STRUCTURE AND UNION

UNIT STRUCTURE

Learning Objectives
Introduction
Structure
Defining a Structure
Structure Declaration
Giving Values to Members
Array of Structures
Structure within a Structure
Passing Structures to Functions
Pointer to Structures
Union
Enumerated Data Types
Defining Your Own Types (Typedef)
Let Us Sum Up

LEARNING OBJECTIVES

After going through this unit, you will be able to :

- use structure in a program
- learn how structures are defined and how their individual members are accessed and processed within a program
- declare structure variables
- learn about array of structures
- declare and use pointer to structure
- learn about union
- describe enumerated data type and type definition

INTRODUCTION

In the previous unit (i.e.,Unit 6) we have studied the array which is an example of data structure. It takes simple data types like *int*, *char* or *double* and organises them into a linear array of elements. The array serves most but not all of the needs of the typical C program. The restriction is that an array is composed of elements all of which are of the same type. If we need to use a collection of different data type items it is not possible by using an array. When we require using a collection of different data items of different data types we can use a *structure*.

This unit will help you to learn about *structure* and *union*, giving values to members, initializing structure, functions and structures, passing structure to elements to functions, arrays of structure, structure within a structure and union.

STRUCTURE

A structure is a heterogeneous user defined data type. It is a convenient method of handling a group of related data items of different data types. A structure contains a number of fields or variables. The variables can be of any of the valid data types. In order to use structures, we have to first define a unique structure.

Defining a Structure

Let us consider for a moment a book in detail which records *title*, *author, page* and *price*. The book name (i.e., title) and author name would have to be stored as **string**, i.e., array of chars terminated with an ASCII null character, and the page and price could be **int** and **float** respectively. At the moment, the only way we can work with this collection of data is as separate variables. This is not as convenient as a single data structure using a single name and so the C language provides **struct**. General format of defining a structure is as follows:

```
struct tag_name
{
        data_type  member1;
        data_type  member2;
        ................
        ...............
        data_type   membern;
};
```

In this declaration, ***struct*** is the keyword for structure; ***tag_name*** is the name that identifies sructure of particular type; and ***member1***, ***member2***,....***membern*** are individual member declarations. The individual members can be ordinary variables, pointers, arrays or other structures.

For example:

```
struct lib_books
{
        char title[25];
        char author[20];
        int pages;
        float price;
};
```

The keyword *struct* declares a structure to hold the details of four fields namely title, author, pages and price. These are members of the structures. Each member may belong to different or same data type. The *tag_name* can be used to define objects that have the tag names structure. It is not always necessary to define the structure within the *main()* function.

Structure Declaration

Once the composition of the structure has been defined, individual structure-type variable can be declared as followes:

struct tag_name variable 1, variable 2, ..., variable n;

where ***struct*** is a required keyword, ***tag_name*** is the name that appeared in the structure declaration, and ***variable 1***, ***variable 2***,............, ***variable n*** are structure variable of type ***tag_name***. We can declare structure variables using the tag_name any where in the program. For example, the statement,

```
struct lib_books book1, book2, book3;
```

declares *book1, book2, book3* as variables of type *struct lib_books* . Each declaration has four elements of the structure lib_books. The complete structure declaration might look like this:

```
struct lib_books
{
        char title[25], author[20];
        int pages;
        float price;
};
struct lib_books book1, book2, book3;
```

Structures do not occupy any memory until it is associated with the structure variable such as book1. The t*ag_name* such as *lib_books* can be used to declare structure variables of its data type later in the program. The *tag_name* such as *lib_books* can be used to declare structure variables of its data type later in the program.

