UNIT 9 STRUCTURES AND UNIONS

Structure

9.0 INTRODUCTION

We have seen so far how to store numbers, characters, strings, and even large sets of these primitives using arrays, but what if we want to store collections of different kinds of data that are somehow related. For example, a file about an employee will probably have his/her name, age, the hours of work, salary, etc. Physically, all of that is usually stored in someone's filing cabinet. In programming, if you have lots of related information, you group it together in an organized fashion. Let's say you have a group of employees, and you want to make a database! It just wouldn't do to have tons of loose variables hanging all over the place. Then we need to have a single data entity where we will be able to store all the related information together. But this can't be achieved by using the arrays alone, as in the case of arrays, we can group multiple data elements that are of the same data type, and is stored in consecutive memory locations, and is individually accessed by a subscript. That is where the user-defined datatype *Structures* come in.

Structure is commonly referred to as a user-defined data type. C's *structures* allow you to store multiple variables of any type in one place (the structure). A structure can contain any of C's data types, including arrays and other structures. Each variable within a structure is called a *member* of the structure. They can hold any number of variables, and you can make arrays of structures. This flexibility makes structures ideally useful for creating databases in C. Similar to the structure there is another user defined data type called *Union* which allows the programmer to view a single storage in more than one way i.e., a variable declared as union can store within its storage space, the data of different types, at different times. In this unit, we will be discussing the user-defined data type structures and unions.

9.1 OBJECTIVES

After going through this unit you should be able to:

- declare and initialize the members of the structures;
- access the members of the structures;
- pass the structures as function arguments;
- declare the array of structures;
- declare and define union; and
- perform all operations on the variables of type Union.

9.2 DECLARATION OF STRUCTURES

To declare a structure you must start with the keyword ***struct*** followed by the *structure name* or *structure tag* and within the braces the list of the structure's member variables. Note that the structure declaration does not actually create any variables. The syntax for the structure declaration is as follows:

```
struct structure-tag {
                    datatype variable1;
                    datatype variable2;
                    dataype variable 3;
                        ...
                    };
```

For example, consider the student database in which each student has a roll number, name and course and the marks obtained. Hence to group this data with a structure-tag as ***student***, we can have the declaration of structure as:

```
struct student {
                int roll_no;
                char name[20];
                char course[20];
                int marks_obtained ;
              };
```

The point you need to remember is that, till this time no memory is allocated to the structure. This is only the definition of structure that tells us that there exists a user-defined data type by the name of student which is composed of the following members. Using this structure type, we have to create the structure variables:

```
 struct student stud1, stud2 ;
```

At this point, we have created two instances or structure variables of the user-defined data type student. Now memory will be allocated. The amount of memory allocated will be the sum of all the data members which form part of the structure template.

The second method is as follows:

```
struct {
          int roll_no;
          char name[20];
          char course[20];
          int marks_obtained ;
       }  stud1, stud2 ;
```

In this case, a tag name *student* is missing, but still it happens to be a valid declaration of structure. In this case the two variables are allocated memory equivalent to the members of the structure.

The advantage of having a tag name is that we can declare any number of variables of the tagged named structure later in the program as per requirement.

If you have a small structure that you just want to define in the program, you can do the definition and declaration together as shown below. This will define a structure of type *struct telephone* and declare three instances of it.

Consider the example for declaring and defining a structure for the telephone billing with three instances:

```
struct telephone{
                    int tele_no;
                    int cust_code;
```

```
                char cust_address[40];
                int bill_amt;
        }       tele1, tele2, tele3;
```

The structure can also be declared by using the typedefinition or typedef. This can be done as shown below:

```
typedef struct country{
                char name[20];
                int population;
                char language[10];
              } Country;
```

This defines a structure which can be referred to either as *struct country* or *Country*, whichever you prefer. Strictly speaking, you don't need a tag name both before and after the braces if you are not going to use one or the other. But it is a standard practice to put them both in and to give them the same name, but the one after the braces starts with an uppercase letter.

