Structures and Unions

7.1 INTRODUCTION

We seen that arrays can be used to represent a group of data items that belong to the same type, such as int or float. However, if we want to represent a collection of data items of different types using a single name, then we cannot use an array. Fortunately, **C** supports a constructed data type known as ***structure***, which is a method for packing data of different types. ***A structure is a convenient tool for handling a group of logically related data items.*** These fields are called structure elements or members.

7.2 Declaring A Structure

The general form of a structure declaration statement is given below:

```
struct <structure name>
{
   structure element1;
   structure element2;
   structure element3;
   ……….
   ……….
}
```

Example:

```
struct book
{
  char name[20];
  char author[15];
   int pages;
   float price;
}
```

Note that the above declaration has not declared any variables. It simply describes a format called ***template*** to represent information.

We can declare structure variables using the tag name anywhere in the program. For example, the statement

struct book book1, book2,book3;

declares **book1**, **book2**, **book3** as variables of type **struct book**.

It is also allowed to combine both the template declaration and variables declaration in one statement. The declaration

struct book

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

is valid. The use of tag name is optional. For example,

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

declares book1, book2, book3 as structure variables representing three books, but does not include a tag name for later use in declarations.

Note the following points while declaring a structure type:

(a) The closing brace in the structure type declaration must be followed by a semicolon.

(b) It is important to understand that a structure type declaration does not tell the compiler to reserve any space in memory. All a structure declaration does is, it defines the 'form' of the structure.

(c) Usually structure type declaration appears at the top of the source code file, before any variables or functions are defined.

7.3 ACCESSING STRUCTURES ELEMENTS

We can assign values to the members of a structure in a number of ways. As mentioned earlier, the members themselves are not variables. They should be linked to the 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 also known as ***'dot operator'*** or ***'period operator'***. The general form is

Structure-Variable. Structure-Member;

For example,

book1.**price**;

We can also use **scanf** to give the values through the keyboard.

```
scanf("%s",book1.name);
scanf("%d",&book1.pages);
```

are valid input statements.

Example:

Program: Defining and Assigning Values to Structure Members

```
#include<stdio.h>
#include<conio.h>
struct personal
{
    char name[20];
    int day;
    char month[10];
    int year;
    float salary;
};
void main()
{
   struct personal person;
   clrscr();
   printf("Input values\n");
   scanf("%s%d%s%d%f",person.name, &person.day, person.month,
     &person.year, &person.salary);
   printf("%s %d %s %d,%.2f\n",person.name,person.day,
     person.month,person.month,person.year,person.salary);
    getch();
}
```

Output

```
Input Values

Mahesh 15 February 1982 5000
Mahesh 15 February 1982 5000.00
```

7.4 ARRAYS OF STRUCTURES

We use structures to describe the format of a number of related variables. For example, in analyzing the marks obtained by a class of students, we may use a template to describe student name and marks obtained in various subjects and then declare all the students as structure variables. In such cases, we may declare an array of structures, each element of the array representing a structure variable. For example,

struct class student[100];

defines an array called **student**, that consists of 100 elements. Each element is defined to be of the type **struct class**. Consider the following declaration:

```
struct marks
{
   int sub1;
   int sub2;
   int sub3;
}s[5];
```

Program: Usage of an array of structures

```
#include<stdio.h>
#include<conio.h>
struct book
{
    char name[15];
    float price;
    int pages;
}b[3];
void main()
{
   int i;
   clrscr();
   printf("Enter name, price and pages(3 records):");
   for(i=0;i<=2;i++)
   scanf("%s %f %d",b[i].name,&b[i].price,&b[i].pages);
   printf("\nThe details are:\n");
   printf("name\tprice\tpages");
   for(i=0;i<=2;i++)
   printf("\n%s %f %d",b[i].name,b[i].price,b[i].pages);
   getch();
}
linkfloat()
{
    float a=0,*b;
    b = &a;
    a = *b;
}
```

Now a comment about the program

What is the function **linkfloat()** doing here? If you don't define it you are bound to get the error ***"Floating Point Formats Not linked"*** with majority of **C** compilers. What causes this error to occur? When parsing our source file, if the compiler encounters a reference to the address of a float, it sets a flag to have the linker link in the floating-point emulator. A floating point emulator is used to manipulate floating point numbers in runtime library functions like scanf().