We can also combine both template declaration and variables declaration in one statement, the declaration

```
struct lib_books
{
        char title[20];
        char author[15];
        int pages;
        float price;
} book1,book2,book3;
```

is valid. The use of tag_name is optional.

book1, *book2*, *book3* declare *book1*, *book2*, *book3* as structure variables representing three books but do not include a tag_name for use in the declaration. A structure is usually define before ***main()***. In such cases the structure assumes global status and all the functions can access the structure. For example:

```
struct employee
{
        char fname[15];
        char lname[15];
        int id_no;
        int bmonth;
        int bday;
        int byear;
} emp1;
```

Here, we have declared one variable, ***emp1***, to be structure with six fields, some integers, some strings. Right after the declaration, a portion of the main memory is reserved for the variable ***emp1****.* This variable takes a size of 38 bytes for different members of struct ***employee***: 15 bytes for fname, 15 bytes for lname, 2 bytes for id_no, 2 bytes for bmonth, 2 bytes for bday, 2bytes for byear.

Giving Values to Members

The members of structure themselves are not variables. They should be linked to structure variables in order to make them meaningful members. The link between a member and a variable is established using the member operator '.' Which is known as ***dot operator*** or ***period operator***.
For example,

```
book1.price;
book1.pages;
book2.price;
```

book1.price is the variable representing the price of *book1* and can be treated like any other ordinary variable. We can use *scanf()* function to assign values as follows:

```
scanf("%f",&book1.price);
scanf("%d",&book1.pages);
```

We can also assign values to the members of the structure *lib_books*. If we want to assign some values to *book1*, then the statements will be like this:

```
                strcpy(book1.title,"C Language");
                strcpy(book1.author,"Kanetkar");
                book1.pages=255;
                book1.price=325.00;
```

Program 1: Reading information of one student and displaying those information.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    struct studentinfo
    {
          int roll;
          char name[20];
          char address[30];
          int age;
     }s1;
     clrscr();
    printf("Enter the student information:");
    printf("\nEnter the student roll no.:");
     scanf("%d",&s1.roll);
    printf("\nEnter the name of the student:");
     scanf("%s",&s1.name);
    printf("\nEnter the address of the student:");
     scanf("%s",&s1.address);
    printf("\nEnter the age of the student:");
     scanf("%d",&s1.age);
    printf("\n\nStudent information:");
    printf("\nRoll no.:%d",s1.roll);
     printf("\nName:%s",s1.name);
     printf("\nAddress:%s",s1.address);
    printf("\nAge of student:%d",s1.age);
    getch();
}
```

The members of a structure variable can be assigned initial values in the same way as the lements of an array. The initial values must appear in the order in which they will be assigned to their corresponding structure members, enclosed in braces and separated by commas. The general syntax is as follows:

```
storage_class struct tag_name variable={value1, value2,...., value n};
```

where value1 refers to the value of the first structure member, value2 refers to the value of the second member, and so on. A structure variable, can be initialized only if its strorage class is either *static* or *external*.

```
struct date
 {
     int month;
     int day;
     int year;
 };
struct employee
 {
     char empid[5];
     char name[30];
     char dept[20];
     struct date dob;
 };
static struct employee e = {A25, ‘Rajib Dutta’, ‘Sales’, 6, 21,85 };
```

Here, e is a static data structure variable of type employee, whose members are assigned initial values. The first member (empid) is assigned the character value A25, the second member (name) is assigned the character value Rajib Dutta, the third member (dept) is assigned the character value Sales. The fourth member is itself a structure that contains three members (month, day and year). And the last member of customer is assigned the integer value 6, 21 and 85.

EXERCISE

Q. Modify the above program (Program 1) for storing and displaying information of two strudents.

ARRAY OF STRUCTURES

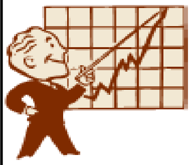
A useful program may need to handle many records. If we need to store a list of items, we can use an array as our data structure. In this case, the elements of the array are structures of a specified type. For example:

