Chapter 13: Structures and unions

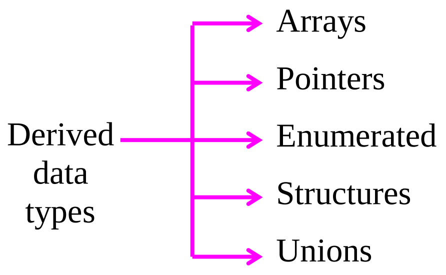
What are we studying in this chapter?

- **The type definition**
- **Structure, Accessing structures**
- **Complex structures**
- **Array of structures**
- **Structures and functions**
- **Unions, Bit-fields** **- 5 hours**

13.1 Introduction

The various primitive data types that are available in C language are **int**, **float**, **char**, **double** and **void**. Using these primitive data types, we can derive some other data types. In this chapter let us discuss about the need for the various derived types such as *structures* and *unions*. First we shall see *"What is a derived data type?"*

Definition: The data types that are derived from the basic data types such as **int**, **float**, **char**, **double** are called *derived data types.* The various derived data types are shown below:



Arrays, pointers and enumerated data types have already been discussed in earlier chapters. In this chapter we discuss *structures* and *unions*.

13.2 The type definition

Now, let us see *"What is type definition?"*

Definition: The **typedef** is a keyword that allows the programmer to create a new data type name for an already existing data type. The purpose of **typdef** is to give alternate meaningful names for any data type. So, the process of creating a new datatype name for an any data type is called **type definition.** The general syntax of the **typedef** is shown below:

Syntax:

```
        typedef     data_type   new_name1, new_name2,….new_name;
           ↓            ↓              ↓
ex 1:   typedef        int         MARKS, ID_NUMBER;
ex 2:   typedef       float        AMOUNT;
             ↓           ↓              ↓
        keyword     existing       user-defined data types
                    data type
```

Points to remember:

- We are not creating new data types using **typedef**. Instead, we are creating only new alternate names for the existing data types.
- MARKS and ID_NUMBER are alternate names given to the data type **int**
- AMOUNT is alternate name given to the data type **float.**
- So, the new data type names MARKS, ID_NUMBER and AMOUNT are called *user-defined data types.*

Now, let us see *"What are user-defined data types?"*

Definition: The new data type names that are defined by the user using already existing data types with the help of keyword **typedef** are called user-defined data types. In the above example, MARKS, ID_NUMBER and AMOUNT are user defined data types.

Example 13.1: Suppose, we want to store marks scored in various subjects in variables **sub1**, **sub2**, **sub3** and the identification numbers in the variables **s1**, **s2** and **s3**. These variables can be declared as shown below:

```
int     sub1, sub2, sub3;
int     s1, s2, s3;
```

Using the user-defined data types, the variables can be declared as shown below:

```
typedef      int      MARKS, ID_NUMBER;     // user-defined data types
MARKS                 sub1, subj2, subj3;
ID_NUMBER             s1, s2, s3;
```

Note: Normally, the user defined data types will be in uppercase. This alerts the programmer that there is something unusual about the type.

Now, let us see *"What are the advantages of **typedef**?"* The advantages of **typedef**s are shown below:

- Provides alternate meaningful names for any data type. [Example 13.1]
- Provides a meaningful way of declaring the variables [Example 13.1]
- Increases the readability of the program. [Example 13.1]

- A complex and lengthy declaration can be reduced to short and meaningful declaration [look at section 13.4.4]
- Helps us to understand the source code and also save time and energy spent in understanding the complex programs.

Now, let us see *"Where the type definitions are placed?"* The **typedef** definitions are normally placed:

- prior to the prototypes of any functions or function definitions i.e., immediately after #include's
- At the beginning of a function prior to all other declarations

Consider the program to find simple interest using various **typdef** definitions.

Example 13.4: Program to compute simple interest using **typedef** definitions

```
#include <stdio.h>

typedef    float   AMOUNT;          // various typedef definitions
typedef    float   TIME;
typedef    float   RATE;

void main()
{
        AMOUNT     p, si;           // principle amount and simple interest
        TIME       t;               // time in years
        RATE       r;               // rate of interest

        printf("Enter p, t and r\n");
        scanf("%f %f %f", &p, &t, &r);

        si = p * t * r / 100;       // compute simple interest

        printf("SI = %f\n", si);
}
```

Observe the following points:

- AMOUNT is a new user defined data type. Using this new data type AMOUNT, all variables that manipulate money related items can be declared. For example,
  ```
  AMOUNT     p, si;
  ```
- TIME is a new user defined data type. Using this new data type TIME, all variables that manipulate time related items can be declared. For example,
  ```
  TIME       t;
  ```
- RATE is a new user defined data type. Using this new data type RATE, all variables that manipulate rate of interest related items can be declared. For example,
  ```
  RATE       r;
  ```

13.3 Why Structures?

We know that an array is a collection of similar data items such as integer, float, char etc. If more number of data items of same data type are grouped, we normally use arrays. For example, the marks of 5 students can be stored using array as shown below:

```
marks[5] = {80, 90, 45, 99, 100};
```

The ordinary variables and arrays can handle variety of situations. But, in real world, we often deal with entities that are collection of dissimilar data types. For example, suppose we want to have the information related to a student. We may be interested in a group having:

♦ name of the student	(string type)
♦ university number	(string type)
♦ average marks	(**float** type)

Note that the above information is a collection of various items of dissimilar data types. So, arrays cannot be used (because, array is a collection of similar data types). This is where the structures are used.

13.4 Structures

Now, we shall see *"What is a structure? What is the syntax of a structure?"*

Definition: A structure is a collection of data items that are *logically related*. The data items may be of the same data type or dissimilar data types grouped together as a single entity. The data items inside the structure are defined as variables and are called *members* of the structure or *fields* of the structure. A structure is derived data type in C. A structure is identified using the keyword **struct.**

Ex 1: A student information to be grouped may consist of:

♦ name of the student	// array of characters
♦ university number	// integer or array of character
♦ average marks	// floating point number

The general syntax of a structure along with example is shown below:

Syntax

```
struct tag_name
{
        type1     member1;
        type2     member2;
        ......    ......
} ;
```

Example

```
struct student
{
        char    name[10];
        int     usn_number;
        float   average_marks;
};
```

where

- **struct** is the keyword which tells the compiler that a structure is being defined.
- tag_name is followed by the keyword **struct**. The tag_name is used as a shorthand for the declarations in braces.
- The variables in a structure such as member1, member2,….. declared within braces are called *members of the structure.* They are also called *fields of the structure.*
- The members can be any of the data types such as **int**, **char**, **float** etc.
- There should be semicolon at the end of closing brace

Note: The structure definition does not reserve any space in memory for the members. So, it is called a structure template and memory will not be allocated for the template. For example, in the above structure definition, we have declared *name, usn_number* and *average_marks* inside the structure whose tag name is *student*. Even then memory is not allocated because the structure **student** is only a template.

13.4.1 Structure definition and declaration

Once we know how to define the structure, the next question is *"Where to define a structure?"*

We know that all the variables are defined in the beginning of the function or before the function definition. On similar lines, the structures also should be defined either in the beginning of the function or before the function definition. Normally, the structures are defined before all the functions where global variables are defined.

A structure can be defined using **three** different ways as shown below:

- → Tagged structure
- → Tagless structure (Nameless or un-named structure)
- → Typedefined structure

In the following subsections, we see how various structures are defined and declared.

13.4.2 Tagged Structure

Now, let us see "*What is a tagged structure? Explain with example*"

Definition: In the structure definition, the keyword **struct** can be followed by an identifier. This identifier is called *tag name*. The structure definition associated with a tag name is called *tagged structure.* The syntax of tagged structure definition is shown below:

Syntax

```
struct tag_name
{
        type1     member1;
        type2     member2;
        ……        ……
} ;
```

Example

```
struct student
{
        char    name[10];
        int     usn_number;
        float   average_marks;
};
```

- **struct** is the keyword
- **student** is the name of the structure. It is also called tag name.
- **struct student** together represent the data type. Since it is derived from basic data types, it is called derived data type.
- Just by defining the structure as shown above, note that memory is not reserved for the structure. Memory is allocated for structure only when the variables are declared.

Once we have seen how to define a structure, the next question is *"How to declare structure variables?"* The syntax along with examples is shown below:

Syntax:

```
        struct   tag_name    v1,v2,…..vn;
          ↓         ↓           ↓
ex 1:   struct   student     cse, ise;
```

Observe the following points:

- Once the variables are declared, then memory for the variables *cse* and *ise* are allocated.
- The number of bytes allocated for the variable ***cse*** is shown below:
  - 10 bytes are allocated for the field **name**.
  - 2 bytes for the field **usn_number**.
  - 4 bytes for the field **average_marks**.

  $\}$ **10 + 2 + 4 = 16 bytes**

Note: Another 16 bytes are allocated for variable *ise*. The complete structure definition along with structure declaration is shown below:

```
struct student                 /* Structure definition */
{
        char    name[10];
        int     roll_number;
        float   average_marks;
};                                     /* No memory is allocated for structure */

/* Structure declaration */
struct student cse, ise;               /* Memory is allocated for the variables */
```

13.4.3 Structure without tag (Tag less structure)

Now, let us see *"What is a structure without tag? Explain with example"*

Definition: In the structure definition, the keyword **struct** can be followed by an identifier. This identifier is called *tag name and* it is optional. The structure definition without tag name is called *structure without tag*. The syntax of defining this structure is shown below:

Syntax

```
struct
{
        type1     member1;
        type2     member2;
        ……        ……
} v1, v2,….vn;
```

Note: Here, v1, v2,….vn are variables

Example

```
struct
{
        char    name[10];
        int     usn_number;
        float   average_marks;
}cse, ise;
```

Note: Here, *cse*, *ise* are variables

Observe the following points:

- In the absence of tag name immediately after the right brace we have to declare the variables. In the above example, there is no tag name and hence immediately after the closing right brace, we have a list of variables *cse* and *ise*.
- *name*, *usn_number* and *average_marks* are members of a structure and are not variables. So, they themselves do not occupy any memory.
- But, once the structure definition is associated with variables such as ***cse*** and ***ise*** the compiler allocates memory for the structure variables.
- The number of bytes allocated for the variable ***cse*** is shown below:
  - 10 bytes are allocated for the field **name**.
  - 2 bytes for the field **roll_number**.
  - 4 bytes for the field **average_marks**.