The *typedef* statement doesn't occupy storage: it simply defines a new type. Variables that are declared with the *typedef* above will be of type *struct country,* just like population is of type integer. The structure variables can be now defined as below:

Country Mexico, Canada, Brazil;

9.3 ACCESSING THE MEMBERS OF A STRUCTURE

Individual structure members can be used like other variables of the same type. Structure members are accessed using *the structure member operator* (.), also called the *dot operator*, between the structure name and the member name. The syntax for accessing the member of the structure is:

structurevariable. member-name;

Let us take the example of the coordinate structure.

```
struct coordinate{
                int x;
                int y;
              };
```

Thus, to have the structure named first refer to a screen location that has coordinates x=50, y=100, you could write as,

```
first.x = 50;
first.y = 100;
```

To display the screen locations stored in the structure second, you could write,

```
printf ("%d,%d", second.x, second.y);
```

The individual members of the structure behave like ordinary date elements and can be accessed accordingly.

Now let us see the following program to clarify our concepts. For example, let us see, how will we go about storing and retrieving values of the individual data members of the student structure.

Example 9.1

```
/*Program to store and retrieve the values from the student structure*/
```

```
#include<stdio.h>
struct student {
                    int roll_no;
                    char name[20];
                    char course[20];
                    int marks_obtained ;
              };
main()
{
student s1 ;
printf (“Enter the  student roll number:”);
scanf (“%d”,&s1.roll_no);
printf (“\nEnter the student name: “);
scanf (“%s”,s1.name);
printf (“\nEnter the student  course”);
scanf (“%s”,s1.course);
printf (“Enter the student percentage\n”);
scanf (“%d”,&s1.marks_obtained);
printf (“\nData entry is complete”);
printf ( “\nThe data entered is as follows:\n”);
printf (“\nThe student roll no is %d”,s1.roll_no);
printf (“\nThe student name is %s”,s1.name);
printf (“\nThe student course is %s”,s1.course);
printf (“\nThe student percentage is %d”,s1.marks_obtained);
}
```

OUTPUT

```
Enter the student roll number: 1234
Enter the student name: ARUN
Enter the student course: MCA
Enter the student percentage: 84
Date entry is complete

The data entered is as follows:
The student roll no is 1234
The student name is ARUN
The student course is MCA
The student percentage is 84
```

Another way of accessing the storing the values in the members of a structure is by initializing them to some values at the time when we create an instance of the data type.

9.4 INITIALIZING STRUCTURES

Like other C variable types, structures can be initialized when they’re declared. This procedure is similar to that for initializing arrays. The structure declaration is followed by an equal sign and a list of initialization values is separated by commas and enclosed in braces. For example, look at the following statements for initializing the values of the members of the *mysale* structure variable.

Example 9.2

```
struct sale {
              char customer[20];
             char item[20];
              float amt;
               } mysale = { "XYZ Industries",
```

```
                    "toolskit",
                    600.00
                  };
```

In a structure that contains structures as members, list the initialization values in order. They are placed in the structure members in the order in which the members are listed in the structure definition. Here's an example that expands on the previous one:

Example 9.3

```
struct customer {
                char firm[20];
                char contact[25];
                }

struct sale {
           struct customer buyer1;
           char item [20];
           float amt;
           } mysale = {
                       { "XYZ Industries", "Tyran Adams"},
                        "toolskit",
                        600.00
                       };
```

These statements perform the following initializations:

- the structure member *mysale.buyer1.firm* is initialized to the string "XYZ Industries".
- the structure member *mysale.buyer1.contact* is initialized to the string "Tyran Adams".
- the structure member *mysale.item* is initialized to the string "toolskit".
- the structure member *mysale.amount* is initialized to the amount 600.00.

For example let us consider the following program where the data members are initialized to some value.

