10 Structures

- Why Use Structures
  - Declaring a Structure
  - Accessing Structure Elements
  - How Structure Elements are Stored
- Array of Structures
- Additional Features of Structures
- Uses of Structures
- Summary
- Exercise

Which mechanic is good enough who knows how to repair only one type of vehicle? None. Same thing is true about C language. It wouldn't have been so popular had it been able to handle only all **int**s, or all **float**s or all **char**s at a time. In fact when we handle real world data, we don't usually deal with little atoms of information by themselves—things like integers, characters and such. Instead we deal with entities that are collections of things, each thing having its own attributes, just as the entity we call a 'book' is a collection of things such as title, author, call number, publisher, number of pages, date of publication, etc. As you can see all this data is dissimilar, for example author is a string, whereas number of pages is an integer. For dealing with such collections, C provides a data type called 'structure'. A structure gathers together, different atoms of information that comprise a given entity. And structure is the topic of this chapter.

Why Use Structures

We have seen earlier how ordinary variables can hold one piece of information and how arrays can hold a number of pieces of information of the same data type. These two data types can handle a great variety of situations. But quite often we deal with entities that are collection of dissimilar data types.

For example, suppose you want to store data about a book. You might want to store its name (a string), its price (a float) and number of pages in it (an int). If data about say 3 such books is to be stored, then we can follow two approaches:

(a) Construct individual arrays, one for storing names, another for storing prices and still another for storing number of pages.

(b) Use a structure variable.

Let us examine these two approaches one by one. For the sake of programming convenience assume that the names of books would

be single character long. Let us begin with a program that uses arrays.

```
main( )
{
      char  name[3] ;
      float  price[3] ;
      int  pages[3], i ;

      printf ( "\nEnter names, prices and no. of pages of 3 books\n" ) ;

      for ( i = 0 ; i <= 2 ; i++ )
            scanf ( "%c %f %d", &name[i], &price[i], &pages[i] );

      printf ( "\nAnd this is what you entered\n" ) ;
      for ( i = 0 ; i <= 2 ; i++ )
            printf ( "%c %f %d\n", name[i], price[i], pages[i] );
}
```

And here is the sample run...

```
Enter names, prices and no. of pages of 3 books
A  100.00  354
C  256.50  682
F  233.70  512

And this is what you entered
A  100.000000  354
C  256.500000  682
F  233.700000  512
```

This approach no doubt allows you to store names, prices and number of pages. But as you must have realized, it is an unwieldy approach that obscures the fact that you are dealing with a group of characteristics related to a single entity—the book.

The program becomes more difficult to handle as the number of items relating to the book go on increasing. For example, we would be required to use a number of arrays, if we also decide to store name of the publisher, date of purchase of book, etc. To solve this problem, C provides a special data type—the structure.

A structure contains a number of data types grouped together. These data types may or may not be of the same type. The following example illustrates the use of this data type.

```
main( )
{
    struct book
    {
        char  name ;
        float  price ;
        int  pages ;
    } ;
    struct book  b1, b2, b3 ;

    printf ( "\nEnter names, prices & no. of pages of 3 books\n" ) ;
    scanf ( "%c %f %d", &b1.name, &b1.price, &b1.pages ) ;
    scanf ( "%c %f %d", &b2.name, &b2.price, &b2.pages ) ;
    scanf ( "%c %f %d", &b3.name, &b3.price, &b3.pages ) ;

    printf ( "\nAnd this is what you entered" ) ;
    printf ( "\n%c %f %d", b1.name, b1.price, b1.pages ) ;
    printf ( "\n%c %f %d", b2.name, b2.price, b2.pages ) ;
    printf ( "\n%c %f %d", b3.name, b3.price, b3.pages ) ;
}
```

And here is the output...

```
Enter names, prices and no. of pages of 3 books
A  100.00  354
C  256.50  682
F  233.70  512
```

```
And this is what you entered
A  100.000000  354
C  256.500000  682
F  233.700000  512
```

This program demonstrates two fundamental aspects of structures:

(a) declaration of a structure
(b) accessing of structure elements

Let us now look at these concepts one by one.