  } **10 + 2 + 4 = 16 bytes**

Note: The size of the memory allocated is the sum of size of individual members. So, totally 16 bytes are reserved for the variable *cse* and another 16 bytes are reserved for the variable *ise*.

13.4.4 Type-Defined Structure

Now, let us see *"What is a type-defined structure? Explain with example"*

Definition: The structure definition associated with keyword **typedef** is called *type-defined structure*. This is the most powerful way of defining the structure. The syntax of type-defined structure along with example is shown below:

```
typedef struct
{
        type1     member1;
        type2     member2;
        ……        ……
} TYPE_ID;
```

```
typedef struct
{
        char    name[10];
        int     roll_number;
        float   average_marks;
} STUDENT;
```

where

- **typedef** and **struct** are keywords.
- The closing brace must end with type definition name (TYPE_ID in the syntax shown) which in turn ends with semicolon. In the example, **STUDENT** is the new data type using which variables can be declared.

Once we have seen how to define a type-defined structure, the next question is *"How to declare type-defined structure variables?"*

The syntax along with examples is shown below:

Syntax:

```
TYPE_ID    v1,v2,…..vn;   // v1, v2,….vn are variables
   ↓            ↓
STUDENT    cse, ise;
```

ex 1: **STUDENT** cse, ise;

Here, STUDENT is the new data type defined using type-definied structure. **Note:** Normally all **typedef** statements are defined at the beginning of the file immediately after #includes and #define statements and just before global variables and function prototypes.

The complete structure definition along with **typedef** definition and declaration is shown below:

```
/* Structure definition */
typedef struct
{
        char    name[10];
        int     roll_number;
        float   average_marks;
} STUDENT;                      /* No memory is allocated for structure */

/* Structure declaration */
STUDENT cse, ise;               /* Memory is allocated for the variables */
```

The above structure definition and declaration can also be written as shown below:

```
/* Structure definition */
struct student
{
        char    name[10];
        int     roll_number;
        float   average_marks;
};                          /* No memory is allocated for structure */

typedef struct student STUDENT;/* STUDENT is user-defined data type */

/* Structure declaration */
STUDENT cse, ise;           /* Memory is allocated for the variables */
```

Note: The word **student** which is written using fully lowercase letters is the tag name of the structure whereas, **STUDENT** which is written using fully capital letters is the user-defined data type.

13.4.5 Structure initialization

Now, let us see *"How structures are initialized?"* The syntax of initializing structure variables is similar to that of arrays i.e., all the data items will be enclosed within braces i.e., **'{'** and **'}'** and are separated by commas. The syntax is shown below:

struct tag_name **variable** = {v1, v2, …vn};

where

- **"struct tag_name"** is derived data type.
- **v1, v2,……vn** are all the initial values. These values are called initializers; they should be separated by commas and should be enclosed between '{' and '}'

Example 13.5: Structure definition, declaration and initialization

Consider the following structure definition:

```
struct employee
{
        char name[20];
        int   salary;
        int   id;
};
```

The above structure can be initialized as shown below:

```
struct employee                            /* Define employee structure        */
{                                          /* Various members are              */
        char name[20];                     /*      member name                 */
        int  salary;                       /*      member salary               */
        int  id;                           /*      member employee id          */
} a = {"Shivashankar", 10950, 2001};       /* Variable a has values            */
                                           /*      name = “Shivashankar”       */
                                           /*      salary = 10950              */
                                           /*      id = 2001                   */
```

The above example can also be written as shown below:

Example 13.6: Structure definition, declaration and initialization

```
        struct employee
        {
                char name[20];
                int  salary;
                int  id;
        };

        struct employee a = {"Shivashankar", 10950, 2001};
```

Note: The various members of the structure have the following initial values:

- The first member **name** has the value **"Shivashankar"**
- The second member **salary** has the value **10950**
- The last member in the structure **id** has the value **2001**.

Example 13.7: Initialization during structure declaration with more than one variable

```
                struct employee
                {
                        char name[20];
                        int  salary;
                        int  id;
                } a, b = {"Shivashankar", 10950, 2001};
```

Note: The various members of the structure have the following initial values for the variable **b**:

- The first member **name** has the value **"Shivashankar"**
- The second member **salary** has the value **10950**
- The last member in the structure **id** has the value **2001.**
- Only the variable **b** is initialized and the variable **a** is not initialized.

Example 13.8: Initialization during structure declaration with more than one variable

The initialization of both the variables **a** and **b** can be achieved as shown below:

```
/* Structure definition and Initialization */
struct employee
{
        char name[20];
        int   salary;
        int   id;
}
a = {"Rama", 20000, 1000}, b = {"Shivashankar", 10950, 2001};
```

Initializing both variables **a** and **b** can also be done as shown below:

Example 13.9: Initialization during structure declaration with more than one variable

```
struct employee
{
        char name[20];
        int   salary;
        int   id;
};

struct employee a = {"Rama", 20000, 1000};
struct employee b = {"Shivashankar", 10950, 2001};
```

Now, let us see some important points to remember when we initialize the structures.

- The members of the structure cannot be initialized in the structure definition. For example,

  ```
  struct employee
  {
          char name[20] = "Shivashankar";
          int   salary = 20000;
          int   id = 25;
  };
  ```

  is invalid.

- *The initial values are assigned to members of the structure on one-to-one basis i.e., the values are assigned to various members in the order specified from the first member.* For example, the following statement:

  ```
  struct employee a = {"Shivashankar", 10950, 2001};
  ```

  is valid. Here, the string "Shivashankar" will be assigned to the first member, 10950 is assigned to the second member and 2001 will be assigned to the third member.

- *During partial initialization (i.e., if there are fewer initial values i.e., few initializers, than the members of a structure), the values are assigned to members in the order specified and the remaining members are initialized with default values.* If the remaining members are of type integer or float, they are initialized with the default value zero and if the remaining members are of type string or character, they are initialized with default value of '\0'. For example, in the initialization statement

  struct employee a = {"Shivashankar"};

  observe that the string "Shivashankar" will be assigned to structure member **name**. The other members of the structure namely **salary** and **id** are initialized to zero by default.

- *During initialization, the number of initializers (i.e., the values to be initialized to members of a structure), should not exceed the number of members.* It leads to syntax error. For example, for the following statement

  struct employee a = {"Rama",10950, 2001,10.2};

  the compiler issues a syntax error saying "too many initializers".

- *During initialization, there is no way to initialize members in the middle of a structure without initializing the previous members.* For example,

  struct employee a = {10950, 2001};

  is invalid, even though it is syntactically correct. Because, without initializing the first member namely **name**, we are trying to initialize last two members. In this case, the number 10950 will be copied into **name** field, the number 2001 will be copied into **salary** field and the result is unpredictable.

13.5 Accessing structures

We know that the data stored in variables can be accessed using variable names and manipulated using expressions and operators. On similar lines, the data stored in structures can be accessed using structure variables and manipulated using expressions and operators.

Before manipulating let us see *"How structure members are accessed?"* The members of a structure are separate entities and hence they should be processed individually. A member of a structure can be accessed by specifying the variable name followed by a period (also called dot) which in turn is followed by the member name using the syntax shown below:

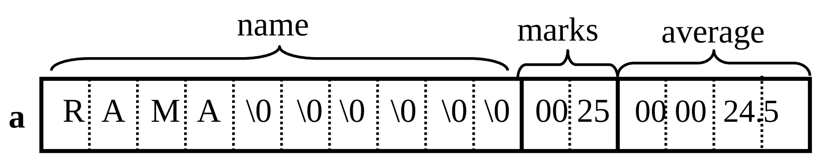
variable.member

For example, consider the structure definition and initialization shown below:

```
struct student
{
        char    name[10];
        int     marks;
        float   average;
};

struct student a = {"RAMA",25, 24.5};
```

The memory representation for the above structure is shown below:



The various members can be accessed using the variable **a** as shown below:

- By specifying **a.name** we can access the string **"RAMA".**
- By specifying **a.marks** we can access the value **25**
- By specifying **a.average** we can access the value of **24.5**

Now, the question is "How to display the various members of a structure?" The various members of a structure can be accessed and printed as shown below:

Programming statements	Output
printf("%s\n", a.name);	Ram
printf("%d\n", a.marks);	25
printf("%f\n", a.average);	24.5

Once we know how to display the members of a structure, let us see "How to read the values for various members of a structure?" We know that format specifications such as %s %d %f are used to read a string, an integer and a float. The same format specifications can be used to read the members of a structure. For example, we can read the **name** of a student, the **marks** and **average marks** as shown below:

```
gets (a.name);                  /* or scanf("%[^\n]);   */
scanf ("%d", &a.marks);
scanf ("%f", &a.average);
```

Now, let us see *"How to write a program to simulate the multiplication of two fractions?"* Before writing the program, let us give the design procedure by taking example:

Design: Consider two fractions 2/8 and 3/7. These two numbers can be multiplied and as shown below:

$$\frac{2}{8} * \frac{3}{7} = \frac{6}{56}$$

Since there are no fractional data types in C language, this can be simulated using structures. A fraction number can be defined as a structure with two members namely ***n*** stands for numerator and *d* stands for **denominator** as shown below:

```
typedef struct
{
        int     n;                  /* Numerator */
        int     d;                  /* Denominator */
} FRACTION;
```

