements such as

student1 == student2
student1 != student2

are not permitted. C does not permit any logical operations of structure variables. In case we need to compare them, we may do so by comparing members individually.

Nested Structure

If a structure contains one or more structures as its members, it is known as nested structure. Let us consider an example that declares a nested structure to store bate of birth of a student.

We can declare a structure to store date as

```
struct date
{
        int dd;
        int mm;
        int yy;
};
```

This structure date can be used as a member in another structure as shown in the screen.

```
struct student
{
        char name[20];
        int rollno;
        float fees;
        struct date dob;
} s;
```

The structure date has been made as the member of structure student. To access the elements of the structure date, which is the part of another structure, we can the statements lik,

s.dob.dd;

s.dob.mm;

s.dob.yy;

Module – 2

Array of structures

A group of structures may be organized in an array resulting in an array of structures. Each element in the array is a structure.

Let us now consider a problem where we need to store data of 100 students. Here we would be required to use 100 different structure variables from **s1** to

s100, which is definitely not very convenient. A better approach would be to use an array of structures. Let us now go through a program segment that shows how to use an array of structures.

```
………….
struct student
{
        int rollno;
        float fees;
        char division;
} s[100];
```

Here we define an array of 100 structures. Each structure variable s[0], s[1],…., s[100] contains structure as its values. The members can be accessed as

```
S[0].rollno
S[1].fees
.
.
.
S[100].rollno
S[100].fees
```

Now let us see a program segment that reads and writes the above defined array of structure.

```
int i;

for ( i = 0 ; i <= 99 ; i++ )
{
        printf ( "\nEnter roll_no, fees and division " ) ;
        scanf ( "%d %f %c", &s[i].rollno, &s[i].fees, &s[i].division ) ;
}
for ( i = 0 ; i <= 99 ; i++ )
        printf ( "%d %f %c", s[i].rollno, s[i].fees, s[i].division ) ;
…………..
```

Pointer to structure

A pointer is helpful to create a structure dynamically. A group of structures may also be created dynamically using a pointer. A pointer to a structure is similar to a pointer to as an ordinary variable. It is created in the same way that a pointer to an ordinary variable is created. For example,

```
typedef struct stud
{
        Int regno;
        char name [20];
        char result[10];

}  STUDENT;
```

Declares a structure STUDENT. The definition

STUDENT *sp;
Creates a pointer variable sp which points to the structure STUDENT. After the creation of the pointer variable, it should be assigned with a suitable pointer value. For example,

sp= (STUDENT *)malloc(sizeof(int) + 20 + 10);

assigns a pointer to a block of memory large enough to store an integer and 30 characters which are members of the structure. It is very tedious to calculate the size of the structure in this way if the members are more. The operator sizeof may be used to find the size of a structure as given below.

sp=(STUDENT *)malloc(sizeof(STUDENT));

Observe the cast operator to convert the generic pointer returned by the function malloc() into a pointer to the structure STUDENT. After assigning the appropriate pointer value, the expressions (*sp).regno and (*sp).name may be used to access the first two members of the structure. An alternate notation is also provided in C for pointer to structure. The operator -> (minus sign immediately followed by >) can be used to refer to a member. Now, (*sp).regno can be written as sp->regno and (*sp).name can be written as sp->name.

Module – 3

Structure and function

A structure can also be used in functions. While using the same structure in different functions, it is essentials to pass the structure as an argument. C supports the passing of structure values as arguments to functions. The structure variable or a pointer to the structure may be passed.
In the first method, we pass each member of structure as an actual argument of the function call. The actual arguments are then treated independently like ordinary variables. This is the most elementary method and becomes unmanageable and inefficient when the structure size is large. When the structure variable is passed the changes made in the members of the structure are available only in the called function (call by value)
In the second method, we pass the pointer to the function. Here we pass the copy of the entire structure to the called function. The function is working on a copy of the structure, any changes to structure members within the function are not reflected in the original structure (effect of pass by reference).