Declaring a Structure

In our example program, the following statement declares the structure type:

```
struct book
{
     char  name ;
     float  price ;
     int  pages ;
} ;
```

This statement defines a new data type called **struct book**. Each variable of this data type will consist of a character variable called **name**, a float variable called **price** and an integer variable called **pages**. The general form of a structure declaration statement is given below:

```
struct <structure name>
{
     structure element 1 ;
     structure element 2 ;
     structure element 3 ;
     ......
     ......
```

```
} ;
```

Once the new structure data type has been defined one or more variables can be declared to be of that type. For example the variables **b1**, **b2**, **b3** can be declared to be of the type **struct book**, as,

```
struct book  b1, b2, b3 ;
```

This statement sets aside space in memory. It makes available space to hold all the elements in the structure—in this case, 7 bytes—one for **name**, four for **price** and two for **pages**. These bytes are always in adjacent memory locations.

If we so desire, we can combine the declaration of the structure type and the structure variables in one statement.

For example,

```
struct book
{
      char  name ;
      float  price ;
      int  pages ;
} ;
struct book  b1, b2, b3 ;
```

is same as...

```
struct book
{
      char  name ;
      float  price ;
      int  pages ;
} b1, b2, b3 ;
or even...
struct
```

```
{
     char  name ;
     float  price ;
     int  pages ;
} b1, b2, b3 ;
```

Like primary variables and arrays, structure variables can also be initialized where they are declared. The format used is quite similar to that used to initiate arrays.

```
struct book
{
     char  name[10] ;
     float  price ;
     int  pages ;
} ;
struct book  b1 = { "Basic", 130.00, 550 } ;
struct book  b2 = { "Physics", 150.80, 800 } ;
```

Note the following points while declaring a structure type:

(a) The closing brace in the structure type declaration must be followed by a semicolon.
(b) It is important to understand that a structure type declaration does not tell the compiler to reserve any space in memory. All a structure declaration does is, it defines the 'form' of the structure.
(c) Usually structure type declaration appears at the top of the source code file, before any variables or functions are defined. In very large programs they are usually put in a separate header file, and the file is included (using the preprocessor directive #include) in whichever program we want to use this structure type.

Accessing Structure Elements

Having declared the structure type and the structure variables, let us see how the elements of the structure can be accessed.

In arrays we can access individual elements of an array using a subscript. Structures use a different scheme. They use a dot (.) operator. So to refer to **pages** of the structure defined in our sample program we have to use,

```
b1.pages
```

Similarly, to refer to **price** we would use,

```
b1.price
```

Note that before the dot there must always be a structure variable and after the dot there must always be a structure element.

How Structure Elements are Stored

Whatever be the elements of a structure, they are always stored in contiguous memory locations. The following program would illustrate this:

```
/* Memory map of structure elements */
main( )
{
    struct book
    {
        char  name ;
        float  price ;
        int  pages ;
    } ;
    struct book  b1 = { 'B', 130.00, 550 } ;

    printf ( "\nAddress of name = %u", &b1.name ) ;
```

```
    printf ( "\nAddress of price = %u", &b1.price ) ;
    printf ( "\nAddress of pages = %u", &b1.pages ) ;
}
```

Here is the output of the program...

```
Address of name = 65518
Address of price = 65519
Address of pages = 65523
```

Actually the structure elements are stored in memory as shown in the Figure 10.1.

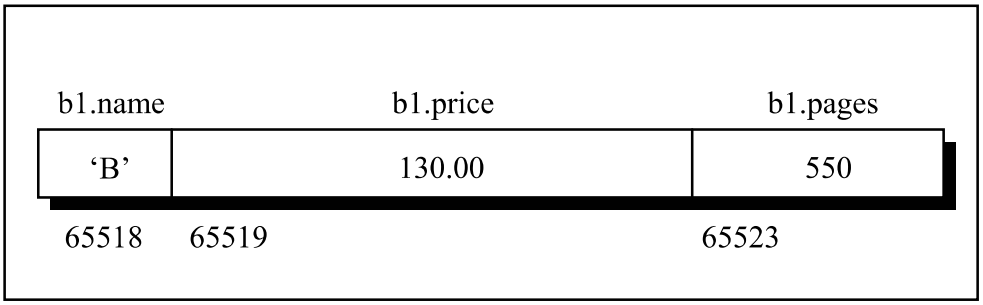

Figure 10.1

Array of Structures

Our sample program showing usage of structure is rather simple minded. All it does is, it receives values into various structure elements and output these values. But that's all we intended to do anyway... show how structure types are created, how structure variables are declared and how individual elements of a structure variable are referenced.