Let **a** and **b** are two given fractions and **c** is the resultant fraction. The resultant fraction **c** can be computed as shown below:

```
        c.n = a.n * b.n;
        c.d = a.d * b.d;
```

Now, the complete program is shown below:

Example 13.10: Program to simulate the multiplication of two fractions

```
#include <stdio.h>

typedef struct
{
        int     n;              /* Numerator */
        int     d;              /* Denominator */
} FRACTION;

void main()
{
        FRACTION a, b, c;

        printf("Enter fraction1 in the form x/y: ");
        scanf("%d/%d",&a.n, &a.d);

        printf("Enter the fraction2 in the form x/y: ");
        scanf("%d/%d",&b.n, &b.d);

        c.n = a.n * b.n;
        c.d = a.d * b.d;

        printf("%d/%d * %d/%d = %d/%d\n",a.n, a.d,
                                          b.n, b.d,
                                          c.n,c.d);
}
```

TRACING

```
Enter fraction1: 3/8

Enter fraction2: 4/7
```

Output

```
3/8 * 4/7 = 12/56
```

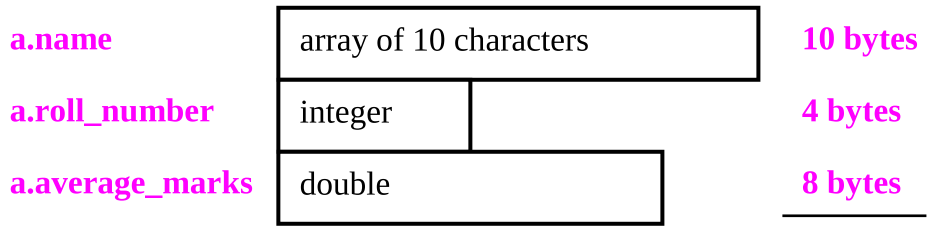
13.6 sizeof a structure (Concept of Slack Bytes)

Now, let us see *"What is the size of the structure?"*

Definition: The size of a structure is defined as the sum of sizes of each member of the structure. For example, consider the structure declaration shown below:

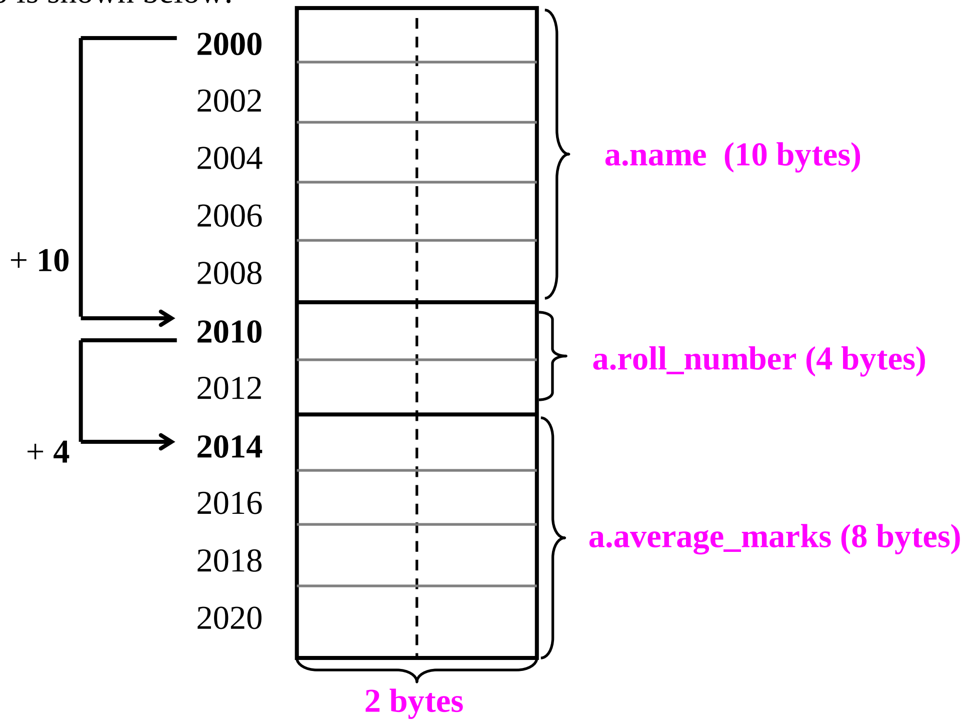
```
struct student
{
        char      name[10];          /* Assume size of char is 1 byte */
        int       roll_number;       /* Assume size of int is 4 bytes */
        double    average_marks;     /* Assume size of double is 8 bytes */
} a;
```

The size of each member of the structure is shown below:



So, total size of the structure = **22 bytes**

If 2000 is the starting address of the variable **a,** then the starting address of each member depends on sum of sizes of previous members and the complete memory map is shown below:



Note: The address of a member = address of its immediate previous member + the size of its immediate previous member

Address of member **name = 2000**
Address of member **roll_number = 2000 + 10 = 2010**
Address of member **average_marks = 2010 + 4 = 2014**

Note: The address of each member is greater than the address of its previous member and hence, address of each member is different.

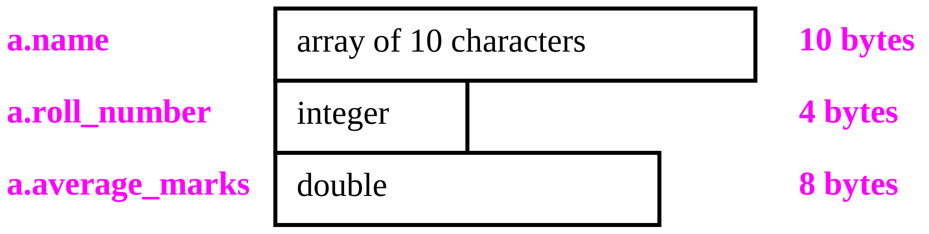
Some times, the size of the structure will not be equal to sum of sizes of the individual members. This is because of ***slack bytes.*** Now, let us see *"What are slack bytes?"*

Definition: In some computers, the memory for the members of a structure is allocated in sequence based purely on the size of individual members. In some computers, the memory for the members of a structure is allocated at certain boundaries called "word boundaries". In such case, extra bytes are padded at the end of each member whose size is less than the size of the largest data type so that the address of each member starts at the word boundary. These extra bytes that are inserted to maintain the boundary requirements are called ***slack bytes.*** These extra bytes do not contain any valid information and are useless wasting the memory space.

For example, if **char, int, float** and **double** are the data types of members of a structure, then a member whose type is **double** occupy more memory. Assuming sizeof **double** is 8 bytes, then the address of each member is multiple of 8. Consider the structure declaration shown below:

```
struct student
{
        char      name[10];          /* Assume size of char is 1 byte */
        int       roll_number;       /* Assume size of int is 4 bytes */
        double    average_marks;     /* Assume size of double is 8 bytes */
} a ;
```

The size of each member of the structure is shown below:



Sum of sizes of individual structure members = **22 bytes**

Let us see "What is the size of structure if slack bytes are introduced?" If 2000 is the starting address of the variable **a,** then the starting address of each member depends on sum of sizes of previous members along with the word boundary and the complete memory map is shown below:

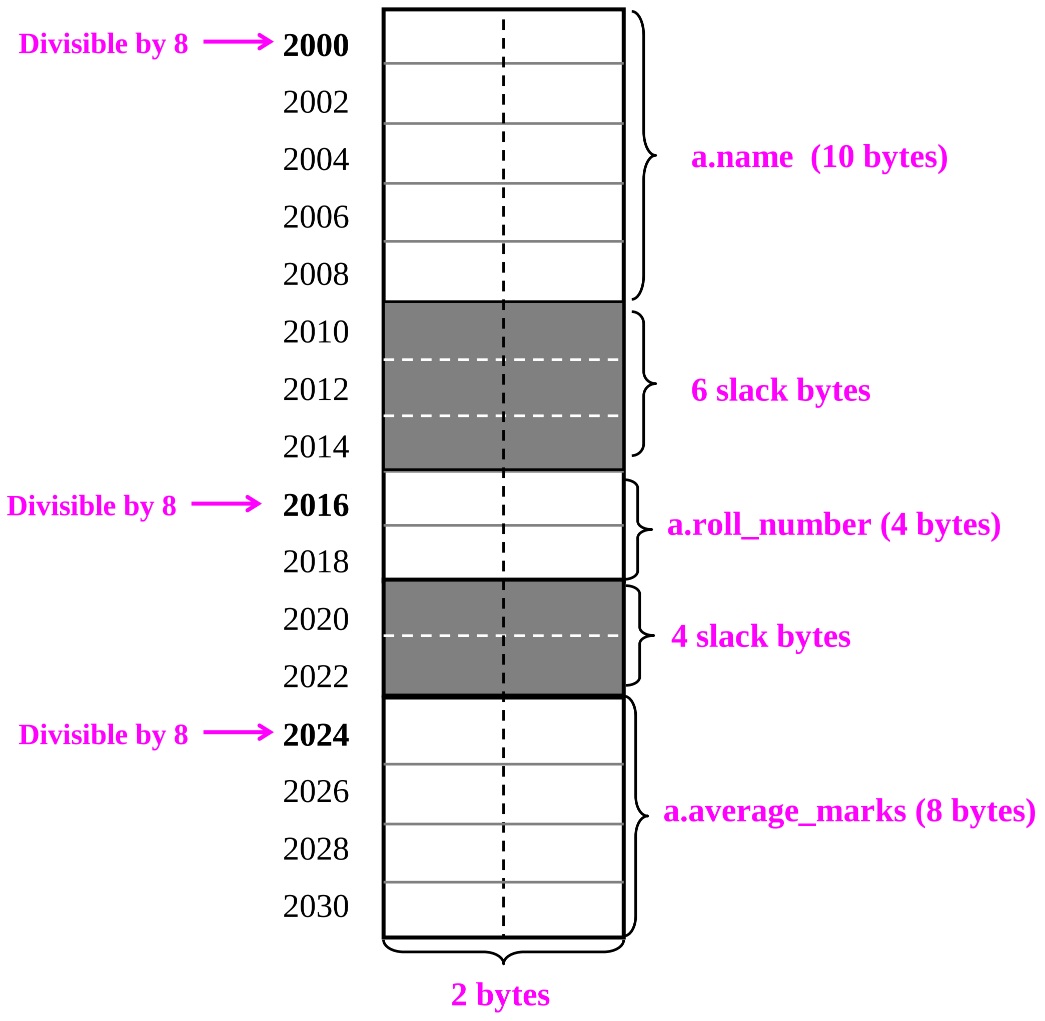


Fig. 12.5.2 Memory map showing slack bytes

Observe the following points:

- In the structure definition, **double** is the largest data type with size 8 bytes. So, the starting address of each member should be divisible by 8.
- The size of the structure is 32 bytes which is greater than the sum of sizes of each member. So, when slack bytes are used, the size of a structure is greater than or equal to the sizes of its individual members.