Let us now consider an example for passing entire structure to a function.

Example : passing entire structure to a function

struct student

```
{
        char name[20];
        int rollno;
        float fees;
};
```

```
void display(struct student s)
{
        printf("Name : %s" ,s.name);
        printf("Rollno: %d" ,s.rollno);
        printf("Fees : %f" ,s.fees);
}
```

```
#include<stdio.h>
main()
{
        struct student s = {"Arun",12,3500};
        display(s);

}
```

Self referential structures

It is a structure which contains one member which is a pointer of its own type. For example,

```
     struct student
{
        char name[20];
        int rollno;
        float fees;
        struct student *next;
};
```

Here the structure contains an element which is a pointer to a structure of the same type called next. There for this is a self referential structure. Self referential structures are very useful in applications that involve linked data structures, such as list and trees. Here the declaration, assignment of pointer value and accessing the data using the pointer must be done appropriately.

Module – 4

Unions

A union can be considered as a special type of structure. It is a derived data type that permits different types of data items in which each member shares the same block of memory. A union behaves as a storage buffer capable of holding different data types. In structure each member has its own storage location, whereas all the members of a union use the same location. That is even though the union may contain many members of different types, it can handle only one member at a time. Like structure, a union can be declared using the keyword union. For example,

```
union mixed_type    /* union declaration */
{
        int a;
        float b;
        char c;
};

union mixed_type mt;  /* union definition */
```

This declares a variable mt of type union mixed_type. The union contains three members, each with a different data type including mt.a, mt.b and mt.c. The variable mt will be large enough to hold the largest of the three data types. However we can use only one of them at a time. This is due to the fact that only one location is allocated for a union variable, irrespective of its size.

Initialization of a union variable is restricted so that the value of its first member can only be initialized. For example,

```
union mixed_type
{
        int a;
        float b;

} mix=5;                /*initialization*/
```

Here a float value 5.0 would be stored in the member mix.a.

Unions are often used as members of structures and structures are used as members of unions. An array of unions, a pointer to a union, functions using a union are also possible in C. Also, a nested union and a self-referential union can be used. The notation for accessing a member of a union is identical to that of a structure.

Let's summarize what we have discussed in this session ,

A structure is a derived data type used to organize a group of related data items having different data types. The syntax of the structure declaration differs from that of a structure definition. The declaration informs the compiler about the prototype of the structure, whereas the definition creates the structure variable. The declaration alone does not reserve space, whereas the definitions allocate space for storing the members. The members of the structure are accessed by the structure member operator, denoted by a dot (.).

A nested structure allows another structure as its member. An array of structures is used to organize a group of structures. A pointer to a structure is helpful in the creation and manipulation of a structure or a group of structures. To access a member using a pointer to structure, the operator -> is used. A self-referential structure is possible by having a member of a structure as a pointer to itself.
A union is similar to a structure, except that only one member may be stored at a time in a union variable. Certain programs may be written using unions to conserve memory.

Please try to solve these **Assignments.**

1. What are structures? Differentiate a structure from an array.
2. Explain how structures are defined in C and how they are passed to functions?
3. What do a nested structure and an array of structures mean?
4. Differentiate between a structure and an union with respect to allocation of memory by the compiler. Give suitable examples.
5. Write a C program using a structure to print the mark list for a class of 30 students studying 5 different subjects using separate function for input and output.

Here are some books for your **Reference.**

1.B. W. Kernighan and D. M. Ritchie, The C Programming Language, 2/e, Prentice-Hall, 1988.

2.Yashavant Kanetkar; Let us C, 10th Edn.,BPB Publications, New Delhi, 2010.

3.Greg M Perry, Absolute Beginners' guide to C, 2/e SAMS publishing, 1994.

Hope now you are fascinated with the C language and interested in writing C programs. Wish you all good luck. Bye.