Chapter-10

Structure and Union

10.1 Introduction to Structure

A structure is a collection of variables under a single name. The variables may be of different types, and each has a name which is used to select it from the structure. The variables are called members of the structure. A structure is a convenient way of grouping several pieces of related information together. A structure can be defined as a new named type or user defined data type, thus extending the number of available types. It can use other structures, arrays or pointers as its members.

An array can be used only to represent a group of data items that belong to the same data type, such as `int` or `float`. However, if we want to represent a collection of data items of different types using a single name, array can not be used. In such situation, structure is used, which is a method for packing/encapulating data of different types. A structure is a convenient tool for handling a group of logically related data items. For example: `name`, `roll`, `fee`, `marks`, `address`, `gender`, and `phone` are related information of a student. To store information about a student, we would require to store the `name` of student which is array of characters, `roll` of student which is integer type of data, `fee` of student which is `float` type of data, `marks` of student which is `float` type of data, `address` of student which is array of characters, `gender` of student which is `M` or `F` (i.e. character type) and `phone` of student which is integer type or array of characters. These attributes of students are grouped into a single entity, `student`. Here, `student` is known as structure which organizes different data items in a more meaningful way.

10.2 Defining a Structure

Syntax:

```
struct   structure_name
                {
                 data_type member_variable1;
                 data_type member_variable2;
                 . . . . .
                 . . . .
                 data_type member_variablen;
                };
```

Once `structure_name` is declared as new data type, then variables of that type can be declared as

```
struct structure_name structure_variable;
```

Examples for Structures:

a) Let us create a structure named `student` that has `name`, `roll`, `marks` and `remarks` as members.

```
struct student
    {
     char name[20];
     int roll;
     float marks;
     char remarks;
    };
```

Here, `student` is structure name and its members are `name`, `roll`, `marks` and `remarks`. Here, `student` is new data type and various variables of type `struct student` can be declared as

```
struct student st;
```

Here, `struct student` is new data type and st is variable of that type. The multiple variables are declared as

```
struct student st1, st2, st2;
```

Each variable of structure has its own copy of member variables. The member variables are accessed using dot (.) operator. For example, `st1.name` is member variable, `name`, of `st1` structure variable and `st2.name` is the member variable, `name`, of second structure variable `st2`. Similarly, `st1.roll` is the member, `roll`, of first structure variable `st1` and `st2.roll` is the member, `roll`, of second structure variable `st2`.

The structure definition and it's variable declaration can be combined as following

```
struct student
    {
    char name[20];
    int roll;
    float marks;
    char remarks;
    }st1,st2;
```

or

```
struct
{
char name[20];
int roll;
float marks;
char remarks;
}st1,st2;
```

b) Let us consider another structure `date`, which is collection of `day`, `month` and `year`. It is defined as

```
struct date
    {
      int day;
      int month;
      int year;
    };
```

Then different objects or variables of type `date` can be declared as

```
date dateOfBirth;
date d1, d2, d3;
```

In this case also, the structure definition and variable declaration can be combined as following

```
struct date
 {
   int day;
   int month;
   int year;
 }dateOfBirth;
```

or

```
struct
 {
 int day;
 int month;
  int year;
  }dateOfBirth;
```

c) `book` structure which contains members: `title, author, pages and price.`

```
struct book
  {
   char title[15];
   char author[20];
   int pages;
   float price;
   }b1,b2;
```

d) `employee` structure which contains attributes `empID, empName, salary, allowance, sex` and `age` of employee.

```
struct employee
  {
  int empID;
  char empName[20];
  float salary;
  float allowance;
  char sex;
  int age;
  };
```

e) `account` structure which organizes `account_no, account_type` and `balance` together.

```
struct account
  {
   int account_no;
   char account_type[10];
   float balance;
  };
```

f) `department` structure which contains `deptID`, `deptName`, `location` and `noOfStaffs` within it.

