9.1 Introduction

AU : June-11, Dec.-16

So far, we have seen one kind of compound (user defined) data type - the array in chapter 6 and we have seen how we can group information into one common data structure. However, the use of arrays is limited to cases where all of the information to be grouped together is of the same type. In this chapter we present the other compound data type provided in C - **the structure**, which removes the above limitation. We will discuss structures, pointers to structures, and arrays of structures. As with our previous data types, we will see how such structures can be declared; how information in them can be accessed, and how we can pass and return structures in functions. We will also see how arrays of structures are sorted and searched. We illustrate these points with several example programs.

Finally, we will introduce unions which are similar to structures; however, the elements in the union share the same memory cells. In a union, different types of data may be stored in a variable but at different times.

Review Question

1. *What is a structure ?*

AU : June-11, Dec.-16, Marks 2

9.2 Defining a Structure

AU : Jan.-09, 13, 14, Dec.-14, 15

In C, a **structure** is a derived data type consisting of a collection of member elements and their data types. Thus, a variable of a structure type is the name of a group of one or more **members** which may or may not be of the same data type. In programming terminology, a structure data type is referred to as a **record data type** and the members are called **fields.**

For defining the structures, we must specify the names and types of each of the fields of the structure and declare variables of that type. Here, is an example structure definition.

```
struct student
        {
                char name[64];
                char course[128];
                int age;
                int year;
        };
```

This defines a new data type student. It contains four members :

- 64 - element character array, **name [64].**
- 128 - element character array, **course [128].**
- Integer quantity, **age.**
- Integer quantity, **year.**

The composition of this student structure is illustrated in Fig. 9.2.1.

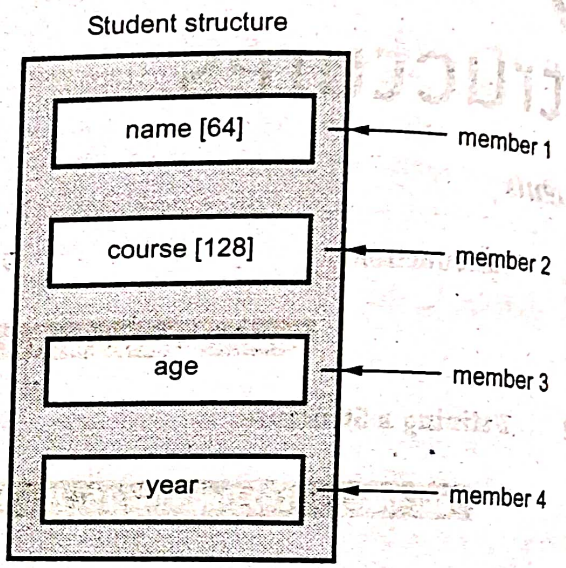

Fig. 9.2.1 Structure student

In general terms, the composition of structure may be defined as

```
struct tag
        {
                member 1;
                member 2;
                    :
                    :
                member n;
        };
```

where struct is a keyword.

tag is a name that identifies structures of this type.

The left brace indicates the beginning of a structure and right brace indicates the end of the structure. The members of the structure are specified within the pair of curly braces separated by semicolons. The structure template is terminated by semicolon. Each member of structure may belong to a different type of data. The

individual members can be ordinary variables, arrays, pointers or other structures.

The variables of type student can be declared as follows.

```
struct student s1, s2 ,s3;
```

The s1, s2, s3 are variables of data type student. They have members(fields) called name, course, age and year. In general terms, individual structure-type variables can be declared as :

```
struct tag variable 1, variable 2, variable 3,
                                    . . . ,variable n;
```

Declaring structure variable includes :

- The keyword struct
- The structure tag name
- List of variables separated by commas
- A terminating semicolon

The structure variable declaration statement sets aside the space in memory. It holds the space for all the members in the structure and all the declared variables of the structure. In case of student structure, each variable reserves 64 bytes for name, 128 bytes for course, four bytes for age and four bytes for year.

We can combine the definition of structure and declaration of structure variables in one statement. For example,

```
struct student
                    {
                    char name[64];
                    char course[128];
                    int age;
                    int year;
                    } s1, s2, s3;
```

In such statements structure tag may be omitted.

9.2.1 Initialization of a Structure

Like initialization of array elements, structure members can be initialized within the variable declaration part. In such initialization, the values of members must appear in the order in which they are defined in the structure. The initialization may include the storage class it belongs to at the beginning of the declaration. For example,

```
struct student s1 = { "Ruturaj", "Information
                                technology", 18, 2006};
static struct student s2 = { "Neha", "Computer
                                science", 20, 2006};
```

The above initializations assign values to variables s1 and s2 as shown in the Table 9.2.1.

structure : student	s1	s2
name	Ruturaj	Neha
course	Information technology	Computer science
age	18	20
year	2006	2006

Table 9.2.1

We can define an array of structures i.e., an array in which each element is a structure. For example,

```
struct student s[100];
```

In this declaration s is a 100-element array of structures. Hence, each element of s is a separate structure of type student (i.e. each element of student represents an individual student record.)