```
struct inventory
{
        int part_no;
        float cost;
        float price;
};
struct inventory table[4];
```

which defines an array with four elements, each of which is of type *struct inventory*, i.e., each is an inventory structure. Again, if we are maintaining information of all students in a school and if there are 100 students studying in the school, we need to use an array rather than single variables. It is possible to define an array of structures as shown in the program below :

Program 2:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    struct studentinfo
    {
        int roll;
        char name[20];
        char address[30];
        int age;
        };
       struct studentinfo s[100];
 /* s[100]  is an array of structure where information of maximum 100
      students can be stored */
        clrscr();
       int n,i;
      printf("\nHow many students information do you want to enter?");
       scanf("%d",&n);
      printf("Enter Student Information:");
       for(i=1;i<=n;i++)
       {
            printf("\nEnter Roll no.:");
            scanf("%d",&s[i].roll);
            printf("\nEnter the name of the student:");
            scanf("%s",&s[i].name);
            printf("\nEnter the address of the student:");
            scanf("%s",&s[i].address);
            printf("\nEnter the age of the student:");
            scanf("%d",&s[i].age);
       }
      printf("\n\nInformation of all studenst:");
       for(i=1;i<=n;i++)
       {
            printf("\nRoll no.:%d",s[i].roll);
            printf("\nName:%s",s[i].name);
            printf("\nAddress:%s",s[i].address);
            printf("\nAge of student:%d\n\n",s[i].age);
       }
       getch();
}
```



CHECK YOUR PROGRESS

1. Define a structure consisting of two floating point members, called *real* and *imaginary.* Include the tag *complex* within the definition. Declare the variables c1,c2 and c3 to be structure of type *complex*.

2. Declare a variable ***a*** to be a structure variable of the following structure type ***struct account*** {

```
                    int ac_no;
                    char ac_type;
                    char name[30];
                    float balance;
            };
```

and initiaze ***a*** as follows:

ac_no : 12437

ac_type: Saving

name: Rahul Anand

balance: 35000.00

3. State whether the following statements are true(T) or false(F)

(i) Collection of different datatypes can be used to form a struc ture.

(ii) Structure variables can be declared using the tag name any where in the program.

(iii) Tag-name is mandatory while defining a structure.

(iv) A program may not contain more than one structure.

(v) We cannot assign values to the members of a structure.

(vi) It is always necessary to define the structure variable within the main() function.

STRUCTURE WITHIN A STRUCTURE

A structure may be defined as a member of another structure. In such structures the declaration of the embedded structure must appear before the declarations of other structures. For example,

```
struct date
{
        int day;
        int month;
        int year;
};
struct student
{
        int roll;
        char name[20];
        char combination[3];
        int age;
        structure date dob;
}student1,student2;
```

the sturucture ***student*** constains another structure ***date*** as one of its members.

PASSING STRUCTURES TO FUNCTIONS

Structure variables may be passed as arguments and returned from functions just like other variables. A structure may be passed into a function as individual member or a separate variable. A program example to display the contents of a structure passing the individual elements to a function is shown below :

Program 2:

```
#include<stdio.h>
#include<conio.h>
void display(int,float);
void main()
{
    struct employee
    {
                        int emp_id;
                   char name[25];
                   char department[15];
                   float salary;
    };
   static struct employee e1={15, "Rahul","IT",8000.00};
    clrscr();
   /* only emp_id and salary are passed to the display fucntion*/
   display(e1.emp_id,e1.salary);
    getch();
}

void display(int eid, float s)
{
        printf("\n%d\t%5.2f",eid,s);
}
```

Output :

```
15   8000.00
```

When we call the display function using *display(e1.emp_id,e1.salary);* we are sending the emp_id and name to function display( ); it can be immediately realized that to pass individual elements would become more tedious as the number of structure elements go on increasing. A better way would be to pass the entire structure variable at a time.

Passing entire structure to functions :

There may be some structures having numerous structure members (elements). Passing these individual elements would be a tedious task. In such cases we may pass the whole structure to a function as shown below :

Program 3 :

```
#include<stdio.h>
#include<conio.h>
struct employee
{
        int emp_id;
```

```
            char name[25];
            char department[10];
            float salary;
};
static struct employee e1={12,"Dhruba","Sales",6000.00};
void display(struct employee e);  //prototype decleration
void main()
{
            clrscr();
            display(e1);    /*sending entire employee structure*/
            getch();
}
void display(struct employee e)
{
printf("%d\t%s\t%s\t%5.2f", e.emp_id,e.name,e.department,e.salary);
}
```