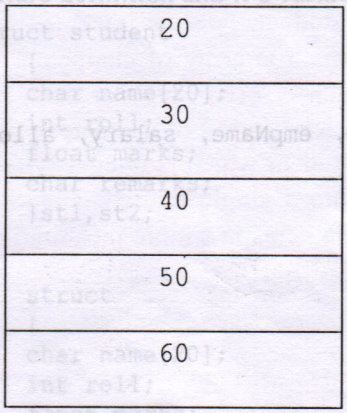
```
struct department
 {
  int deptID;
  char deptName[20];
  char location[10];
  int noOfStaffs;
 };
```

10.3 Comparison of Array and Structure

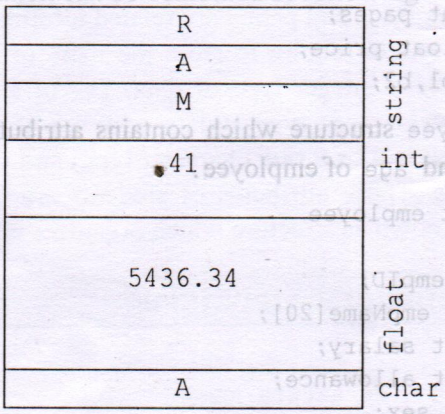
We have already defined the array (in CHAPTER-8) as a collection of data items of same types, while structure is simply defined as the collection data items of different types. Due to the capacity of holding different types of variables, structures help to form the more complex constructions such as linked list. Array can be the member of structure. Similarly more than one structure of same types can be stored in an array so called "`array of structure`".

A structure is a data type whose format is defined by the programmer. The "." Operator connects a structure variable name with a member of the structure.

Array: many data items of the same type



Structure: many data items of different data types



Array	Structure
1) An array is a collection of related data elements of same type (homogeneous).	1) A structure can have elements of different types (heterogenous).
2) An array is a derived data type	2) A structure is a user-defined data type
3) An array behaves like a built-in data types. We can declare an array variable and use it.	3) First we have to define a structure and then declare the variables of that type.
4) Array name is pointer to the first element of it.	4) Structure name is not pointer.

10.4 Structure Initialization

Like standard data type, the structure variable of a structure (i.e. the members of a structure variable) can be initialized with some values. The members to be initialized must appear in order as in the definition of structure within braces and separated by commas. But, C does not allow the initialization of individual structure members within its definition.

Syntax:

```
struct structure_name structure_variable={value1, value2,. . . valuen};
```

Where `value1` is initialized to the first member, `value2` is initialized to the second member and so on.

If the structure `student` is declared as

```
struct student
    {
     char name[20];
     int rollno;
     float marks;
    };
```

A variable of `structure student` can be initialized during its declaration as:

```
struct student st={"Ram",100,56.5};
```

This line is equivalent to

```
struct student st;
st.name="Ram";
st.rollno=100;
st.marks=56.5;
```

10.5 Processing a Structure / Accessing Member of Structure

The members of a structure are usually processed individually, as separate entity. Therefore, we must be able to access the individual structure members. A structure member can be accessed by using period or dot (i.e. ".") operator. The syntax for accessing member of a structure variable is as follows:

```
structure_variable.member
```

Here, `structure_variable` refers to the name of a structure-type variable and `member` refers to the name of a member within the structure. Again, dot separates the structure variable name from the its member name. The dot operator must have a structure variable on its left and a legal member on its right side. In the case of nested structure (if a structure member is itself a structure), the member within inner structure is accessed as:

```
structure_variable.member.submember
```

10.5.1 Precedence of Dot operator

The dot operator is a member of the highest precedence group and associates from left to right. Since it is an operator of the highest precedence, the dot operator will take precedence over various arithmetic, relational, logical, assignment and unary operators. Thus, `++st.marks` is equivalent to `++(st.marks)`, implying that the dot operator acts first and then unary operator.