- Data can be accessed much faster way if data is present in word boundaries. So, even though slack bytes do not contain valid information, they are useful so that data is available always in word boundaries and can be accessed very fast.

13.7 Structure operations

In this section, let us see "What are the various operations that can be performed on structures?" The various operations that can be performed on structures are shown below:

- Copying of structure variables
- Comparison of two structure variables
- Arithmetic operations on structures

13.7.1 Copying of structure variables

Consider the structure definition and declaration statements shown below:

```
struct                          struct
{                               {
        char name[10];                  char name[10];
        int  salary;                    int  salary;
        int  id;                        int  id;
} a, b;                         } c, d;
```

Note: Observe that even though the members of both structures are same in number and type, both are considered to be of different structures.

- Now, the question is "Is copying one structure variable to other structure variable of same type is allowed or not allowed?"

  Yes, it is allowed. So, the statements

  ```
  a  = b;                     c = d;
  b = a;          and         d = c;
  ```

  are valid.

- "Is copying one structure variable to other structure variable of different type is allowed or not allowed?"

  No, it is not allowed. So, the following statements

  ```
  a = c;                      c = a;
  b = d;          and         d = b;
  ```

  are invalid.

- Now, the question is "How to copy the data stored in one structure variable to other structure variable of different data type?"

  This is possible by copying the individual members of the same type and size. So, if we want to copy structure *c* to structure *a*, it can be achieved by copying the individual members as shown below:

  ```
  strcpy(a.name, c.name);
  a. salary = c.salary;
  a.id = c.id;
  ```

13.7.2 Comparison of two structure variables or members

Now, the question is "How to compare two structure variables?"

- Comparing two structure variables of same type or dissimilar types is not allowed. For example, if *a* and *b* are two structure variables, then the following expressions results in error:

  ```
  a > b;          // Error
  a < b;          // Error
  a == b;         // Error
  a != b;         // Error
  ```

- But, the members of two structures can be compared if the type of members are **int**, **float**, **double** or **char**.

- So, the members of two structures can be compared using relational operators. For example,

  ```
  c.marks > d.marks;          // OK
  c.average > d.average;      // OK
  ```

  are valid relational operations that can be performed on members of a structure.

13.7.3 Arithmetic operations on structures

Now, the question is "How to perform arithmetic operations on structure variables?"

- Arithmetic operations on two structure variables of same type or dissimilar types is not allowed. For example, if *a* and *b* are two structure variables, then the following expressions results in error:

  ```
  a + b;          // Error
  a * b;          // Error
  a / b;          // Error
  a - b;          // Error
  ```

- But, arithmetic operations can be performed on members of a structure if the type of member is **int**, **float**, **double** or **char**.

- So, the members of a structure can be manipulated by using expressions and operators. For example,

```
++c.marks;                 // OK
c.marks++;                 // OK
c.marks + d.marks;         // OK
c.average + d.average;     // OK
```

  are all valid arithmetic operations that can be performed on members of a structure.

13.8 Pointer to Structures

Now, let us see *"What is pointer to a structure?"*

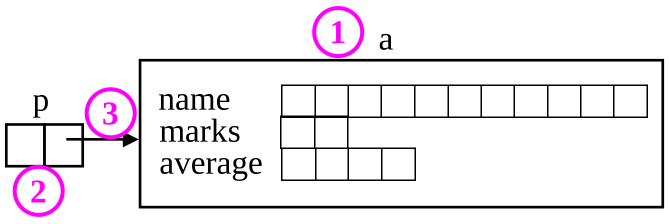
Definition: A variable which contains address of a structure variable is called *pointer to a structure.* For example, consider the following declaration:

```
typedef struct
{
        char     name[10];
        int      marks;
        float    average;
} STUDENT ;
```

```
(1)  STUDENT    a;

(2)  STUDENT    *p;

(3)  p = &a;
     ……
```



In the above program segment

(1) Memory is allocated for the structure variable *a*.

(2) Memory is allocated for pointer to a structure.

(3) The address of structure *a* is copied into pointer variable *p*.

Now, the variable ***p*** holds the address of a structure. So, *p* is a pointer to a structure.

Now, the question is *"How to access the members of structure?"* The members of the structure can be accessed using three methods:

- Using dot(.) operators
- Using de-referencing operator * and dot (.) operator
- Using selection operator (->)

Using de-referencing operator * and dot (.) operator: If **p** is pointer to a structure, then the structure itself can be accessed using indirection operator as shown below:

```
        *p        /* Refers to the whole structure */
```

Once the structure is accessed, each member of the structure can be accessed using dot operator as shown below:

```
    (*p).name               /* Access the name */
    (*p).roll_number        /* Access roll number */
    (*p).average_marks      /* Access average marks */
```

Note: The parentheses in all the three expressions are necessary. We should not omit the parentheses. For example,

```
    *p.name                 /* Invalid way of accessing the member */
    *p.roll_number          /* Invalid way of accessing the member */
    *p.average_marks        /* Invalid way of accessing the member */
```

Using selection operator: This operator is also called *arrow operator*. If **p** is pointer to a structure, then the members of the structure can also be accessed using **selection operator** denoted by -> (which is formed by the minus sign and greater than symbol). Using this operator, various members of the structure can be accessed as shown below:

```
    p->name                 /* Access the name */
    p->roll_number           /* Access roll number */
    p->average_marks         /* Access average marks */
```

13.9 Complex structures

In this section, let us study some of the complex structures such as:

- Nested structures
- Structures containing arrays
- Structures containing pointers
- Arrays of structures

Let us discuss each of these one by one

13.9.1 Nested structures

Let us see *"What is a nested structure?"*

Definition: A structure inside a structure is called nested structure. As we declare variables inside a structure, a structure can also be declared inside other structure. So, a structure whose member itself is a structure is called nested structure.

For example, consider a structure consisting of marks of three subjects:

```
typedef struct
{
        int marks1;          /* Marks in subject 1 */
        int marks2;          /* Marks in subject 2 */
        int marks3;          /* Marks in subject 3 */
} MARKS;
```

The student information consisting of name, university seat number and the marks of three subjects can be represented using a structure as shown below:

```
typdef struct
{
        char        name[10];        /* Name of the student */
        int         usn;             /* University serial number */

        MARKS       m;               /* Marks of three subjects */

} STUDENT;           /* STUDENT is the user defined data type */
```

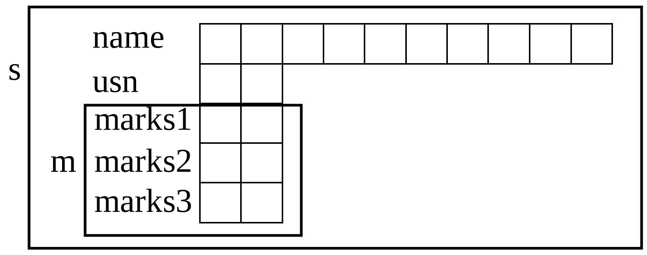
Note the following points in the above structure definition:

- The member *m* is declared as type MARKS which is a structure.
- The structure MARKS is used as a type in another structure STUDENT.
- Since, the structure MARKS is defined inside a structure STUDENT, we say the structure **STUDENT** is a nested structure.

Consider the following declaration:

```
STUDENT    s;            /* Variable s is of type STUDENT */
```

The memory allocated for variable *s* can be pictorially represented as shown below:



Now, using the variable **s**, we can access the various fields using dot operator as shown below:

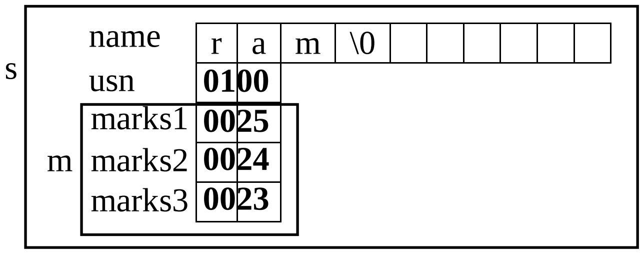
```
s.name          /* Access name of the student */
s.USN           /* Access USN of a student */
s.m.marks1      /* Marks of subject1 */
s.m.marks2      /* Marks of subject 2*/
s.m.marks3      /* Marks of subject 3 */
```

Now, the question is "How to initialize the nested structure members?" A nested structure can be initialized as shown below:

- The variable in the declaration must be followed by "=" sign and followed by data items
- The data items that are to be initialized must be separated by commas
- The data items that are to be initialized must be enclosed within braces as shown below:

```
STUDENT   s = { "ram",
                100,
                {25, 24, 23}
              };
```

The data items thus initialized are stored in memory as shown below:



Now, the question is "How to access the members of a nested structure?" The data stored in each member can be accessed using dot operator as shown below:

```
s.name        // ram
s.usn         //100
s.m.marks1    // 25
s.m.marks2    //24
s.m.marks3    //23
```