In our sample program, to store data of 100 books we would be required to use 100 different structure variables from **b1** to **b100**, which is definitely not very convenient. A better approach would be to use an array of structures. Following program shows how to use an array of structures.

```
/* Usage of an array of structures */
main( )
{
      struct book
      {
            char  name ;
            float  price ;
            int  pages ;
      } ;

      struct book  b[100] ;
      int  i ;

      for ( i = 0 ; i <= 99 ; i++ )
      {
            printf ( "\nEnter name, price and pages " ) ;
            scanf ( "%c %f %d", &b[i].name, &b[i].price, &b[i].pages ) ;
      }

      for ( i = 0 ; i <= 99 ; i++ )
            printf ( "\n%c %f %d", b[i].name, b[i].price, b[i].pages ) ;
}

linkfloat( )
{
      float a = 0, *b ;
      b = &a ;  /* cause emulator to be linked */
      a = *b ;    /* suppress the warning - variable not used */
}
```

Now a few comments about the program:

(a) Notice how the array of structures is declared...

```
struct book b[100] ;
```

This provides space in memory for 100 structures of the type **struct book**.

(b) The syntax we use to reference each element of the array **b** is similar to the syntax used for arrays of **int**s and **char**s. For example, we refer to zeroth book's price as **b[0].price**. Similarly, we refer first book's pages as **b[1].pages**.

(c) It should be appreciated what careful thought Dennis Ritchie has put into C language. He first defined array as a collection of similar elements; then realized that dissimilar data types that are often found in real life cannot be handled using arrays, therefore created a new data type called structure. But even using structures programming convenience could not be achieved, because a lot of variables (**b1** to **b100** for storing data about hundred books) needed to be handled. Therefore he allowed us to create an array of structures; an array of similar data types which themselves are a collection of dissimilar data types. Hats off to the genius!

(d) In an array of structures all elements of the array are stored in adjacent memory locations. Since each element of this array is a structure, and since all structure elements are always stored in adjacent locations you can very well visualise the arrangement of array of structures in memory. In our example, **b[0]**'s **name**, **price** and **pages** in memory would be immediately followed by **b[1]**'s **name**, **price** and **pages**, and so on.

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

scanf( ) and **atof( )**. There are some cases in which the reference to the **float** is a bit obscure and the compiler does not detect the need for the emulator. The most common is using **scanf( )** to read a **float** in an array of structures as shown in our program.

How can we force the formats to be linked? That's where the **linkfloat( )** function comes in. It forces linking of the floating-point emulator into an application. There is no need to call this function, just define it anywhere in your program.

Additional Features of Structures

Let us now explore the intricacies of structures with a view of programming convenience. We would highlight these intricacies with suitable examples:

(a) The values of a structure variable can be assigned to another structure variable of the same type using the assignment operator. It is not necessary to copy the structure elements piece-meal. Obviously, programmers prefer assignment to piece-meal copying. This is shown in the following example.

```
main( )
{
    struct employee
    {
        char  name[10] ;
        int  age ;
        float  salary ;
    } ;
    struct employee  e1 = { "Sanjay", 30, 5500.50 } ;
    struct employee  e2, e3 ;

    /* piece-meal copying */
    strcpy ( e2.name, e1.name ) ;
    e2.age = e1.age ;
```

```
    e2.salary = e1.salary ;

    /* copying all elements at one go */
    e3 = e2 ;

    printf ( "\n%s %d %f", e1.name, e1.age, e1.salary ) ;
    printf ( "\n%s %d %f", e2.name, e2.age, e2.salary ) ;
    printf ( "\n%s %d %f", e3.name, e3.age, e3.salary ) ;
}
```

The output of the program would be...

```
Sanjay 30 5500.500000
Sanjay 30 5500.500000
Sanjay 30 5500.500000
```

Ability to copy the contents of all structure elements of one variable into the corresponding elements of another structure variable is rather surprising, since C does not allow assigning the contents of one array to another just by equating the two. As we saw earlier, for copying arrays we have to copy the contents of the array element by element.