Example 10.1

Create a structure named `student` that has `name`, `roll`, `marks` and `remarks` as members. Assume appropriate types and size of members. Write a program using structure to read and display the data entered by the user.

```
int main()
  {
    struct student
      {
      char name[20];
      int roll;
      float marks;
      char remark;
      };
    struct student s;
    printf("Enter name:\t");
    gets(s.name);
    printf("\nEnter roll:\t");
    scanf("%d",&s.roll);
    printf("\nEnter marks:\t");
    scanf("%f",&s.marks);
    printf("Enter remark p for pass or f for fail:\t");
    s.remark=getche();
    printf("\n\nThe student's Information is\n");
    printf("Student Name\t\t\tRoll\tMarks\tRemarks");
    printf("\n-----------------------------------------------\n");
    printf("%s\t\t%d\t%.2f\t%c",s.name,s.roll,s.marks,s.remark);
    getch();
    return 0;
    }
```

Output

```
Enter name:     Purushottam Sharma
Enter roll:     531
Enter marks:    89
Enter remark p for pass or f for fail:   P
The student's Information is
Student Name                    Roll    Marks   Remarks
-----------------------------------------------
Purushottam Sharma              531     89.00   P
```

10.6 How Structure Elements are Stored?

The elements or members of a structure are always stored in contiguous memory locations. A structure variable reserves number of bytes equal to sum of bytes needed to each of its members. For example, in following example, structure `student's` variable `st` takes `7` bytes in memory as its member variable `roll` needs `2` bytes, `marks` needs `4` bytes and `remarks` needs one byte. This can be illustrated as following example.

Example 10.2

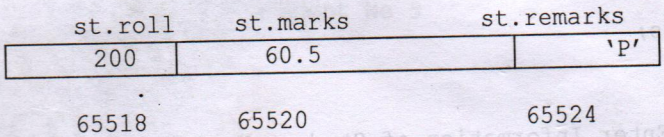
Write a program to illustrate memory locations of each member of structure.

```
int main()
    {
        struct student
         {
         int roll;
         float marks;
         char remarks;
        };
        struct student st={200,60.5,'P'};
        printf("\nAddress of roll=%u",&st.roll);
        printf("\nAddress of marks=%u",&st.marks);
        printf("\nAddress of remarks=%u",&st.remarks);
        getch();
        return 0;
    }
```

Output:

```
Address of roll=65518
Address of marks=65520
Address of remarks=65524
```

Explanation:



10.7 Array of Structure

Like array of `int`, `float` or `char` data type, there may be array of structures. In this case, the array will have individual structure as its elements. In our previous structure example, if we want to keep record of `50` students, we have to make `50` structure variables like `st1`, `st2`, `st3`, ...,`st49`. But this technique is not good. At this situation, we can use array of structures to store records of `50` students. Similar to declaring structure variable, an array of structure can be declared in two ways as illustrated below.

```
struct Employee
   {
    char name[20];
    int empId;
    float salary;
   }emp[10];
```

Here, `emp` is an array of `10` `Employee` structures. Each element of the array `emp` will contain the structure of the type `Employee`. The another way to declare structure is

```
struct Employee
   {
    char name[20];
    int empId;
    float salary;
   };
struct Employee emp[10];
```

Example 10.3

Write a program to create a structure named `student` that has `name`, `roll`, `marks` and `remarks` as members. Assume appropriate types and size of members. Use this structure to read and display records of `5` students.

```
int main()
   {
      struct student
            {
            char name[30];
            int roll;
            float marks;
            char remark;
            };
      struct student s[5];
      int i;
      float tempForMarks;
       for(i=0;i<5;i++)
        {
          printf("\n\nEnter Information of Student No %d\n",i+1);
          printf("Name:\t");
          scanf("%s",s[i].name);
          printf("\nRoll:\t");
          scanf("%d",&s[i].roll);
          printf("\nMarks:\t");
          scanf("%f",&tempForMarks);
          s[i].marks=tempForMarks;
          printf("Remarks(P/F):\t");
```

```
            s[i].remark=getche();
        }
        printf("\n\nThe Detail Information is\n");
        printf("Student Name\tRoll\tMarks\tRemarks");
        printf("\n-------------------------------------------\n");
        for(i=0;i<5;i++)
            printf("%s\t\t%d\t%.2f\t%c\n",s[i].name,s[i].roll,
                s[i].marks, s[i].remark);
        getch();
        return 0;
    }
```