Now, the question is "How to display the data stored in nested structure members?" The data stored in each member can be accessed using dot operator and can be printed using printf as shown below:

```
        printf("%s", s.name);           // ram
        printf("%d", s.usn);            //100
        printf("%d", s.m.marks1);       // 25
        printf("%d", s.m.marks2);       //24
        printf("%d", s.m.marks3);       //23
```

Now, the question is "How to read the data into nested structure members?" The data can be read from the keyboard and stored in various members using scnaf() and dot operator as shown below:

```
        scanf("%s", s.name);            // ram
        scanf("%d", &s.usn);            //100
        scanf("%d", &s.m.marks1);       // 25
        scanf("%d", &s.m.marks2);       //24
        scanf("%d", &s.m.marks3);       //23
```

The complete program to read student details and to print student details is shown below:

Example 13.11: Program to read and display student details using nested structures.

```
#include <stdio.h>

typedef struct
{
        int marks1;                     /* Marks of subject 1 */
        int marks2;                     /* Marks of subject 2 */
        int marks3;                     /* Marks of subject 3 */
} MARKS;

typedef struct
{
        char        name[10];           /* Name of the student */
        int         USN;                /* University serial number */
        MARKS       m;                  /* Marks of 3 subjects */
} STUDENT;

void main()
{
        STUDENT     s;

        /* To read student details */
        printf("Name = ");              scanf("%s", s.name);
        printf("USN = ");               scanf("%d", &s.usn);
        printf("marks1 = ");            scanf("%d", &s.m.marks1);
        printf("marks2 = ");            scanf("%d", &s.m.marks2);
        printf("marks3 = ");            scanf("%d", &s.m.marks3);

        /* To display student details */
        printf("Name = %s\n", s.name);
        printf("USN = %d\n", s.usn);
        printf("marks1 = %d\n", s.m.marks1);
        printf("marks2 = %d\n", s.m.marks2);
        printf("marks3 = %d\n", s.m.marks3);
}
```

13.9.2 Structures containing arrays

Now, the question is *"Is it possible to use arrays within structures?"* The answer is yes.

- An array of ***int***, ***float, char*** and ***double*** data type can be used within the structures.
- Array of structures can also be used as members (discussed later)

For example, consider the following structure definition:

```
        typedef struct
        {
                char    name[10];               /* Name of the student */
                int     usn;                    /* University serial number */
                int     marks[3]                /* Marks of 3 subjects*/
        } STUDENT;
```

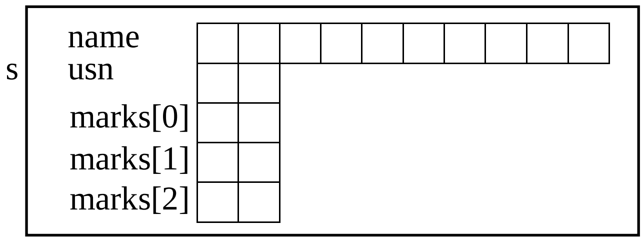
where

- **STUDENT** is user-defined data type.
- **marks** is an array of three integers. This array is used to store the marks of three subjects.
- **name** is an array of characters which is used to hold the name of the student

By defining a structure as shown above memory space is not allocated. To allocate the memory space, a variable has to be declared as shown below:

```
        STUDENT s;
```

By looking at the above declaration, the compiler allocates a block of memory for the variable *s* as shown below:



Using the variable **s**, we can access the various fields using dot operator as shown below:

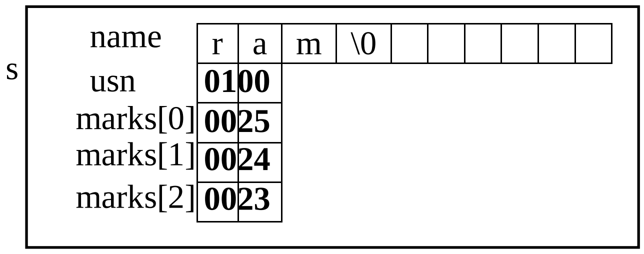
s.name	/* Access name of the student */
s.usn	/* Access USN of a student */
s.marks[0]	/* Marks of subject1 */
s.marks[1]	/* Marks of subject 2*/
s.marks[2]	/* Marks of subject 3 */

Now, the question is "How to initialize the nested structure members?" A nested structure can be initialized as shown below:

- The variable in the declaration must be followed by "=" sign and followed by data items
- The data items that are to be initialized must be separated by commas
- The data items that are to be initialized must be enclosed within braces as shown below:

```
STUDENT   s = { "ram",
                100,
                {25, 24, 23}
              };
```

The data items thus initialized are stored in memory as shown below:



Now, the question is "How to access the members of a nested structure?" The data stored in each member can be accessed using dot operator as shown below:

```
s.name        // ram
```

```
        s.usn           //100
        s.marks[0]      // 25
        s.marks[1]      //24
        s.marks[2]      //23
```

Now, the question is "How to display the data stored in nested structure members?" The data stored in each member can be accessed using dot operator and can be printed using printf as shown below:

```
    printf("%s", s.name);          // ram
    printf("%d", s.usn);           //100
    printf("%d", s.marks[0]);      // 25
    printf("%d", s.marks[1]);      //24
    printf("%d", s.marks[2]);      //23
```

Now, the question is "How to read the data into nested structure members?" The data can be read from the keyboard and stored in various members using scnaf() and dot operator as shown below:

```
    scanf("%s", s.name);           // ram
    scanf("%d", &s.usn);           //100
    scanf("%d", &s.marks[0]);      // 25
    scanf("%d", &s.marks[1]);      //24
    scanf("%d", &s.marks[2]);      //23
```

The complete program to read student details and to print student details is shown below:

Example 13.12: Program to read, display student details using arrays inside structures.

```
#include <stdio.h>

typedef struct
{
        char        name[10];    /* Name of the student */
        int         USN;         /* University serial number */
        int         marks[3];    /* Marks of 3 subjects */
} STUDENT;

void main()
{
        STUDENT   s;
        /* To read student details */
        printf("Name = ");              scanf("%s", s.name);
        printf("USN = ");               scanf("%d", &s.usn);
        printf("marks1 = ");            scanf("%d", &s.marks[0]);
        printf("marks2 = ");            scanf("%d", &s.marks[1]);
        printf("marks3 = ");            scanf("%d", &s.marks[2]);

        /* To display student details */
        printf("Name = %s\n", s.name);
        printf("USN = %d\n", s.usn);
        printf("marks1 = %d\n", s.marks[0]);
        printf("marks2 = %d\n", s.marks[1]);
        printf("marks3 = %d\n", s.marks[2]);
}
```

13.9.3 Structures containing pointers

We have seen that members of a structure can be arrays, ints, floats etc. Now, the question is "Is it possible to use pointers within structures?" Yes, a pointer can also be used as a member of the structure.

For example, the following structure definition defines a new data type called DATE:

```
        typedef struct
        {
                int     day;
                int     month;
                int     year;
        } DATE;
```

In the above type definition, **day** is represented as integer. Suppose, we have the following declarations in our program:

```
        char    mon[]  = "Monday";
        char    tue[]  = "Tuesday";
        char    wed[]  = "Wednesday";
        char    thu[]  = "Thursday";
        char    fri[]  = "Friday";
        char    sat[]  = "Saturday";
        char    sun[]  = "Sunday";
```

Note that the days are represented as strings. If it is required to store the days as strings, then the earlier type definition and declaration with respect to DATE can be written as shown below:

```
typdef struct
{
        char    *day;
        int     month;
        int     year;
} DATE;

DATE d;
```

Suppose we want to store the date as “Thursday 11 1965”. This can be done using the following statements:

```
d.day = “Thursday”;  /* Note: pointer to a string is copied instead of string */
d.month = 11;
d.year = 1965;
```

13.10 Array of structures

As we have an array of integers, we can have an array of structures also. For example, suppose we want to store the information of 10 students consisting of name, marks and year of passing. The structure definition can be written as shown below:

```
/* Definition of student structure */
typedef struct
{
        char name[20];
        int   marks;
        int   year;
} STUDENT;
```

To store the information of more number of students, we can have the following declaration:

```
STUDENT a[10];
```

Note: **a** is an array consisting of 10 memory locations. Each location is of type STUDENT and hence the information of 10 students can be stored.

Now, the next question is “How to access the elements of array and information related to a particular student?” Since ***a*** is an array, each student information can be accessed by specifying the index along with member name. The information of i^{th} student can be accessed by specifying:

```
a[i].name      /* Access the name of ith student */
a[i].marks     /* Access the marks of ith student */
a[i].year      /* Access the year */
```

Now, let us write a program to sort the student details in alphabetical order. The function to read student details using nested structures is shown below:

Example 13.13: Function to read the student details

```
void read_student_details(int n, STUDENT a[])
{
        int i;

        for (i = 0; i < n; i++)
        {
                printf("Name = ");          scanf("%s", a[i].name);
                printf("USN = ");           scanf("%d", &a[i].USN);
                printf("Marks1 = ");        scanf("%d", &a[i].m.marks1);
                printf("Marks2 = ");        scanf("%d", &a[i].m.marks2);
                printf("Marks3 = ");        scanf("%d", &a[i].m.marks3);
        }
}
```

The function to arrange student's information in ascending order using nested structures is shown below:

Example 13.14: Function to arrange student's information in ascending order using nested structures.

```
void sort_student_details (int n, STUDENT a[])
{
        int             i, j;
        STUDENT         temp;

        for ( j = 1; j < n; j++)  /* Sort the student records in alphabetical order */
        {
                for ( i = 0; i < n-j; i++)
                {
                        if (strcmp(a[i].name, a[i+1].name) > 0)
                        {
                                temp = a[i];
                                a[i] = a[i+1];
                                a[i+1] = temp;
                        }
                }
        }
}
```

The function to display student's information using nested structures is shown below:

Example 13.15: Function to display student information using nested structures.

```
void display_student_details (int n, STUDENT a[])
{
        int     i;

        printf("The sorted record\n");
        for ( i = 0; i < n; i++)
        {
                printf("%s %d %d %d %d\n",a[i].name,a[i].USN,a[i].m.marks1,
                                        a[i].m.marks2, a[i].m.marks3);
        }
}
```

Example 13.16: Program to arrange student's information in ascending order using nested structures.

```
#include <stdio.h>
#include <string.h>
typedef struct
{
        int marks1;                     /* Marks of subject 1 */
        int marks2;                     /* Marks of subject 2 */
        int marks3;                     /* Marks of subject 3 */
} MARKS;

typedef struct
{
        char            name[10];       /* Name of the student */
        int             USN;            /* University serial number */
        MARKS           m;              /* Marks of 3 subjects */
} STUDENT;

/* Include: Example 13.13: To read student details */
/* Include: Example 13.14 : To sort student details */
/* Include: Example 13.15 : To display student details */

void main()
{

        STUDENT     a[9];
        int             n;

        printf("Enter the number of students\n");
        scanf("%d", &n);

        read_student_details(n, a);

        sort_student_details(n, a);

        display_student_details(n, a);
}
```

Now, let us search for student name in an array of N student records using binary search. The concept of searching is already discussed in 9.6.2. The same algorithm is used to search for a string in an array of strings.