This copying of all structure elements at one go has been possible only because the structure elements are stored in contiguous memory locations. Had this not been so, we would have been required to copy structure variables element by element. And who knows, had this been so, structures would not have become popular at all.

(b) One structure can be nested within another structure. Using this facility complex data types can be created. The following program shows nested structures at work.

```
main( )
{
    struct address
```

```
    {
        char  phone[15] ;
        char  city[25] ;
        int  pin ;
    } ;

    struct emp
    {
        char  name[25] ;
        struct address  a ;
    } ;
    struct emp  e = { "jeru", "531046", "nagpur", 10 };

    printf ( "\nname = %s phone = %s", e.name, e.a.phone ) ;
    printf ( "\ncity = %s pin = %d", e.a.city, e.a.pin ) ;
}
```

And here is the output...

```
name = jeru phone = 531046
city = nagpur pin = 10
```

Notice the method used to access the element of a structure that is part of another structure. For this the dot operator is used twice, as in the expression,

```
e.a.pin  or  e.a.city
```

Of course, the nesting process need not stop at this level. We can nest a structure within a structure, within another structure, which is in still another structure and so on... till the time we can comprehend the structure ourselves. Such construction however gives rise to variable names that can be surprisingly self descriptive, for example:

```
maruti.engine.bolt.large.qty
```

This clearly signifies that we are referring to the quantity of large sized bolts that fit on an engine of a maruti car.

(c) Like an ordinary variable, a structure variable can also be passed to a function. We may either pass individual structure elements or the entire structure variable at one go. Let us examine both the approaches one by one using suitable programs.

```
/* Passing individual structure elements */
main( )
{
    struct book
    {
        char  name[25] ;
        char  author[25] ;
        int  callno ;
    } ;
    struct book b1 = { "Let us C", "YPK", 101 } ;

    display ( b1.name, b1.author, b1.callno ) ;
}

display ( char  *s, char  *t, int  n )
{
    printf ( "\n%s %s %d", s, t, n ) ;
}
```

And here is the output...

```
Let us C YPK 101
```

Observe that in the declaration of the structure, **name** and **author** have been declared as arrays. Therefore, when we call the function **display( )** using,

```
display ( b1.name, b1.author, b1.callno ) ;
```

we are passing the base addresses of the arrays **name** and **author**, but the value stored in **callno**. Thus, this is a mixed call—a call by reference as well as a call by value.

It can be immediately realized that to pass individual elements would become more tedious as the number of structure elements go on increasing. A better way would be to pass the entire structure variable at a time. This method is shown in the following program.

```
struct book
{
    char  name[25] ;
    char  author[25] ;
    int  callno ;
} ;

main( )
{
    struct book  b1 = { "Let us C", "YPK", 101 } ;
    display ( b1 ) ;
}

display ( struct book  b )
{
    printf ( "\n%s %s %d", b.name, b.author, b.callno ) ;
}
```

And here is the output...

```
Let us C YPK 101
```

Note that here the calling of function **display( )** becomes quite compact,

```
display ( b1 ) ;
```

Having collected what is being passed to the **display( )** function, the question comes, how do we define the formal arguments in the function. We cannot say,

```
struct book b1 ;
```

because the data type **struct book** is not known to the function **display( )**. Therefore, it becomes necessary to define the structure type **struct book** outside **main( )**, so that it becomes known to all functions in the program.

(d) The way we can have a pointer pointing to an **int**, or a pointer pointing to a **char**, similarly we can have a pointer pointing to a **struct**. Such pointers are known as 'structure pointers'.

Let us look at a program that demonstrates the usage of a structure pointer.

```
main( )
{
    struct book
    {
        char  name[25] ;
        char  author[25] ;
        int  callno ;
    } ;
    struct book  b1 = { "Let us C", "YPK", 101 } ;
    struct book  *ptr ;

    ptr = &b1 ;
    printf ( "\n%s %s %d", b1.name, b1.author, b1.callno ) ;
    printf ( "\n%s %s %d", ptr->name, ptr->author, ptr->callno ) ;
}
```

The first **printf( )** is as usual. The second **printf( )** however is peculiar. We can't use **ptr.name** or **ptr.callno** because **ptr** is not a structure variable but a pointer to a structure, and the dot

operator requires a structure variable on its left. In such cases C provides an operator **->**, called an arrow operator to refer to the structure elements. Remember that on the left hand side of the **'.'** structure operator, there must always be a structure variable, whereas on the left hand side of the **'->'** operator there must always be a pointer to a structure. The arrangement of the structure variable and pointer to structure in memory is shown in the Figure 10.2.