Output

```
Enter Information of Student No 1
Name:   Ram
Roll:   100
Marks:  89
Remarks(P/F):   P
Enter Information of Student No 2
Name:   Bina
Roll:   101
Marks:  46
Remarks(P/F):   P
Enter Information of Student No 3
Name:   Shyam
Roll:   102
Marks:  78
Remarks(P/F):   P
Enter Information of Student No 4
Name:   Kunti
Roll:   103
Marks:  12
Remarks(P/F):   F
Enter Information of Student No 5
Name:   Nisha
Roll:   104
Marks:  97
Remarks(P/F):   P
The Detail Information is
Student Name    Roll    Marks   Remarks
-------------------------------------------
Ram             100     89.00   P
Bina            101     46.00   P
Shyam           102     78.00   P
Kunti           103     12.00   F
Nisha           104     97.00   P
```

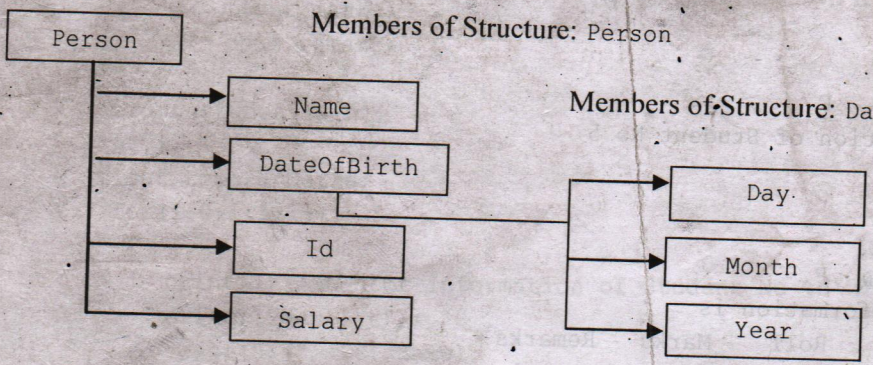
Initializing Array of Structures

We can initialize an array of structures in the same as a single structure. This is illustra as

```
struct Book
  {
  char name[20];
  int pages;
  float price;
  };
struct Book b[5]=
  {
  "Programming In C", 200,150.5,
  "Let Us C",  455, 315.
  "Programming With C", 558, 300.15,
  "Easy Steps in Programming", 800, 580,
  "How to Program in C", 450, 370
  };
```

10.9 Nested Structure (Structure within Another Structure)

One structure may be nested within another structure's definition. The outer structure is known as nesting structure and the inner structure is nested structure. In other words, a structure may be defined as member of another structure. Let us consider a structure `date` which has members `day`, `month` and `year`. The structure `date` can be nested within another structure `person` (i.e. structure `date` is member of another structure `person`).




```
struct date
  {
  int day;
  int month;
  int year;
  };
```

This structure can be nested within another structure as its member.

```
struct person
   {
   char name[20];
   int id;
   st  ct date dateOfBirthday;
    loat salary;
   ;p;
```

r tructure person contains a member dateOfBirhtday which is itself a structure with ree members. The members within structure date are accessed as

```
p.dateOfBirthday.day, p.dateOfBirthday.month , p.dateOfBirthday.year
```

But the members within person structure are accessed as usual

```
p.name, p.id , p.salary
```

The above nested structure can be nested in alternative way.

```
struct person
  {
  char name[20];
  int id;
  struct date
    {
    int day;
    int month;
    int year;
    }dateOfBirthday;
   float salary;
  }p;
```