Output :

```
12      Dhruba          Sales       6000.00
```

Program 4: A program using structure working within a function

```
#include<stdio.h>
#include<conio.h>
struct  item{
            int code;
            float price;
};
struct item a;
void display(struct item i);  //prototype decleration
void main()
{
    clrscr();
   display(a);    /*sending entire employee structure*/
   getch();
}
void display(struct item i)
{
   i.code=20;
   i.price=299.99;
   printf("Item Code and Price of the item:%d\t%5.2f", i.code,i.price);
}
```

Output : 20   299.99

POINTER TO STRUCTURES

Instead of passing a copy of the whole structure to the function, we can pass only the address of the structure in the memory to the function. Then, the program will get access to every member in the function. This can be achieved by creating a pointer to the address of a structure using the indirection operator "*".

To write a program that can create and use pointer to structures, first, let us define a structure:

```
     struct item
    {
              int  code;
              float price;
    };
```

Now let us declare a pointer to struct type *item*.

```
    struct  item *ptr;
```

Because a pointer needs a memory address to point to, we must declare an instance of type *item*.

```
    struct item p;
```

The following figure shows the relationship between a structure and a pointer.

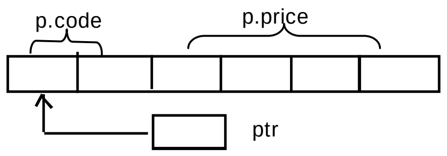
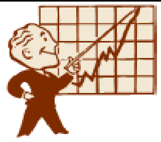


Fig.7.1: A pointer to a structure points to the first byte of the structure

Program 5: Program to demonstrate pointers to structure.

```
#include<stdio.h>
#include<conio.h>
void main()
{
        struct item
        {
                int code;
                float price;
        };
        struct item i;
        clrscr();
        struct item *ptr;       //declare pointer to ptr structure
        ptr=&i;                 // assign address of struct to ptr
        ptr->code=20;
        ptr->price=345.00;
        printf("\nItem Code: %d",ptr->code);
        printf("\tPrice: %5.2f",ptr->price);
        getch();
}
```

Output : Item Code: 20   Price: 345.00



CHECK YOUR PROGRESS

4.State whether the following statements are true(T) or false(F)

(i) It is possible to pass a structure to a function in the same way a variable is passed.

(ii)When one of the fields of a structure is itself a structure, it is called nested structure.

(iii) We cannot create structures within structure in C.

(iv) It is illegal for a structure to contain itself as a member.

(v) A sstructure can include one or more pointers as members.

5. Fill in the blanks:

(i) ___________ can be used to access the members of structure variables.

(ii) The name of a structure is referred to as ___________ .

UNION

In some applications, we might want to maintain information of one of two alternate forms. For example, suppose, we wish to store information about a person, and the person may be identified either by name or by an identification number, but never both at the same time. We could define a structure which has both an integer field and a string field; however, it seems wasteful to allocate memory for both fields. (This is particularly important if we are maintaining a very large list of persons, such as payroll information for a large company). In addition, we wish to use the same member name to access identity the information for a person.

C provides a data structure which fits our needs in this case called a *union* data type. A union type variable can store objects of different types at different times; however, at any given moment it stores an object of only one of the specified types. Unions are also similar to structure datatype except that members are overlaid one on top of another, so members of union data type share the same memory.

The declaration of a union type must specify all the possible different types that may be stored in the variable. The form of such a declaration is similar to declaring a structure template. For example, we can declare a union variable, person, with two members, a string and an integer. Here is the union declaration:

```
union human
{
        int id;
        char name[30];
 } person;
```

This declaration differs from a structure in that, when memory is allocated for the variable person, only enough memory is allocated to accommodate the largest of the specified types. The memory allocated for person will be large enough to store the larger of an integer or an 30 character array. Like structures, we can define a tag for the union, so the union template may be later referenced by the tag name:

Unions obey the same syntactic rules as structures. We can access elements with either the dot operator (.) or the right arrow operator (->). There are two basic applications for union. They are:

(i) Interpreting the same memory in different ways.
(ii) Creating flexible structure that can hold different types of data.