7.5 ARRAYS WITHIN STRUCTURES

C permits the use of arrays as structure members. We have already used arrays of characters inside a structure. Similarly, we can use single-or multi-dimensional arrays of type **int** or **float**. For example, the following structure declaration is valid:

```
struct marks
{
   int number;
   float sub[3];
}s[2];
```

7.6 STRUCTURES WITHIN STRUCTURES

Structures within a structure means nesting of structures. Nesting of structures is permitted in **C**. Let us consider the following structure definition:

```
struct salary
{
    char name[20];
    char dept[10];
    struct
   {
      int dearness;
      int house_rent;
      int city;
    }allowance;
}employee;
```

The salary structure contains a member named allowance which itself is a structure with three members. The members contained in the inner structure namely dearness, house_rent, and city can be referred to as

```
employee.allowance.dearness
employee.allowance.house_rent
employee.allowance.city
```

7.7 STRUCTURES AND FUNCTIONS

We know that the main philosophy of **C** language is the use of the functions. Therefore, it is natural that **C** supports the passing of structure values as arguments to functions. There are three methods by which the values of a structure can be transferred from one function to another.

The first method is to pass each member of the 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.

The second method involves passing of a copy of the entire structure to the called function. Since 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 (in the calling function). It is, therefore, necessary for the function to return the entire structure back to the calling function. All compilers may not support this method of passing the entire structure as a parameter.

The third approach employs a concept *pointers* to pass the structure as an argument. In this case, the address location of the structure is passed to the called function. The function can access indirectly the entire structure and work on it. This is similar to the way arrays are passed to functions. This method is more efficient as compared to the second one.

In this section, we discuss in detail the second method, while the third approach using pointers is discussed in the next chapter, where pointers are dealt in detail.

The general format of sending a copy of a structure to the called function is:

```
Function_Name (structure variable name)
```

The called function takes the following form:

```
data_type function_name(st_name)
struct_type st_name;
{
        ………
        ………
        return(expression);
}
```

The following points are important to note:

1. The called function must be declared for its type, appropriate to the data type it is expected to return. For example, if it is returning a copy of the entire structure, then it must be declared as **struct** with an appropriate tag name.
2. The structure variable used as the actual argument and the corresponding formal argument in the called function must be of the same **struct** type.
3. The **return** statement is necessary only when the function is returning some data. The *expression* may be any simple variable or structure variable or an expression using simple variables.
4. When a function returns a structure, it must be assigned to a structure of identical type in the calling function.
5. The called function must be declared in the calling function for its type, if it is placed after the calling function.

Example:

Program: STRUCTURE AS FUNCTION PARAMETERS

```
#include<stdio.h>
#include<conio.h>
```

```
struct stores
{
   char name[20];
   float price;
   int quantity;
};
struct stores update(struct stores,float,int);
float mul(struct stores);
void main()
{
  float p_increment, value;
  int q_increment;
  static struct stores item={“XYZ”,25.75,12};
  clrscr();
  printf(“\nInput increment values:”)     ;
  printf(“price increment and quantity increment\n”);
  scanf(“%f%d”,&p_increment,&q_increment);
  item = update(item,p_increment,q_increment);
  printf(“\nUpdate values of item”);
  printf(“\n\nName:%s”,item.name);
  printf(“\nPrice:%.2f”,item.price);
  printf(“\nQuantity:%d”,item.quantity);
  value = mul(item);
  printf(“\nValue of the item=%.2f”,value);
  getch();
}
struct stores update(struct stores product, float p, int q)
{
   product.price+=p;
   product.quantity+=q;
   return(product);
}
float mul(struct stores stock)
{
  return stock.price*stock.quantity;
}
```

7.8 UNIONS

Unions are a concept borrowed from structures and therefore follow the same syntax as structures. However, there is major distinction between them in terms of storage. In structures, each member has its own storage location, whereas all the members of a union use the same location. This implies that, although a union may contain many members of different types, it can handle only one member at a time. Like structures, a union can be declared using the keyword **union** as follows:

union item
{
  int m;
  float x;
  char c;
}**code;**

7.9 SIZE OF STRUCTURES

We normally use structures, unions, and arrays to create variables of large sizes. The actual size of these variables in terms of bytes may change from machine to machine. We may use the unary operator **sizeof** to tell us the size of a structure (or any variable). The expression

sizeof(struct *x*)

will evaluate the number of bytes required to hold all the members of the structure ***x***. If y is a simple structure variable of type **struct *x***, then the expression

sizeof(y)

would also give the same answer.