Example 10.4

Write a program to create a structure named date that has day, month and year as its members. Include this structure as a member in another structure named Employee which has name, id and salary as other members. Use this structure to read and display employee's name, id, date of birthday and salary.

```
int main()
 {
 struct date
          {
          int day;
          int month;
          int year;
          };
  struct Employee
          {
          char name[20];
          int id;
          struct date  dateOfBirthday;
          float salary;
          }emp;
```

```
    printf("Name Of Employee:\t");
    scanf("%s",emp.name);
    printf("\nID Of Employee:\t");
    scanf("%d",&emp.id);
    printf("\nday of Bithday:\t");
    scanf("%d",&emp.dateOfBirthday.day);
    printf("\nMonth of Bithday:\t");
    scanf("%d",&emp.dateOfBirthday.month);
    printf("\nYear of Bithday:\t");
    scanf("%d",&emp.dateOfBirthday.year);
    printf("Salary of the Employee:\t");
    scanf("%f",&emp.salary);
    printf("\n\nThe Detail Information Of Entered Employee\n");
    printf("Name\t id\t day\t month\t year\t salary");
    printf("\n-----------------------------------------\n");
    printf("%s\t%d\t%d\t%d\t%d\t%.2f",emp.name,emp.id,
       emp.dateOfBirthday.day,emp.dateOfBirthday.month,
           emp.dateOfBirthday.year,emp.salary);
    getch();
    return 0;
}
```

Output

```
Name Of Employee:       Bina
ID Of Employee: 2000
day of Bithday: 7
Month of Bithday:       2
Year of Bithday:        1987
Salary of the Employee: 12000
The Detail Information Of Entered Employee
Name     id      day     month   year    salary
-----------------------------------------------
Bina     2000    7       2       1987    12000.00
```

10.10 Pointer to Structure

A pointer can also be used with a structure. To store address of a structure variable, we can define a structure type pointer-variable in normal way. Let us consider a structure `book` that has members `name`, `page` and `price`.

```
struct book
  {
  char name[10];
  int page;
  float price;
  };
```

Then, we can define structure variable and pointer variable of the structure type.

```
struct book b; /* b is structure variable */
struct book *bptr;/* bptr is pointer variable of structure type*/
```

Here, b is normal variable of structure type book where bptr is pointer variable which points (i.e. can store address of structure variable) to the structure book type variable. This declaration for a pointer to structure doesn't allocate any memory for a structure but allocates only for a pointer. To use structure's members through pointer bptr, memory must be allocated for a structure by using function malloc() or by adding a pointer assignment statement as given below.

```
bptr=&b;
```

Here, the base address of b is assigned to bptr pointer.An individual structure member can be accessed in terms of its corresponding pointer variable using : ptr_variable->member_name

Where -> is known as arrow operator and there must be pointer to the structure on the left side of the arrow operator.

i.e. Now, the members name, pages and price of book can be accessed using:

b.name or bptr->name or (*bptr).name

b.page or bptr->page or (*bptr).page

b.price or bptr->price or (*bptr).price

> The dot (.) operator connects a structure with a member of the structure, while the arrow (->) operator connects a pointer with the member of the structure.

Example 10.5

Write a program to create a structure named book which has name, pages and price as its members. Read name,page number and price of a book using structure variable. Finally display record of a book using pointer to structure instead of structure itself to access member variables.

```
int main()
    {
     struct book
        {
          char name[20];
          int pages;
          float price;
          };
      struct book b,*bptr;
      printf("Enter Book's Name:\t");
      gets(b.name);
      printf("\nNumber Of Pages:\t");
      scanf("%d",&b.pages);
      printf("\nPrice Of the Book:\t");
      scanf("%f",&b.price);
      bptr=&b;
      printf("\n\nBook Information Using Pointer i.e. arrow operator");
```

```
    printf("\nBook Name=%s\tPages=%d\tPrice=%.2f",bptr->name,
        bptr->pages,bptr->price);
    printf("\n\nBook Information Using Pointer i.e. * operator");
    printf("\nBook Name=%s\tPages=%d\t Price=%.2f",(*bptr).name,
        (*bptr).pages,(*bptr).price);
    printf("\n\nBook Information using structure variable
          i.e. dot operator");
    printf("\nBook Name=%s\tPages=%d\tPrice=%.2f",b.name,
          b.pages,b.price);
    getch();
    return 0;
}
```

Output

```
Enter Book's Name:      Let Us C
Number Of Pages:        540
Price Of the Book:      320.6
Book Information Using Pointer i.e. arrow operator
Book Name=Let Us C      Pages=540       Price=320.60
Book Information Using Pointer i.e. * operator
Book Name=Let Us C      Pages=540       Price=320.60
Book Information using structure variable i.e. dot operator
Book Name=Let Us C      Pages=540       Price=320.60
```

Explanation: Here, to display the value of the member variables of a structure, we have used three ways/methods. All the methods gives the same result.