Example 9.4

Write a program to access the values of the structure initialized with some initial values.

```
/* Program to illustrate to access the values of the structure initialized with some
initial values*/

#include<stdio.h>
struct telephone{
        int tele_no;
        int cust_code;
        char cust_name[20];
        char cust_address[40];
        int bill_amt;
        };
main()
{
 struct telephone tele = {2314345,
                          5463,
                          "Ram",
                          "New Delhi",
```

```
2435    };

printf("The values are initialized in this program.");
printf("\nThe telephone number is  %d",tele.tele_no);
printf("\nThe customer code is  %d",tele.cust_code);
printf("\nThe customer name is  %s",tele.cust_name);
printf("\nThe customer address is  %s",tele.cust_address);
printf("\nThe bill amount is   %d",tele.bill_amt);
}
```

OUTPUT

```
The values are initialized in this program.
The telephone number is 2314345
The customer code is 5463
The customer name is Ram
The customer Address is New Delhi
The bill amount is 2435
```

Check Your Progress 1

1. What is the difference between the following two declarations?

```
struct x1{…….};
typedef struct{……}x2;
```

……

……

2. Why can't you compare structures?

……

……..

3. Why does size of report a larger size than, one expects, for a structure type, as if there were padding at the end?

……

……

4. Declare a structure and instance together to display the date.

……

……

9.5 STRUCTURES AS FUNCTION ARGUMENTS

C is a structured programming language and the basic concept in it is the modularity of the programs. This concept is supported by the functions in C language. Let us look into the techniques of passing the structures to the functions. This can be achieved in primarily two ways: Firstly, to pass them as simple parameter values by passing the structure name and secondly, through pointers. We will be concentrating on the first method in this unit and passing using pointers will be taken up in the next unit. Like other data types, a structure can be passed as an argument to a function. The program listing given below shows how to do this. It uses a function to display data on the screen.

Example 9.5

Write a program to demonstrate passing a structure to a function.

```
/*Program to demonstrate passing a structure to a function.*/

#include <stdio.h>

/*Declare and define a structure to hold the data.*/

struct data{
        float amt;
        char fname [30];
        char lname [30];
        } per;

main()
{
void print_per (struct data x);
printf("Enter the donor's first and last names separated by a space:");
scanf ("%s %s", per.fname, per.lname);
printf ("\nEnter the amount donated in rupees:");
scanf ("%f", &per.amt);
print_per (per);
return 0;
 }

void print_per(struct data x)
{
  printf ("\n %s %s gave donation of amount Rs.%.2f.\n", x.fname, x.lname, x.amt);
}
```

OUTPUT

```
Enter the donor's first and last names separated by a space: RAVI KANT
Enter the amount donated in rupees: 1000.00
RAVI KANT gave donation of the amount Rs. 1000.00.
```

You can also pass a structure to a function by passing the structure's address (that is, a pointer to the structure which we will be discussing in the next unit). In fact, in the older versions of C, this was the only way to pass a structure as an argument. It is not necessary now, but you might see the older programs that still use this method. If you pass a pointer to a structure as an argument, remember that you must use the indirect membership operator (→) to access structure members in the function.

Please note the following points with respect to passing the structure as a parameter to a function.

- The return value of the called function must be declared as the value that is being returned from the function. If the function is returning the entire structure then the return value should be declared as *struct* with appropriate tag name.
- The actual and formal parameters for the structure data type must be the same as the *struct* type.
- The return statement is required only when the function is returning some data.
- When the return values of type is *struct*, then it must be assigned to the structure of identical type in the calling function.

Let us consider another example as shown in the Example 9.6, where *structure salary* has three fields related to an employee, namely - *name, no_days_worked* and d*aily_wage*. To accept the values from the user we first call the function *get_data* that

gets the values of the members of the structure. Then using the *wages* function we calculate the salary of the person and display it to the user.