An array of structures can be assigned initial values just as any other array. Remember that each array element is a structure that must be assigned a corresponding set of initial values, as illustrated in the following example.

➤ **Example 9.2.1 : Initialization of array of structure.**

```
struct student
```

```
{       char name [64];
        char course[128];

        int age;
        int year;
};
static struct student s [ ]=
                                {
                                   "Neha", "Computer science", 20, 2006,
                                   "Ruturaj", "Information technology", 18, 2006,
                                   "Gururaj", "Computer science", 22, 2004,
                                   "Yash", "Electronics", 21, 2005,
                                };
```

The two different structures can have same member name. It can be used to represent different data. This states that the scope of a member is confined to the particular structure within which it is defined.

➤ **Example 9.2.2 : Scope of the member.**

```
struct one
 {
    int a,
    float b,
    char c,
 };
struct two
 {
    int c,
    char b,
    float a,
 };
```

Comment : We can observed that the individual member names a,b and c appear in both structure definitions, but the associated data types are different.

9.2.2 Accessing and Processing a Structure

The members of a structure can be accessed and processed as separate entities. A structure members can be accessed by using dot (.), also called **period** operator. The syntax is,

```
variable . member
```

where **variable** refers to the name of a structure type variable, and **member** refers to the name of a member within the structure.

In our example, we can access name, course, age and year of structure variable s1 by writing.

s1.name
s1.course
s1.age
s1.year

Example 9.2.3 : Simple program to demonstrate the accessing of structure.

AU : Jan.-14, Dec.-15, Marks 8, Dec.-14, Marks 10

```
/* The program creates a variable of type date, assign values to all of its fields and print out the result.
*/
#include <stdio.h>
#include <string.h>        /* We need the system-provided strcpy( ) */
void main( )
{
struct date                /* declare the shape of date */
{
int day;
int year;
char month_name[10];
};
struct date d;             /* create a variable d of type date */
d.day= 25;
/*   d.month_name = "January" ; This is not valid statement */
strcpy(d.month_name,"January");
d.year= 2006;
printf("Day= %d Month= %s Year= %d\n", d.day,d.month_name,d.year);
}
```

Output

Day= 25 Month=January Year= 2006

Example 9.2.4 : This program stores data in a structure and displays the values.

```
#include <stdio.h>
void main(void)
{
 struct demo
 {
    int i;
    int j;
 } s;
 int i;
 i=10;
 s.i=100;
 s.j=101;
 printf("%d %d %d", i, s.i, s.j);
```

It is important to note that student array contains two more arrays : name and course. However, it is not a multidimensional array. To be a multidimensional array, each level must have the same data type. In this case, data types are different : the studentA is student, while name and course is character.

To access the data for one student, we need to refer only to the structure name with an index as shown below.

```
studentA[i]
```

However, to access an individual element in one of the student's arrays, such as the age for the fourth student, we need to specify the field name as shown below.

```
student[3].age
```

To access specific letter in the character array of student structure, we have to specify the field with an index. To access the first letter of the name of the fifth student we write,

```
studentA[4].name[0]
```

9.4 Structures and Functions

9.4.1 Passing Structure Members to Function

We can pass a member of a structure to a function by

```
func (var1.
func1 (var1

func2 (var1
func3 (var1
```

In each cas
is passed t
element is
pass the
structure.
before the
address o
have to w

```
func

func1

func2

func3
```

It is impo
the struct
Also note
and is req

It is important to note that student array contains two more arrays : name and course. However, it is not a multidimensional array. To be a multidimensional array, each level must have the same data type. In this case, data types are different : the studentA is student, while name and course is character.

To access the data for one student, we need to refer only to the structure name with an index as shown below.

```
studentA[i]
```

However, to access an individual element in one of the student's arrays, such as the age for the fourth student, we need to specify the field name as shown below.

```
student[3].age
```

To access specific letter in the character array of student structure, we have to specify the field with an index. To access the first letter of the name of the fifth student we write,

```
studentA[4].name[0]
```

9.8 Self-referential Structures

A structure may contain a pointer to its own type:

```
struct cell
{
int data;
struct cell *link;
}
```

Such structures are known as self-referential structures. Here, **link** refers to the name of the pointer variable. Thus the structure of type **call** contains a member that points to another structure of type **cell.**

Self-referential structures are very useful in applications that involve linked data structures.