10.10.1 Dynamically Allocated Structure

The pointer to structure can be used to define a dynamically allocated structure. We can dynamically allocate memory space for a structure variable with malloc() function and assign the allocated space to a structre pointer variable. Such type of structure pointer variable is called dynamically allocated structure (or dynamic structure).

Example: Let us consider an already defined structure named Book. Now we can define a structure pointer of that type and allocate space dynamically.

```
struct Book *b;

b=(struct Book *)malloc(sizeof(struct Book));
```

Here, `b` is dynamically allocated structre.

10.11 Function and Structure

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 to a function as its argument.

10.11.1 Passing structure members to a function

A member of structure may be treated just as an ordinary variable. For example, an integer member of a structure is treated just as an integer. Thus, the structure members can be passed to a function as argument like ordinary variables.

Example 10.6

Write a program to illustrate passing structure members to a function.

```
void display(char empName[], int id, float sal)
        {
         printf("\nName\t\tID\t\tSalary");
         printf("\n-------------------------------------");
         printf("\n%s\t\t%d\t\t%.1f",empName,id,sal);
         printf("\n-------------------------------------");
        }
int main()
        {
           struct employee
                       {
                       char name[20];
                       int id;
                       float salary;
                       };
         struct employee emp;
         printf("Employee Name:\t");
         scanf("%s",emp.name);
         printf("Employee id:\t");
         scanf("%d",&emp.id);
         printf("Salary of the Employee:\t");
         scanf("%f",&emp.salary);
         printf("\nThe Entered Information Of the Employee is\n");
         display(emp.name,emp.id,emp.salary);
       /*function call in which each elements are passed individually*/
         getch();
         return 0;
        }
```

Output

```
Employee Name:          Ram
Employee id:  100
Salary of the Employee: 15000
The Entered Information Of the Employee is
Name            ID              Salary
-------------------------------------
Ram             100             15000.0
```

10.11.2 Passing an entire structure to a function

It is also possible to pass an entire structure to a function as its argument. The structure variable is treated as like an ordinary variable. An entire structure includes all the members of the structure. Thus, the passing an entire structure variable to a function is equivalent to passing all the members of the structure individually to the function.

Example 10.7

Write a program to illustrate passing whole structure to a function.

```
struct employee
        {
         char name[20];
         int id;
         float salary;
        };
void display(struct employee e)
       {
        printf("\nName\t\tID\t\tSalary");
        printf("\n-------------------------------------");
        printf("\n%s\t\t%d\t\t%.1f",e.name,e.id,e.salary);
        printf("\n-------------------------------------");
       }
int main()
      {
      struct employee emp;
      printf("Employee Name:\t");
      scanf("%s",emp.name);
      printf("Employee id:\t");
      scanf("%d",&emp.id);
      printf("Salary of the Employee:\t");
      scanf("%f",&emp.salary);
      printf("\nThe Entered Information Of the Employee is\n");
      display(emp);
      /*function call in which whole structure is passed */
      getch();
      return 0;
      }
```

Output

```
Employee Name:    Shyam
Employee id:    200
Salary of the Employee: 20000
The Entered Information Of the Employee is
Name            ID              Salary
-------------------------------------
Shyam           200             20000.0
-------------------------------------
```

Explanation: Here, output is similar to previous program in which individual elements were passed to the function. But the difference is that the whole structure has been passed in this example instead of individual members. When a structure variable `emp` of type `Employee` is passed to a function, then the type of formal argument in function definition must be of the type `Employee`. Thus, the structure `Employee` must be used in both in `main()` function and user defined function `display()`. Thus, this structure has been declared global (i.e. outside of `main()` function).

10.11.3 Passing Structure pointer to a function

A structure pointer can be passed to a function. When a pointer to a structure is passed as an argument to a function, the function is called by reference. Therefore, any changes that are made within the function definition are also visible within the calling function.