Example 9.6

Write a program to accept the data from the user and calculate the salary of the person using concept of functions.

```
/* Program to accept the data from the user and calculate the salary of the person*/

#include<stdio.h>
main()
{
  struct sal     {
          char name[30];
          int no_days_worked;
          int daily_wage;       };
          struct sal salary;
          struct sal get_dat(struct);          /* function prototype*/
          float  wages(struct);                /*function prototype*/
          float amount_payable;                /* variable declaration*/
          salary = get_data(salary);
          printf("The name of employee is %s",salary.name);
          printf("Number of days worked is %d",salary.no_daya_worked);
          printf("The daily wage of the employees is %d",salary.daily_wage);
         amount_payable = wages(salary);
          printf("The amount payable to %s is %f",salary.name,amount_payable);
}

struct sal get_data(struct sal income)
          {
          printf("Please enter the employee name:\n");
          scanf("%s",income.name);
          printf("Please enter the number of days worked:\n");
          scanf("%d",&income.no_days_worked);
          printf('Please enter the employee daily wages:\n");
          scanf("%d",&income.daily_wages);
         return(income);
        }

float wages(struct)
{
   struct sal amt;
   int total_salary ;
   total_salary = amt.no_days_worked * amt.daily_wages;
   return(total_salary);     }
```

Check Your Progress 2

1. How is structure passing and returning implemented?

……

……

……

2. How can I pass constant values to functions which accept structure arguments?

……

……

……

3. What will be the output of the program?

```
#include<stdio.h>
main( )
{
struct pqr{
      int x ;
     };
struct pqr  pqr ;
pqr.x =10 ;
printf ("%d", pqr.x);
}
```

……

……

……

9.6 STRUCTURES AND ARRAYS

Thus far we have studied as to how the data of heterogeneous nature can be grouped together and be referenced as a single unit of structure. Now we come to the next step in our real world problem. Let's consider the example of students and their marks. In this case, to avoid declaring various data variables, we grouped together all the data concerning the student's marks as one unit and call it student. The problem that arises now is that the data related to students is not going to be of a single student only. We will be required to store data for a number of students. To solve this situation one way is to declare a structure and then create sufficient number of variables of that structure type. But it gets very cumbersome to manage such a large number of data variables, so a better option is to declare an array.

So, revising the array for a few moments we would refresh the fact that an array is simply a collection of homogeneous data types. Hence, if we make a declaration as:

int temp[20];

It simply means that temp is an array of twenty elements where each element is of type integer, indicating homogenous data type. Now in the same manner, to extend the concept a bit further to the structure variables, we would say,

struct student stud[20] ;

It means that *stud* is an array of twenty elements where each element is of the type *struct student* (which is a user-defined data type we had defined earlier). The various members of the *stud* array can be accessed in the similar manner as that of any other ordinary array.

For example,
struct student stud[20], we can access the *roll_no* of this array as

stud[0].roll_no;
stud[1].roll_no;
stud[2].roll_no;
stud[3].roll_no;

```
…
…
…
stud[19].roll_no;
```

Please remember the fact that for an array of twenty elements the subscripts of the array will be ranging from 0 to 19 (a total of twenty elements). So let us now start by seeing how we will write a simple program using array of structures.

Example 9.7

Write a program to read and display data for 20 students.

```
/*Program to read and print the data for 20 students*/

#include <stdio.h>
struct student {  int roll_no;
                  char name[20];
                  char course[20];
                  int marks_obtained ;
                 };
main( )
{
 struct student stud [20];
 int i;
 printf (“Enter the student data one by one\n”);
 for(i=0; i<=19; i++)
  {
     printf (“Enter the roll number of %d student”,i+1);
     scanf (“%d”,&stud[i].roll_no);
     printf (“Enter the name of %d student”,i+1);
     scanf (“%s”,stud[i].name);
     printf (“Enter the course of %d student”,i+1);
     scanf (“%d”,stud[i].course);
     printf (“Enter the marks obtained of %d student”,i+1);
     scanf (“%d”,&stud[i].marks_obtained);
   }
     printf (“the data entered is as follows\n”);
     for (i=0;i<=19;i++)
  {
     printf (“The roll number of  %d student is %d\n”,i+1,stud[i].roll_no);
     printf (“The name of  %d student is %s\n”,i+1,stud[i].name);
     printf (“The course of  %d student is %s\n”,i+1,stud[i].course);
     printf (“The marks of  %d student is %d\n”,i+1,stud[i].marks_obtained);
   }
}
```

The above program explains to us clearly that the array of structure behaves as any other normal array of any data type. Just by making use of the subscript we can access all the elements of the structure individually.