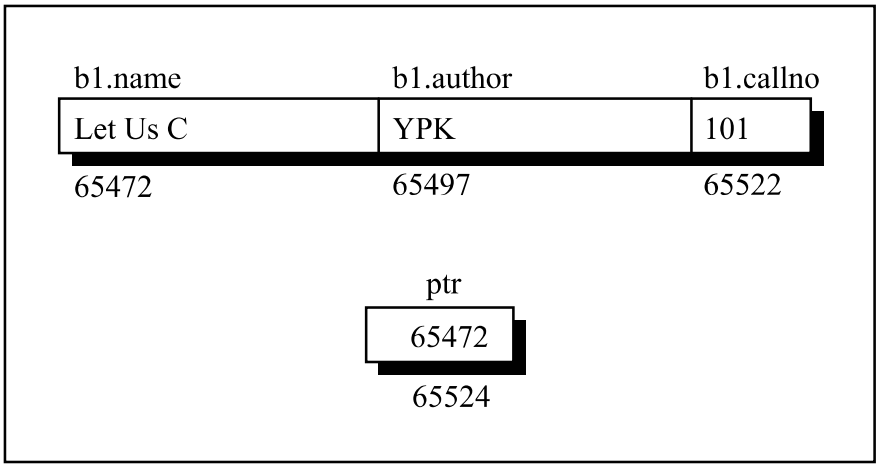

Figure 10.2

Can we not pass the address of a structure variable to a function? We can. The following program demonstrates this.

```
/* Passing address of a structure variable */
struct book
{
    char  name[25] ;
    char  author[25] ;
    int  callno ;
} ;

main( )
{
    struct book  b1 = { "Let us C", "YPK", 101 } ;
    display ( &b1 ) ;
```

```
}

display ( struct book  *b )
{
    printf ( "\n%s %s %d", b->name, b->author, b->callno ) ;
}
```

And here is the output...

```
Let us C YPK 101
```

Again note that to access the structure elements using pointer to a structure we have to use the '**->**' operator.

Also, the structure **struct book** should be declared outside **main( )** such that this data type is available to **display( )** while declaring pointer to the structure.

(e) Consider the following code snippet:

```
struct emp
{
    int a ;
    char ch ;
    float s ;
} ;
struct emp e ;
printf ( "%u %u %u", &e.a, &e.ch, &e.s ) ;
```

If we execute this program using TC/TC++ compiler we get the addresses as:

```
65518  65520  65521
```

As expected, in memory the **char** begins immediately after the **int** and **float** begins immediately after the **char**.

However, if we run the same program using VC++ compiler then the output turns out to be:

```
1245044  1245048  1245052
```

It can be observed from this output that the **float** doesn't get stored immediately after the **char**. In fact there is a hole of three bytes after the **char**. Let us understand the reason for this. VC++ is a 32-bit compiler targeted to generate code for a 32-bit microprocessor. The architecture of this microprocessor is such that it is able to fetch the data that is present at an address, which is a multiple of four much faster than the data present at any other address. Hence the VC++ compiler aligns every element of a structure at an address that is multiple of four. That's the reason why there were three holes created between the **char** and the **float**.

However, some programs need to exercise precise control over the memory areas where data is placed. For example, suppose we wish to read the contents of the boot sector (first sector on the floppy/hard disk) into a structure. For this the byte arrangement of the structure elements must match the arrangement of various fields in the boot sector of the disk. The **#pragma pack** directive offers a way to fulfill this requirement. This directive specifies packing alignment for structure members. The pragma takes effect at the first structure declaration after the pragma is seen. Turbo C/C++ compiler doesn't support this feature, VC++ compiler does. The following code shows how to use this directive.

```
#pragma pack(1)
struct emp
{
    int a ;
    char ch ;
    float s ;
} ;
```

```
#pragma pack( )

struct emp e ;
printf ( "%u %u %u", &e.a, &e.ch, &e.s ) ;
```