Example 10.8

Write a program to illustrate passing structure pointer to functions.

```
struct employee
        {
         char name[20];
         int id;
         float salary;
        };
void display(struct employee e)
       {
        printf("\nName\t\tID\t\tSalary");
        printf("\n-------------------------------------");
        printf("\n%s\t\t%d\t\t%.1f",e.name,e.id,e.salary);
        printf("\n-------------------------------------");
       }
void increaseSalary(struct employee *ee)
       {
        ee->salary=ee->salary+1000;
       }
int main()
       {
        struct employee emp;
        printf("Employee Name:\t");
        scanf("%s",emp.name);
        printf("Employee id:\t");
        scanf("%d",&emp.id);
        printf("Salary of the Employee:\t");
        scanf("%f",&emp.salary);
        printf("\nThe Entered Information Of the Employee is\n");
        increaseSalary(&emp);
            /* function call using pointer to structure*/
        display(emp);
         /*function call in which whole structure is passed */
        getch();
        return 0;
       }
```

Output

```
Employee Name:    BINA
Employee id:    300
Salary of the Employee: 7000
The Entered Information Of the Employee is
Name            ID              Salary
-------------------------------------
BINA            300             8000.0
-------------------------------------
```

Explanation: Here, pointer to structure is passed to the function `increaseSalary()` and the entire structure is passed to function `display()`which displays all the individual members of it.

10.11.4 Passing an array of structure to a function

Passing an array of structure to a function involves the same syntax and characteristics as passing an array of an ordinary type to a function. The passing is done using a pointer and consequently, any changes made within the function definition to the structure, are also visible in the calling program/function.

Example 10.9

Write a program to create a structure named `student` that has `name`, `roll`, `marks` and `remarks` as its members. Assume appropriate types and size of members. Use the structure to read and display records of `four` students. Create two functions: one is to read information of students and other is to display the information on screen. Pass an array of structure to above functions as their arguments.

```
#define size 4
int i;
struct student
        {
        char name[30];
        int roll;
        float marks;
        char remark;
        };
void display(struct student st[])
      {
      printf("Student Name\tRoll\tMarks\tRemarks");
      printf("\n-------------------------------------------------\n");
         for(i=0;i<size;i++)
         printf("%s\t\t%d\t%.2f\t%c\n",st[i].name,st[i].roll,
             st[i].marks,st[i].remark);
       }
void read(struct student stu[])
    {
      float tempForMarks;
      for(i=0;i<size;i++)
      {
       printf("\n\nEnter Information of Student No %d\n",i+1);
       printf("Name:\t");
       scanf("%s",stu[i].name);
       printf("\nRoll:\t");
       scanf("%d",&stu[i].roll);
       printf("\nMarks:\t");
       scanf("%f",&tempForMarks);
       stu[i].marks=tempForMarks;
       printf("Remarks(P/F):\t");
       stu[i].remark=getche();
      }
    }
int main()
```

```
{
 struct student s[size];
 float tempForMarks;
 printf("\nRead information of Employee from user\n");
 read(s);
 printf("\n\nThe Detail Information is\n");
 display(s);
 getch();
 return 0;
}
```

Output

```
Read information of Employee from user
Enter Information of Student No 1
Name:   Hari
Roll:   101
Marks:  56
Remarks(P/F):   p
Enter Information of Student No 2
Name:   Bina
Roll:   102
Marks:  67
Remarks(P/F):   p
Enter Information of Student No 3
Name:   Gita
Roll:   103
Marks:  89
Remarks(P/F):   p
Enter Information of Student No 4
Name:   Krishna
Roll:   104
Marks:  12
Remarks(P/F):   f
The Detail Information is
Student Name    Roll        Marks   Remarks
-------------------------------------------------
Hari            101         56.00       p
Bina            102         67.00       p
Gita            103         89.00       p
Krishna         104         12.00       f
```

10.12 Union

A union is similar to a structure with respect to syntax and use. The union also contains members whose individual data types may differ from one another. The distinction is that all members within a union share the same storage area of computer memory, whereas each member within a structure is assigned its own unique storage. Thus, the union is used to conserve memory space. Since same memory is shared by all it members, only one variable can reside into memory at a time. When another variable is to be stored into the memory, the previous is replaced (i.e. previous can not persist). Thus, a union is not useful for applications

involving multiple members where the values need to be assigned to all of the members at the same time. Therefore, although a union may contain many members of different types, it can handle only one member at a time. The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union.

