Chapter 10

Structures and Unions

10.1 INTRODUCTION

We have 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, we cannot use an array if we want to represent a collection of data items of different types using a single name. Fortunately, C supports a constructed data type known as *structures,* a mechanism for packing data of different types. A structure is a convenient tool for handling a group of logically related data items. For example, it can be used to represent a set of attributes, such as student_name, roll_number and marks. The concept of a structure is analogous to that of a 'record' in many other languages. More examples of such structures are:

time	:	seconds, minutes, hours
date	:	day, month, year
book	:	author, title, price, year
city	:	name, country, population
address	:	name, door-number, street, city
inventory	:	item, stock, value
customer	:	name, telephone, city, category

Structures help to organize complex data in a more meaningful way. It is a powerful concept that we may often need to use in our program design. This chapter is devoted to the study of structures and their applications in program development. Another related concept known as *unions* is also discussed.

10.2 DEFINING A STRUCTURE

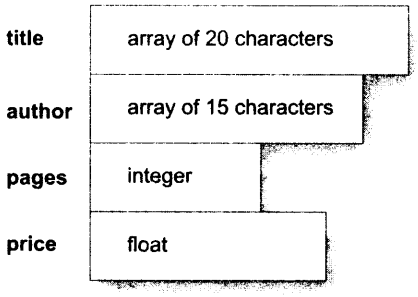
Unlike arrays, structures must be defined first for their format that may be used later to declare structure variables. Let us use an example to illustrate the process of structure definition and the

creation of structure variables. Consider a book database consisting of book name, author, number of pages, and price. We can define a structure to hold this information as follows:

```
struct book_bank
{
    char   title[20];
    char   author[15];
    int    pages;
    float  price;
};
```

The keyword **struct** declares a structure to hold the details of four data fields, namely **title, author, pages**, and **price**. These fields are called *structure elements* or *members*. Each member may belong to a different type of data. **book_bank** is the name of the structure and is called the *structure tag*. The tag name may be used subsequently to declare variables that have the tag's structure.

Note that the above definition has not declared any variables. It simply describes a format called *template* to represent information as shown below:



The general format of a structure definition is as follows:

```
struct   tag_name
{
     data_type     member1;
     data_type     member2;
       ----        ----
       ----        ----
};
```

In defining a structure you may note the following syntax:

1. The template is terminated with a semicolon.
2. While the entire definition is considered as a statement, each member is declared independently for its name and type in a separate statement inside the template.
3. The tag name such as **book_bank** can be used to declare structure variables of its type, later in the program.

Arrays Vs Structures

Both the arrays and structures are classified as structured data types as they provide a mechanism that enable us to access and manipulate data in a relatively easy manner. But they differ in a number of ways.

1. An array is a collection of related data elements of same type. Structure can have elements of different types.
2. An array is derived data type whereas a structure is a programmer-defined one.
3. Any array behaves like a built-in data type. All we have to do is to declare an array variable and use it. But in the case of a structure, first we have to design and declare a data structure before the variables of that type are declared and used.

10.3 DECLARING STRUCTURE VARIABLES

After defining a structure format we can declare variables of that type. A structure variable declaration is similar to the declaration of variables of any other data types. It includes the following elements.

1. The keyword **struct**
2. The structure tag name
3. List of variable names separated by commas
4. A terminating semicolon

For example, the statement

```
struct book_bank, book1, book2, book3;
```

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

Each one of these variables has four members as specified by the template. The complete declaration might look like this:

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

Remember that the members of a structure themselves are not variables. They do not occupy any memory until they are associated with the structure variables such as **book1.** When the compiler comes across a declaration statement, it reserves memory space for the structure variables. It is also allowed to combine both the structure definition and variables declaration in one statement.

The declaration

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

is valid. The use of tag name is optional here. For example:

```
struct
{ ........
  ........
  ........
} book1, book2, book3;
```

declares **book1, book2,** and **book3** as structure variables representing three books, but does not include a tag name. However, this approach is not recommended for two reasons.

1. Without a tag name, we cannot use it for future declarations.
2. Normally, structure definitions appear at the beginning of the program file, before any variables or functions are defined. They may also appear before the **main,** along with macro definitions, such as **#define.** In such cases, the definition is *global* and can be used by other functions as well.

Type-Defined Structures

We can use the keyword **typedef** to define a structure as follows:

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

The type_name represents structure definition associated with it and therefore can be used to declare structure variables as shown below:

type_name variable1, variable2, ... ;

Remember that (1) the name *type_name* is the type definition name, not a variable and (2) we cannot define a variable with *typedef* declaration.

10.4 ACCESSING STRUCTURE MEMBERS

We can access and 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