Here, **#pragma pack (1)** lets each structure element to begin on a 1-byte boundary as justified by the output of the program given below:

```
1245044  1245048  1245049
```

Uses of Structures

Where are structures useful? The immediate application that comes to the mind is Database Management. That is, to maintain data about employees in an organization, books in a library, items in a store, financial accounting transactions in a company etc. But mind you, use of structures stretches much beyond database management. They can be used for a variety of purposes like:

(a) Changing the size of the cursor
(b) Clearing the contents of the screen
(c) Placing the cursor at an appropriate position on screen
(d) Drawing any graphics shape on the screen
(e) Receiving a key from the keyboard
(f) Checking the memory size of the computer
(g) Finding out the list of equipment attached to the computer
(h) Formatting a floppy
(i) Hiding a file from the directory
(j) Displaying the directory of a disk
(k) Sending the output to printer
(l) Interacting with the mouse

And that is certainly a very impressive list! At least impressive enough to make you realize how important a data type a structure is and to be thorough with it if you intend to program any of the

above applications. Some of these applications would be discussed in Chapters 16 to 19.

Summary

(a) A structure is usually used when we wish to store dissimilar data together.
(b) Structure elements can be accessed through a structure variable using a dot (.) operator.
(c) Structure elements can be accessed through a pointer to a structure using the arrow (->) operator.
(d) All elements of one structure variable can be assigned to another structure variable using the assignment (=) operator.
(e) It is possible to pass a structure variable to a function either by value or by address.
(f) It is possible to create an array of structures.

Exercise

[A] What would be the output of the following programs:

(a)
```
main( )
{
    struct gospel
    {
        int  num ;
        char  mess1[50] ;
        char  mess2[50] ;
    } m ;

    m.num = 1 ;
    strcpy ( m.mess1, "If all that you have is hammer" ) ;
    strcpy ( m.mess2, "Everything looks like a nail" ) ;

    /* assume that the strucure is located at address 1004 */
    printf ( "\n%u %u %u", &m.num, m.mess1, m.mess2 ) ;
}
```

```
(b)  struct gospel
     {
         int  num ;
         char  mess1[50] ;
         char  mess2[50] ;
     } m1 =  { 2, "If you are driven by success",
                    "make sure that it is a quality drive"
               } ;
     main( )
     {
         struct gospel  m2, m3 ;
         m2 = m1 ;
         m3 = m2 ;
         printf ( "\n%d %s %s", m1.num, m2.mess1, m3.mess2 ) ;
     }
```

[B] Point out the errors, if any, in the following programs:

```
(a)  main( )
     {
         struct employee
         {
               char  name[25] ;
               int  age ;
               float  bs ;
         } ;
         struct employee  e ;
         strcpy ( e.name, "Hacker" ) ;
         age = 25 ;
         printf ( "\n%s %d", e.name, age ) ;
     }
(b)  main( )
     {
         struct
         {
               char  name[25] ;
```

```
        char  language[10] ;
    } ;
    struct employee  e = { "Hacker", "C" } ;
    printf ( "\n%s %d", e.name, e.language ) ;
}
```

(c)
```
struct virus
{

    char  signature[25] ;
    char  status[20] ;
    int  size ;
} v[2] = {
            "Yankee Doodle", "Deadly", 1813,
            "Dark Avenger",  "Killer", 1795
        } ;
main( )
{
    int  i ;
    for ( i = 0 ; i <=1 ; i++ )
        printf ( "\n%s %s", v.signature, v.status ) ;
}
```

(d)
```
struct s
{
    char  ch ;
    int  i ;
    float  a ;
} ;
main( )
{
    struct s  var = { 'C', 100, 12.55 } ;
    f ( var ) ;
    g ( &var ) ;
}
f ( struct s  v )
{
    printf ( "\n%c %d %f", v -> ch, v -> i, v -> a ) ;
}
```

```
g ( struct s  *v )
{
    printf ( "\n%c %d %f", v.ch, v.i, v.a ) ;
}
```

(e)
```
struct s
{
    int  i ;
    struct s  *p ;
} ;
main( )
{
    struct s  var1, var2 ;

    var1.i = 100 ;
    var2.i = 200 ;
    var1.p = &var2 ;
    var2.p = &var1 ;
    printf ( "\n%d %d", var1.p -> i, var2.p -> i ) ;
}
```

[C] Answer the following:

(a) Ten floats are to be stored in memory. What would you prefer, an array or a structure?