Extending the above concept where we can have arrays as the members of the structure. For example, let’s see the above example where we have taken a structure for the student record. Hence in this case it is a real world requirement that each student will be having marks of more than one subject. Hence one way to declare the structure, if we consider that each student has 3 subjects, will be as follows:

```
struct student {
                  int roll_no;
                  char name [20];
```

```
                char course [20];
                int subject1 ;
                int subject2;
                int subject3;
              };
```

The above described method is rather a bit cumbersome, so to make it more efficient we can have an array inside the structure, that is, we have an array as the member of the structure.

```
struct student {
                int roll_no;
                char name [20];
                char course [20];
                int subject [3] ;
              };
```

Hence to access the various elements of this array we can the program logic as follows:

Example 9.8

```
/*Program to read and print data related to five students having marks of three
subjects each using the concept of arrays */

#include<stdio.h>
struct student {
                int roll_no;
                char name [20];
                char course [20];
                int subject [3] ;
              };
main( )
{
  struct student stud[5];
  int i,j;
printf (“Enter the data for all the students:\n”);
for (i=0;i<=4;i++)
{
printf (“Enter the roll number of %d student”,i+1);
scanf (“%d”,&stud[i].roll_no);
printf(“Enter the name of %d student”,i+1);
scanf (“%s”,stud[i].name);
printf (“Enter the course of %d student”,i+1);
scanf (“%s”,stud[i].course);
for (j=0;j<=2;j++)
  {
   printf (“Enter the marks of the %d subject of the student %d:\n”,j+1,i+1);
  scanf (“%d”,&stud[i].subject[j]);
  }
}
printf (“The data you have entered is as follows:\n”);
for (i=0;i<=4;i++)
   {
        printf (“The %d th student's roll number is %d\n”,i+1,stud[i].roll_no);
        printf (“The %d the student's name is %s\n”,i+1,stud[i].name);
        printf (“The %d the student's course is %s\n”,i+1,stud[i].course);
        for (j=0;j<=2;j++)
        {
        printf (“The %d the student's marks of %d   I  subject are %d\n”,i+1, j+1,
        stud[i].subject[j]);
```

```
        }
    }
  printf ("End of the program\n");
}
```

Hence as described in the example above, the array as well as the arrays of structures can be used with efficiency to resolve the major hurdles faced in the real world programming environment.

9.7 UNIONS

Structures are a way of grouping homogeneous data together. But it often happens that at any time we require only one of the member's data. For example, in case of the support price of shares you require only the latest quotations. And only the ones that have changed need to be stored. So if we declare a structure for all the scripts, it will only lead to crowding of the memory space. Hence it is beneficial if we allocate space to only one of the members. This is achieved with the concepts of the *UNIONS*. *UNIONS* are similar to *STRUCTURES* in all respects but differ in the concept of storage space.

A *UNION* is declared and used in the same way as the structures. Yet another difference is that only one of its members can be used at any given time. Since all members of a Union occupy the same memory and storage space, the space allocated is equal to the largest data member of the Union. Hence, the member which has been updated last is available at any given time.

For example a union can be declared using the syntax shown below:

```
union union-tag {
                    datatype variable1;
                    datatype variable2;
                        ...
                };
```

For example,

```
union temp{
            int   x;
            char y;
            float z;
           };
```

In this case a float is the member which requires the largest space to store its value hence the space required for float (4 bytes) is allocated to the union. All members share the same space. Let us see how to access the members of the union.