Example 13.17: Function search for a string using binary search

```
int binary_search ( char key[],  STUDENT a[20], int n)
{
        int     low,  high,  mid;

        low = 0;                                        /* Initialization */
        high = n-1;
        ..........................................................................................
        while ( low <= high )
        {
                mid = ( low + high ) / 2;               /* Find the mid point */
                ..................................................................................
                flag = strcmp(key, a[mid].name);
                ..................................................................................
                if (flag == 0 ) return mid;             /* Item found */

                if ( flag < 0 )  high = mid – 1;        /* To search left part */

                if ( flag > 0 ) low = mid + 1;           /* To search right part */
        }
        return -1;      /* Unsuccessful search */
}
```

The C program that uses the above function can be written as shown below:

Example 13.18: C program to search for a name using binary search

```
#include <stdio.h>
#include <string.h>
struct student
{
        char name[10];          /* Name of the student */
        int  USN;               /* University serial number */
        int  marks[3];          /* To hold marks of 3 subjects */
};

typedef struct student STUDENT;         /* STUDENT is the new data type */

/* Include: Example  13.17:  Function binary search */

void main()
{
        int             n, pos, i;
        STUDENT         a[20];          /* An array to store student information*/
        char            key[20];        /* Name to be searched    */

        printf ("Enter the number of elements\n");
        scanf ("%d", &n);

        printf("Enter the details of %d students\n", n);

        for (i = 0; i < n; i++)
        {
                printf("Details of %d student\n", i+1);
                printf("Name = ");      scanf(" %[^\n]", a[i].name);
                printf("USN = ");       scanf("%d", &a[i].USN);
                printf("Marks1 = ");    scanf("%d", &a[i].marks[0]);
                printf("Marks2 = ");    scanf("%d", &a[i].marks[1]);
                printf("Marks3 = ");    scanf("%d", &a[i].marks[2]);
        }
        printf ("Enter the name to be searched\n");
        gets(key);
       ..........................................................................
        pos = binary_search (key, a, n);
       ..........................................................................
        if (pos == -1)
                printf ("Name not found \n");

        else
                printf ("Name found at pos: %d\n", pos);
}
```

13.11 Structures and functions

We should note that structures are more useful if we are able to pass them to functions and return them. Now, let us see *"What are the various methods of passing structures to functions?"* The structures or structure members can be passed to the functions in various ways as shown below:

$$\begin{tikzcd}
 & \text{By passing individual members of structure} \\
\text{Various ways of passing structures to functions} \arrow[r] \arrow[ru] \arrow[rd] & \text{By passing the whole structure} \\
 & \text{By passing structures through pointers}
\end{tikzcd}$$

Now, let us discuss each of these in detail.

13.11.1 Sending individual members

A function can be called by passing the members of a structure as actual parameters. For example, the program to simulate the multiplication of two fractions shown in example 13.10, section 13.5 can be written using a function as shown below:

Example 13.18: Program to simulate the multiplication of two fractions by passing individual members to the function

```
#include <stdio.h>

typedef struct
{
        int     n;
        int     d;
} FRACTION;

int multiply(int x, int y)
{
        return x * y;
}

void main()
{
        FRACTION a, b, c;
        printf("Enter fraction1 in the form x/y: ");
        scanf("%d/%d",&a.n, &a.d);

        printf("Enter the fraction2 in the form x/y: ");
        scanf("%d/%d",&b.n, &b.d);

        c.n = multiply (a.n, b.n);
        c.d = multiply (a.d, b.d);

        printf("%d/%d * %d/%d = %d/%d\n",a.n, a.d, b.n, b.d, c.n,c.d);
}
```

Disadvantage

- Passing the individual members to a function is more tedious as the number of parameters increases.
- More memory space is required if more number of parameters are passed by value
- Takes more time to copy the values of actual parameters to formal parameters.
- This method is inefficient when the structure size is too large.

The first disadvantage can be overcome by passing the whole structure as parameter.

13.11.2 Sending the whole structure

Now, the question is "Is there any better way of writing the above program?" Yes, there is. The better way is to represent the fractional number using structure as shown below:

```
        typedef struct
        {
                int     n;              /* n is numerator */
                int     d;              /* d is denominator */
        } FRACTION;
```

Then, use three variables **x, y** and **z** of type FRACTION. Pass two variables **x** and **y** as parameters to the function. The function can multiply the given two fractions storing the result as shown below:

- Multiply the numerators and store the result as numerator
  i.e., z.n = x.n * y.n;
- Multiply denominators and store the result as denominator
  i.e., z.d = x.d * y.d;

The complete program is shown below:

Example 13.19: Program to simulate the multiplication of two fractions

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

```
typedef struct
{
        int     n;
        int     d;
} FRACTION;

FRACTION multiply(FRACTION x, FRACTION y)
{
        FRACTION z;

        z.n = x.n * y.n;
        z.d = x.d * y.d;

        return z;
}

void main()
{
        FRACTION a, b, c;
        printf("Enter fraction1 in the form x/y: ");
        scanf("%d/%d",&a.n, &a.d);

        printf("Enter the fraction2 in the form x/y: ");
        scanf("%d/%d",&b.n, &b.d);

        c = multiply (a, b);

        printf("%d/%d * %d/%d = %d/%d\n",a.n, a.d, b.n, b.d, c.n,c.d);
}
```

Note: The above program is most efficient way to multiply two fractions when compared with program given in example 13.18.

13.11.3 Passing structures through pointers

Instead of passing structures as parameters, we can also pass address of structures as parameters. In such case, the formal parameters should be declared as pointers. The program for multiplication of two fractions shown in previous section can be written exclusively by passing the addresses with three functions:

- read_fraction() – is used to read a fraction from the keyboard
- print_fraction() – is used to the print the result on the screen
- mul_fraction() – is used to multiply two fractions

Example 13.20: Function to read a fraction

```
void read_fraction(FRACTION *x)
{
        printf("Enter fraction in the form x/y: ");

        scanf("%d/%d",&x->n, &x->d);
}
```

Example 13.21: Function to multiply two fractions **x** and **y**

```
void mul_freaction(FRACTION *x, FRACTION *y, FRACTION *z)
{
        z->n = x->n * y->n;
        z->d = x->d * y->d;
}
```

Example 13.22: Function to print the product of two fractions

```
void print_fraction(FRACTION *x, FRACTION *y, FRACTION *z)
{
        printf("%d/%d * %d/%d = %d/%d\n",x->n, x->d, y->n, y->d, z->n, z->d);
}
```

The complete program to multiply two fractions is shown below:

Example 13.23: Program to simulate the multiplication of two fractions using pointers to structures

```
#include <stdio.h>

typedef struct
{
        int     n;
        int     d;
} FRACTION;

/* Include: Example 13.20: Function to read a fraction */
/* Include: Example 13.21: Function to multiply two fractions x and y  */
/* Include: Example 13.22: Function to print the product of two fractions */

void main()
{
        FRACTION a, b, c;

        printf("Enter the first fraction :");
        read_fraction(&a);

        printf("Enter the second fraction :");
        read_fraction(&b);
        mul_fraction (&a, &b, &c);

        print_fraction(&a, &b, &c);
}
```

13.11.4 Uses of structures

Now, we can summarize the use of structures as shown below:

- Structures are used to represent more complex data structures
- Related data items of dissimilar data types can be logically grouped under a common name and all the items can be accessed using a common name.
- Can be used to pass arguments so as to minimize the number of function arguments.
- When more than one data has to be returned from the function, then structures can be used.
- Extensively used in applications involving database management
- To make the program more readable.

13.12 Operation on complex numbers

Now, the question is "Is it possible to perform various operations on complex numbers using C?" The answer is no. This is because the various data types in C language are **int, float, char** and **double.** There is no special data type to represent or manipulate complex numbers in C.

Now, the question is, "How to represent complex numbers in C?" This is where structures come to help. Using structure we can represent and store complex numbers. Using structures, let us try to frame our own data type called **COMPLEX** which has two parts **real part** and **imaginary part**. and then perform various operations on these complex numbers. Thus a complex number can be represented using a structure as shown below:

```
typedef struct
{
        float   r;      /* real part */
        float   i;      /* imaginary part */
} COMPLEX ;
```

Now, to store two complex numbers we need to use two variables. This can be done by declaring two variables as shown below:

```
COMPLEX a, b;           /* a and b represent 2 complex variables */
```

13.12.1 Addition of two complex numbers

Now, let us see "How to add two complex numbers?" Given two complex numbers (x + yi) and (a + bi), in mathematics addition can be performed as shown below:

$$\begin{array}{r} x + yi \\ + \; a + bi \\ \hline (x+a) + (y+b)i \end{array}$$

Implementation in C: Observe the following points:

- Real part of result = Real part of 1st number + Real part of 2nd number
- Imaginary part of result = Imaginary part of 1st + Imaginary part of 2nd number.