Program 6: Program demonstrating initializing the member and displaying the contents.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    union data
    {
         int a;
         float  b;
    };
    union data d;
    d.a=20;
    d.b= 195.25;
    printf("\nFirst member is %d",d.a);
    printf("\nSecond member is %5.2f",d.b);
    getch();
}
Output: First member is 16384
           Second member is 195.25
```

here only the float values are stored and displayed correctly and the interger values are displayed wrongly as the union only holds a value for one data type of the larger storage of their members.

ENUMERATED DATA TYPES

In addition to the predefined types such as int, char,float etc. , C allows us to define our own special data types, called enumerated data types.

An enumeration type is an integral type that is defined by the user . The syntax is:

enum typename {enumeration_list};

Here, *enum* is keyword, type stands for the identified that names the type being defined and *enumerator list* stands for a list of identifiers that define integer constants. For example:

```
enum color {yellow, green, red, blue, pink};
```

defines the type *color* which can then be used to declare variables like this:

```
color flower=pink;
color car[ ]={green, blue, red};
```

Here, *flower* is a variable whose value can be any one of the 5 values of the type *color* and is initialialized to have the value pink.

Program 7:

```
#include<stdio.h>
#include<conio.h>
void main()
{
   enum month { jan, feb, mar, apr, may, jun, jul, aug, sep,
   oct, nov, dec };
   month m;
    clrscr();
    for(m=jan;m<=dec;m++)
                   printf("%d\t", m+1);
     getch();
}
```

Output :
```
1   2   3   4   5   6   7   8   9   10   11   12
```

In the above declaration, *month* is declared as an enumerated data type. It consists of a set of values, jan to dec. Numerically, jan is given the value 1, feb the value 2, and so on. The variable *m* is declared to be of the same type as month, m cannot be assigned any values outside those specified in the initialization list for the declaration of month.

DEFINING YOUR OWN TYPES (TYPEDEF)

Using the keyword ***typedef*** we can rename basic or derived data types giving them names that may suit our program. A typedef declaration is a declaration with typedef as the storage class. The declarator becomes a new type. We can use typedef declarations to construct shorter or more meaningful names for types already defined by C or for types that we have declared. Typedef names allow us to encapsulate implementation details that may change.

A typedef declaration is interpreted in the same way as a variable or function declaration, but the identifier, instead of assuming the type specified by the declaration, becomes a synonym for the type. For example:

```
typedef unsigned long int ulong;
```

The new type (***ulong***) becomes known to the compiler and is treated the same as ***unsigned long int.*** If we want to declare some more variables of type unsigned long int, wwe can use the newly defined type as :

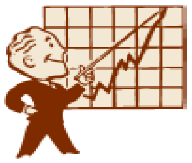
```
ulong distance;
```

We can use the typedef keyword to define a structure as foln be lows:

```
typedef struct
{
   type member1;
   type member2;
   ....
}type_name;
```

type_name can be used to declare structure variables as follows:

```
type_name  variable1,variable2,...;
```



CHECK YOUR PROGRESS

6. State whether the following statements are true(T) or false(F).

(i) Union contains members of different data types which share the same storage area in memory.

(ii) The $\longrightarrow$ operator can be combined with the period operator to access a member of a structure that is itself a member of another structure.

(iii) The keyword typedef is used to define a new data type.

(iv) A structure is a collection of data items under one name in which the item shares the same storage.

7. Choose the correct answer for the declaration:

```
typedef float height [100];
 height men,women;
```

(a) define men and women as 100 element floating point arrays.
(b) define men and women as floting point variables
(c) define height, men and women as floating point variables.
(d) All are illegal.

LET US SUM UP

- Structure is a data type used for packing logically related data of different types. Structure declaration includes the following elements: The keyword struct, the structure tag name, List of variable names separated by commas and the terminating semicolon.

- C permits the use of arrays as structure members.

- Unions are concept borrowed from structures and therefore they follow the same syntax.

- In structure, each member has its own storage location, where as all members of a union use the same location.

- C allows us to define our own special data types, called enumerated data types.

- The keyword *typedef* is used to define a new data type of our own choice. We can use the typedef keyword to define a structure.