(b) Given the statement,

```
maruti.engine.bolts = 25 ;
```

which of the following is True?

1. structure bolts is nested within structure engine
2. structure engine is nested within structure maruti
3. structure maruti is nested within structure engine
4. structure maruti is nested within structure bolts

(c) State True or False:

1. All structure elements are stored in contiguous memory locations.

2. An array should be used to store dissimilar elements, and a structure to store similar elements.

3. In an array of structures, not only are all structures stored in contiguous memory locations, but the elements of individual structures are also stored in contiguous locations.

(d)
```
struct time
{
    int  hours ;
    int  minutes ;
    int  seconds ;
} t ;
struct time  *tt ;
tt = &t ;
```

Looking at the above declarations, which of the following refers to **seconds** correctly:

1. tt.seconds
2. (*tt).seconds
3. time.t
4. tt -> seconds

[D] Attempt the following:

(a) Create a structure to specify data on students given below:

Roll number, Name, Department, Course, Year of joining

Assume that there are not more than 450 students in the collage.

(a) Write a function to print names of all students who joined in a particular year.
(b) Write a function to print the data of a student whose roll number is given.

(b) Create a structure to specify data of customers in a bank. The data to be stored is: Account number, Name, Balance in account. Assume maximum of 200 customers in the bank.
   (a) Write a function to print the Account number and name of each customer with balance below Rs. 100.

   (b) If a customer request for withdrawal or deposit, it is given in the form:

   Acct. no, amount, code (1 for deposit, 0 for withdrawal)

   Write a program to give a message, "The balance is insufficient for the specified withdrawal".

(c) An automobile company has serial number for engine parts starting from AA0 to FF9. The other characteristics of parts to be specified in a structure are: Year of manufacture, material and quantity manufactured.
   (a) Specify a structure to store information corresponding to a part.
   (b) Write a program to retrieve information on parts with serial numbers between BB1 and CC6.

(d) A record contains name of cricketer, his age, number of test matches that he has played and the average runs that he has scored in each test match. Create an array of structure to hold records of 20 such cricketer and then write a program to read these records and arrange them in ascending order by average runs. Use the **qusort( )** standard library function.

(e) There is a structure called **employee** that holds information like employee code, name, date of joining. Write a program to create an array of the structure and enter some data into it. Then ask the user to enter current date. Display the names of those employees whose tenure is 3 or more than 3 years according to the given current date.

(f) Write a menu driven program that depicts the working of a library. The menu options should be:

1. Add book information
2. Display book information
3. List all books of given author
4. List the title of specified book
5. List the count of books in the library
6. List the books in the order of accession number
7. Exit

Create a structure called **library** to hold accession number, title of the book, author name, price of the book, and flag indicating whether book is issued or not.

(g) Write a program that compares two given dates. To store date use structure say **date** that contains three members namely date, month and year. If the dates are equal then display message as "Equal" otherwise "Unequal".

(h) Linked list is a very common data structure often used to store similar data in memory. While the elements of an array occupy contiguous memory locations, those of a linked list are not constrained to be stored in adjacent location. The individual elements are stored "somewhere" in memory, rather like a family dispersed, but still bound together. The order of the elements is maintained by explicit links between them. Thus, a linked list is a collection of elements called nodes, each of which stores two item of information—an element of the list, and a link, i.e., a pointer or an address that indicates explicitly the location of the node containing the successor of this list element.

Write a program to build a linked list by adding new nodes at the beginning, at the end or in the middle of the linked list. Also write a function **display( )** which display all the nodes present in the linked list.

(i) A stack is a data structure in which addition of new element or deletion of existing element always takes place at the same

end. This end is often known as ‘top’ of stack. This situation can be compared to a stack of plates in a cafeteria where every new plate taken off the stack is also from the ‘top’ of the stack. There are several application where stack can be put to use. For example, recursion, keeping track of function calls, evaluation of expressions, etc. Write a program to implement a stack using a linked list.

(j) Unlike a stack, in a queue the addition of new element takes place at the end (called ‘rear’ of queue) whereas deletion takes place at the other end (called ‘front’ of queue). Write a program to implement a queue using a linked list.