So, if **a** and **b** are two complex numbers with **r** as the real part and **i** as imaginary parts respectively, then the resulting complex number **c** which is sum of **a** and **b** is given by

```
c.r = a.r + b.r;
c.i = a.i + b.i;
```

The function to add two complex numbers **a** and **b** is shown below:

Example 13.24: C function to add two complex numbers

```
COMPLEX add(COMPLEX a, COMPLEX b)
{
        COMPLEX c;

        c.r = a.r + b.r;
        c.i = a.i + b.i;

        return c;
}
```

13.12.2 Subtract one complex number from other

Now, let us see "How to subtract one complex number from the other?" Given two complex numbers (x + yi) and (a + bi), we can subtract (a + bi) from (x + yi) as shown below :

$$\begin{array}{r} x + yi \\ -\;(a + bi) \\ \hline (x-a) + (y-b)i \end{array}$$

Implementation in C: Observe the following points:

- real part of result = Real part of 1st number – Real part of 2nd number
- Imaginary part of result = Imaginary part of 1st – Imaginary part of 2nd number.

So, if **a** and **b** are two complex numbers, then the resulting complex number **c** is given by

```
c.r = a.r – b.r;
c.i = a.i – b.i;
```

The function to add two complex numbers **a** and **b** is shown below:

Example 13.25: C function to subtract one complex number from the other

```
COMPLEX subtract(COMPLEX a, COMPLEX b)
{
        COMPLEX c;

        c.r = a.r – b.r;
        c.i = a.i – b.i;

        return c;
}
```

13.12.3 Multiplying two complex numbers

Now, let us see "How to multiply two complex numbers?" Given two complex numbers (x + yi) and (a + bi), multiplication can be performed as shown below :

$$
\begin{aligned}
(x + yi)(a + bi) &= (xa + xbi + yai + ybi^2) \\
&= (xa - yb) + (xb + ya)i
\end{aligned}
$$

Implementation in C: Observe that real part is obtained by multiplying real parts of two complex numbers and subtracting the product of two imaginary parts. So, if **a** and **b** are two complex numbers, then the real part of complex number **c** is given by

```
c.r = a.r*b.r – a.i*b.i
```

Imaginary part is obtained by multiplying real part of 1st number with imaginary part of 2nd number and adding to the product of imaginary part of 1st number with real part of 2nd number i.e.,

```
c.i = a.r*b.i + a.i*b.r
```

The function to multiply two complex numbers **a** and **b** is shown below:

Example 13.26: C function to multiply two complex numbers

```
COMPLEX multiply (COMPLEX a, COMPLEX b)
{
        COMPLEX c;
```

```
	c.r = a.r*b.r – a.i*b.i;
	c.i = a.r*b.i + a.i*b.r;

	return c;
}
```

13.12.4 Dividing a complex number by the other

Now, let us see "How to divide a complex number by other complex number?" Given two complex numbers (x + yi) and (a + bi), let us divide (x + yi) by (a + bi). This can be done mathematically as shown below:

$$= \frac{(x + yi)}{(a + bi)} = \frac{(x + yi)(a - bi)}{(a + bi)(a - bi)} = \frac{xa - xbi + yai - ybi^2}{a^2 + b^2} = \frac{(xa + yb)}{a^2 + b^2} + \frac{(ya - xb)i}{a^2 + b^2}$$

So, the real part is obtained by adding the product of two real parts with the product of two imaginary parts and dividing the result by $a^2 + b^2$ which is sum of squares of real part and imaginary part of the dividend. If **a** and **b** are two complex numbers, then the real part of **c** (which is the result) is given by

```
c.r =  ( a.r*b.r + a.i*b.i ) / (b.r*b.r + b.i*b.i);
```

The imaginary part of the result is obtained by multiplying the imaginary part of 1st number with real part of 2nd number and then subtract the product of real part of 1st number and imaginary part of 2nd number. Finally, the whole result should be divided by $a^2 + b^2$. If **a** and **b** are two complex numbers, then the imaginary part of **c**(which is the result) is given by

```
c.i = ( a.i*b.r – a.r*b.i ) / (b.r*b.r + b.i*b.i);
```

Thus, the function to divide **a** by **b** and storing the result in **c** is shown below:

Example 13.27: C function to divide **a** by **b**

```
COMPLEX divide (COMPLEX a, COMPLEX b)
{
	COMPLEX c;

	c.r =  ( a.r*b.r + a.i*b.i ) / (b.r*b.r + b.i*b.i);
	c.i = ( a.i*b.r – a.r*b.i ) / (b.r*b.r + b.i*b.i);

	return c;
}
```

13.12.5 Reading a complex number

The function to read a complex number is shown below:

Example 13.28: C function to read a complex number

```
/* function to read a complex number */
COMPLEX read_complex()
{
        COMPLEX x;
        float a;

        printf("RPart = ");    scanf("%f",&a);
        x.r = a;

        printf("IPart = ");    scanf("%f",&a);
        x.i = a;

        return x;       /* Complex number */
}
```

Note: The function read_complex() reads a complex number and the function write_complex() is used to display a complex number. These two functions are self-explanatory and left as an exercise to the reader to understand the logic.

13.12.6 Writing a complex number

The function to display a complex number is shown below:

Example 13.29: C function to display a complex number

```
/* function to display a complex number */
void write_complex(COMPLEX a)
{
        if (a.i >=0)
                printf("%5.2f + %5.2f i\n",a.r, a.i);   /* To print +ve imaginary part */
        else
                printf("%5.2f %5.2f i \n",a.r, a,i);    /* To print –ve imaginary part */
}
```

The complete program to add, subtract, multiply and divide two complex numbers is shown below:

Example 13.30: C program to perform arithmetic operations on complex numbers

```
#include <stdio.h>
/* Structure definition of a complex number */
```

```
typedef struct
{
        float   r;      /* real part */
        float   i;      /* imaginary part */
} COMPLEX ;

/* Include: Example 13.24: C function to add two complex numbers */
/* Include: Example 13.25: function to subtract a complex number from the other */
/* Include: Example 13.26: C function to multiply two complex numbers */
/* Include: Example 13.27: C function to divide a by b */
/* Include: Example 13.28: C function to read a complex number */
/* Include: Example 13.29: C function to display a complex number */

void main()
{
        COMPLEX a;                /* First complex number */
        COMPLEX b;                /* Second complex number */
        COMPLEX c;                /* Resultant complex number */

        printf("Enter first complex number\n");
        a = read_complex();
        printf("Enter second complex number\n");
        b = read_complex();

        printf("First complex number = ");
        write_complex(a);

        printf("Second complex number = ");
        write_complex(b);

        c = add(a,b);
        printf("Sum of complex numbers = ");
        write_complex(c);

        c = subtract(a,b);
        printf("Difference = ");
        write_complex(c);

        c  = multiply(a,b);
        printf("Product of complex numbers = ");
        write_complex(c);

        c  = divide(a,b);
        printf("Division of complex numbers = ");
        write_complex(c);
}
```

Input

```
Enter first complex number
Rpart = 4,    Ipart = 2
 Enter the second complex no.
Rpart = 2, Ipart = 2

First number = 4.00 +  2.00i

Second number = 2.00 +  2.00i

Sum =  6.00 +  4.00i

 Difference =  2.00 +  0.00i

      Product =  4.00 + 12.00i

      Divsn result =  1.50 + -0.50i
```

13.13 Union and its definition

In this section, we shall see another concept called **union** which is derived from structures. Let us see "What is a union?"

Definition: A union is a derived data type like structure. So, union can also be treated as a collection of variables under a single name. All these variables may contain data items of similar or dissimilar data types. Each variable in the union is called a member or a field.

Now let us see "How union is declared and used in C?" The syntax, declaration and use of **union** is similar to the structures but its functionality is totally different. Thus, the general format (syntax) of a **union** definition is shown below:

```
union tag_name
{
        type1     member1;
        type2     member2;
        ……        ……
        ……        ……
} ;     Note: semicolon is must
```

Observe the following points while defining a union.

- **union** is the keyword which tells the compiler that a union is being defined.
- member1, member2,….. are called members of the union. They are also called fields of the union.
- The members are declared within curly braces. The members can be any of the data types such as **int**, **char**, **float** etc.
- There should be semicolon at the end of closing brace

Example 13.31: Consider the following definition and declaration:

```
typedef union          ⎫
{                      ⎪
        int     i;     ⎬ union definition
        double  d;     ⎪
        char    c;     ⎪ Note: For the union definition memory is not allocated
} ITEM;                ⎭

ITEM x;                } union declaration
                         Note: For the union declaration memory is allocated
```

Note that **ITEM** is the *derived data type* and the variable **x** is of type **ITEM**. The definition of a **union** does not allocate memory since the definition is not associated with any variable. But, in the declaration, the definition is associated with a variable **x.** So, the memory is allocated for the variable **x**.

Example 13.32: Another way of declaring the same variable x is shown below:

```
union item
{
        int      i;
        double  d;
        char     c;
} x;
```

Note: Since the variable **x** is associated with the definition, the memory is allocated.

Now, let us see *"How to reference or access members of the union?"* To access the members of **union**, the same syntax used to access various members of a structure can be used. For example, by using the dot operator ('.'), we can access various members of the union as shown below:

```
x.i,     x.d,     x.c
```

The variable **x** is associated with union can also be declared as a pointer as shown below:

```
union item *x;
```

For the pointer variables, the indirection operator * and dot operator or arrow/selection operator -> can be used to access the members. For example, using pointer variable **x**, the members can be accessed as shown below:

```
x->i    or      (*x).i
x->d    or      (*x).d
x->c    or      (*x).c
```

Let us consider some examples to understand the difference between structure and union.