Example 9.9

Write a program to illustrate the concept of union.

```
/* Declare a union template called tag */
union tag {
            int nbr;
            char character;
            }
/* Use the union template */
union tag mixed_variable;
/* Declare a union and instance together */
union generic_type_tag {
                        char c;
                        int i;
```

```
        float f;
        double d;
        } generic;
```

9.8 INITIALIZING AN UNION

Let us see, how to initialize a Union with the help of the following example:

Example 9.10

```
union date_tag {
        char complete_date [9];
        struct part_date_tag {
                char month[2];
                char break_value1;
                char day[2];
                char break_value2;
                char year[2];
                        } parrt_date;
        }date = {"01/01/05"};
```

9.9 ACCESSING THE MEMBERS OF AN UNION

Individual union members can be used in the same way as the structure members, by using the member operator or dot operator (.). However, there is an important difference in accessing the union members. Only one union member should be accessed at a time. Because a union stores its members on top of each other, it's important to access only one member at a time. Trying to access the previously stored values will result in erroneous output.

Check Your Progress 3

1. What will be the output?

```
#include<stdio.h>
main()
{
union{
        struct{
                char x;
                char y;
                char z;
                char w;
               }xyz;

       struct{
              int p;
              int q ;
              }pq;
              long  a ;
              float b;
              double d;
              }prq;
       printf ("%d",sizeof(prq));
              }
```

9.10 SUMMARY

In this unit, we have learnt how to use structures, a data type that you design to meet the needs of a program. A structure can contain any of C's data types, including other structures, pointers, and arrays. Each data item within a structure, called a *member*, is accessed using the structure member operator (.) between the structure name and the member name. Structures can be used individually, and can also be used in arrays.

Unions were presented as being similar to structures. The main difference between a union and a structure is that the union stores all its members in the same area. This means that only one member of a union can be used at a time.

9.11 SOLUTIONS / ANSWERS

Check Your Progress 1

1. The first form declares a *structure tag*; the second declares a *typedef*. The main difference is that the second declaration is of a slightly more abstract type - users do not necessarily know that it is a structure, and the keyword struct is not used while declaring an instance.

2. There is no single correct way for a compiler to implement a structure comparison consistent with C's low-level flavor. A simple byte-by-byte comparison could detect the random bits present in the unused "holes" in the structure (such padding is used to keep the alignment of later fields correct). A field-by-field comparison for a large structure might require an inordinate repetitive code.

3. Structures may have this padding (as well as internal padding), to ensure that alignment properties will be preserved when an array of contiguous structures is allocated. Even when the structure is not part of an array, the end padding remains, so that *sizeof* can always return a consistent size.

4. struct date {
```
        char month[2];
        char day[2];
        char year[4];
} current_date;
```

Check Your Progress 2

1. When structures are passed as arguments to functions, the entire structure is typically pushed on the stack, using as many words. (Programmers often choose to use pointers instead, to avoid this overhead). Some compilers merely pass a pointer to the structure, though they may have to make a local copy to preserve pass-by value semantics.

   Structures are often returned from functions in a pointed location by an extra, compiler-supplied "hidden" argument to the function. Some older compilers used a special, static location for structure returns, although this made structure - valued functions non-reentrant, which ANSI C disallows.

2. C has no way of generating anonymous structure values. You will have to use a temporary structure variable or a little structure - building function.

3. 10

Check Your Progress 3

1. 8

9.12 FURTHER READINGS

1. The C Programming Language, *Kernighan & Richie*, PHI Publication, 2002.
2. Computer Science A structured programming approach using C, *Behrouza .Forouzan, Richard F. Gilberg*, Second Edition, Brooks/Cole, Thomson Learning, 2001.
3. Programming with C, Schaum Outlines, Second Edition, *Gottfried*, Tata McGraw Hill, 2003.