Example 13.33: Program to access various members of a union

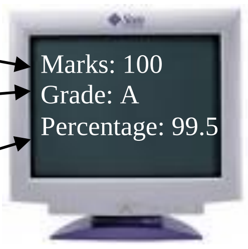
```
#include <stdio.h>
void main()
{
        typedef union
        {
                int     marks;
                char    grade;
                float   percentage;
        } STUDENT;
        STUDENT x;

        x.marks = 100;
        printf("Marks : %d\n",x.marks);

        x.grade = 'A';
        printf("Grade : %c\n",x.grade);

        x.percentage = 99.5;
        printf("Percentage: %f\n", x.percentage);
}
```

Output



```
Marks: 100
Grade: A
Percentage: 99.5
```

Observe the following points during execution of the above program:

- After executing the statement **x.marks = 100**, the member **marks** will hold the value 100 whereas other members such as **grade** and **percentage** contains garbage value and should not be accessed.
- After executing the statement **x.grade = 'A'**, the member **grade** will hold the value 'A' whereas other members **marks** and **percentage** contains garbage value and should not be accessed
- After executing the statement **x.percentage = 99.5**, the member **percentage** will hold the value 99.5 whereas other members *marks* and *grade* contains garbage value and should not be accessed.

Note: It is observed from the above output that only one member of union can hold a value at a time. It is not possible to access all the members simultaneously. So, the variable of type STUDENT can be treated as integer variable or char variable or float variable. Now, consider the same program with structure

Example 13.34: Program to access various members of a structure

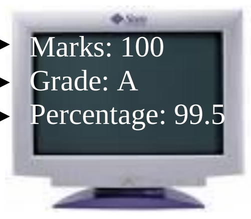
```
#include <stdio.h>

void main()
{
        typedef struct
        {
                int     marks;
                char    grade;
                float   percentage;
        } STUDENT;

        STUDENT x;
        x.marks = 100;
        x.grade = 'A';
        x.percentage = 99.5;

        printf("Grade : %c\n",x.grade);
        printf("Marks : %d\n",x.marks);
        printf("Percentage: %f\n", x.percentage);
}
```

Output



```
Marks: 100
Grade: A
Percentage: 99.5
```

Note: It is observed from the above output that the all the members of a structure can hold individual values at a time. It is possible to access all the members of a structure simultaneously.

Example 13.35: Now, let us see "What is the memory representation for the following structure and union?"

```
typedef union                     typedef struct
{                                 {
        char    c;                        char    c;
        int     i;                        int     i;
        float   f;                        float   f;
} DATA;                           } DATA ;

DATA x;                           DATA x;
```

Solution: The memory allocated by the compiler is large enough to hold the largest member of the union. Suppose, int occupy 2 bytes, char occupy 1 byte, float occupy 4 bytes. In this example, the member **f** occupies more memory (4 bytes) and so 4 bytes are allocated for the variable **x**. But, this allocated memory is shared by all the members namely **i**, **c** and **f** as shown below:

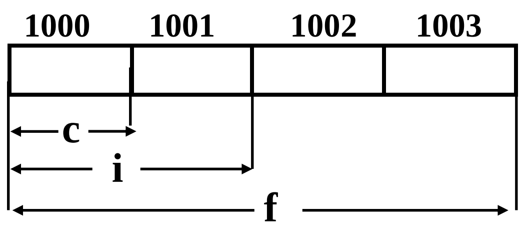


Fig. 13.13.a sharing of memory by union members

The memory representation for the structure with same fields is shown below:

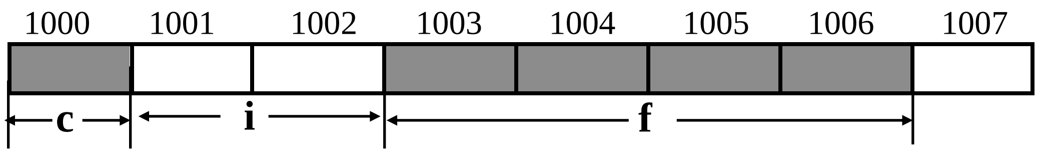


Fig. 13.13.b Unique memory locations for members of structure

Now, let us see "What are the differences between a structure and union?" consider the following examples:

Structure	Union
1. The keyword ***struct*** is used to define a structure	1. The keyword ***union*** is used to define a union.
2. When a variable is associated with a structure, the compiler allocates the memory for each member. The sizeof structure is greater than or equal to the sum of sizes of its members. The smaller members may end with unused slack bytes (see section 13.6 and 13.13.b)	2. When a variable is associated with a union, the compiler allocates the memory by considering the size of the largest member. So, size of union is equal to the size of largest member (Fig. 13.13.a)
3. Each member within a structure is assigned unique storage area (Fig. 13.13.b)	3. Memory allocated is shared by individual members of union (Fig. 13.13.a)
4. The address of each member will be in ascending order This indicates that memory for each member will start at different offset values (Fig 13.13.b)	4. The address is same for all the members of a union. This indicates that every member begins at offset values (Fig. 13.13.a)
5. Altering the value of a member will not affect other members of the	5. Altering the value of any of the member will alter other member
6. Individual members can be accessed at a time.	6. Only one member can be accessed at a time.
7. Several members of a structure can be initialized at once	7. Only the first member of a union can be initialized

13.14 Linear search in an array of structures

Now, let us *"Write a C program to maintain a record of **n** student details using an array of structures with four fields (Roll number, Name, Marks, and Grade). Assume* *appropriate data type for each field. Print the marks of the student, given the student name as input."*

The structure definition to store the student details such as Roll number, Name, Marks and Grade can be written as shown below:

```
typedef struct
{
        int     roll_num;
        char    name[20];
        int     marks;
        char    grade;
} STUDENT;
```

It is required to search for the student name in a record consisting of *n* students starting from the first record till the last record. This results in linear search (See section 9.6.1 for detailed design and example 9.17 in chapter 9). Now, the C function to search for the name in an array of structures is shown below:

Example 13.36: C function to implement linear search.

```
int   linear ( char   key[20],  STUDENT  a[],  int n)
{
        int   i;
        ..............................................................................
        for (i = 0; i < n; i++)
        {
                if ( strcmp(key, a[i].name) == 0 ) return i;    /* Successful search */
        }
        ..............................................................................
        return – 1;      /* Unsuccessful search */
}
```

The C program that uses the above function can be written as shown below:

Example 13.37: C program to search for a name in an array of structures

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

typedef struct
{
        int     roll_num;       /* Student roll number */
        char    name[20];       /* Name of the student */
        int     marks;          /* Student marks */
        char    grade;          /* Student grade */
} STUDENT;

/* Include: Example  13.36:  Function linear search */

void main()
{
        int             n;                      /* Number of names to be searched */
        STUDENT         a[20];                  /* An array to store student information*/
        char            key[20];                /* Name to be searched    */
        int             pos, i;                 /* Contains position of key if found:
                                                   -1 if not found                    */
        printf ("Enter the number of students\n");
        scanf ("%d", &n);

        printf("Enter the details of %d students\n", n);

        for (i = 0; i < n; i++)
        {
                printf("Details of %d student\n", i+1);
                printf("Name = ");              scanf(" %[^\n]", a[i].name);
                printf("Roll Number = ");       scanf("%d", &a[i].roll_num);
                printf("Marks = ");             scanf("%d", &a[i].marks);
                printf("Grade = ");             scanf("%c", &a[i].grade);
        }

        printf ("Enter the name to be searched\n");
        gets(key);
        ..........................................................................
        pos = linear_search (key, a, n);
        ..........................................................................
        if (pos == -1)
                printf ("Name not found \n");

        else
                printf ("Marks of %s = %d\n", a[pos].name, a[pos].marks);
}
```

Output

```
Enter the number of students 5
Enter the details of 5 students
```

```
student  Name          Roll Number   Marks         Grade
KRISHNA                100           95            A
MITHIL                 101           99            A
MONA                   102           99            A
SONU                   103           90            B
RAMA                   104           80            C

Enter the name to be searched : MITHIL
Marks of MITHIL = 99
Enter the name to be searched: MAHESH
Name not found
```

Exercises:

1) What is a derived data type?
2) What is type definition? What are user-defined data types?
3) What is the location of **typedef** definitions? What are the advantages of **typedef?**
4) What is a structure? What is the syntax of a structure? How to declare a structure?
5) What is a tagged structure? Explain with example.
6) What is a structure without tag? Explain with example.
7) What is a type-defined structure? Explain with example.
8) How to declare type-defined structure variables? How are structures initialized?
9) How structure members are accessed?
10) Write a program to simulate the multiplication of two fractions using structures
11) What is the size of the structure? What are slack bytes?
12) What is pointer to a structure? Using pointer to a structure how to access the members of structure
13) What is a nested structure? Write a program to arrange student's information in ascending order using nested structures.
14) Is it possible to use arrays within structures? Write a function search for a string using binary search using array of structures
15) Is it possible to use pointers within structures Explain with example
16) Explain the concept of array of pointers with a program

17) What are the various methods of passing structures to functions? Explain with examples

18) Write a program to multiply two fractional numbers using structures by passing by address

19) What are the uses of structures?

20) Write a program to perform arithmetic operations on two complex numbers

21) What is a union explain with example. How to reference or access members of the union?

22) What are the differences between structures and unions

23) How the members of a structure can be accessed? Explain when '.' is used and when '->' or * operators are used to access the fields within the structure?

24) What is a structure? How does a structure differ from an array? How are structure members assigned values and are accessed? Explain.

25) Explain array of structures with example

26) Is it possible to have structures within structures? Explain with an example

27) Is it possible to return the value of a structure using return statement? If the values of two or more structures is needed in the calling function, what to do? Explain with an example.

28) What is a union? How is it different from structure? With a suitable example show how union is declared and used in C.

29) Write a C program to arrange the student record based on increasing order of roll numbers, increasing order of marks, increasing order of their age. Assume the student record contains the following fields: name, age, branch, marks, roll number and address.