Examples:

The union is created as:

```
union student
    {
    int roll;
    float marks;
    } ;
```

Here, the union `student` has members `roll` and `marks`. The data type of `roll` is integer which contains `2` bytes in memory and the data type of `marks` is `float` which contains `4` bytes in memory. As all union members share same memory, the compiler allocates largest memory (i.e. `4` bytes in this case). The declaration of union and its variable is similar to that of structure. The union variable can be passed to function and it can be member of any other structure. The union can be illustrated as following examples;

> The union use the shared memory space and so handle only one member at a time, while different memory will be reserved for each structure member and so can handle operation on more members at a time.

Example 10.10

Create a union named `student` that has `roll` and `marks` as member. Assign some values to these members one at a time and display the result one at a time.

```
int main()
{
 union student
     {
      int roll;
      float marks;
     };
 union student st;
 st.roll=455;
 printf("\nRoll=%d",st.roll);
 st.marks=78;
 printf("\nMarks=%f",st.marks);
 getch();
 return 0;
}
```

Output

```
Roll=455
Marks=78.000000
```

If two members are used simultaneously, the output is unexpected as following:

```
int main()
{
 union student
       {
       int roll;
       float marks;
       };
 union student st;
 st.roll=455;
 st.marks=78;
 printf("\nRoll=%d",st.roll);
 printf("\nMarks=%f",st.marks);
 getch();
 return 0;
}
```

Output

```
Roll=0
Marks=78.000000
```

Here, `roll` is zero as memory is replaced by another variable `st.marks`.

Difference between Structure and Union

Structure	Union
1) Each member of a structure is assigned its own unique storage. It takes more memory spaces than that of union.	1) All members of a union share the same storage area of computer memory. It takes less memory than that of structure.
2) The amount of memory required to store a structure is the sum of the memory sizes required by all its members.	2) The amount of memory required to store a union is same as memory size occupied by a member which requires largest memory space among the members.
3) All the members of structure can be accessed at any point of time.	3) Only one member of union can be accessed at any given time.

4) Structure is declared as
`struct student`
`{`
`char name[20];`
`int roll;`
`float marks;`
`}st;`
	4) Union is declared as
	`union student`
	`{`
	`char name[20];`
	`int roll;`
	`float marks;`
	`}st;`

10.13 Enumerated Data Types

An enumerated data type is an user-defined integer type consisting of set of named integer constants. The keyword 'enum' is used to declare the enumerated data. The syntax of declaring enumerated data type is:

```
enum enum-name{identifier1, identifier2, identifier3,.....,identifiern};
```

Examples:

```
enum days {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
```

Here, 'days' is an user-defined enumerated data type which can only hold values defined inside braces in a sequence where each value is sequentially indexed unless otherwise specified. The integer value is assigned to each members of enumerator. Zero is assigned to first member, 1 is assigned to second and so on.

```
Sunday = 0, Monday = 1, Tuesday = 2 , ….., Saturday = 6
```

Example 10.11

Write a program to illustrate the use of enumerated data type.

```
#include<stdio.h>
int main ()
{
enum days{ Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
enum days weekday=Monday; //define enum data type as weekday
//print value of each enum data itmes
printf("The week days are:\n");
printf("Sunday=%d\n", Sunday);
printf("Monday=%d\n", Monday);
printf("Tuesday=%d\n", Tuesday);
printf("Wednesday=%d\n", Wednesday);
printf("Thursday=%d\n", Thursday);
printf("Friday=%d\n", Friday);
printf("Saturday=%d\n\n", Saturday);
printf("Weekday=%d", weekday); //weekday is assigned Monday.
return 0;
}
```

```
Output:
The week days are:
Sunday=0
Monday=1
Tuesday=2
Wednesday=3
Thursday=4
Friday=5
Saturday=6
Weekday=1
```

Explanation: If we want to explicitly assign values to members of an enumerator,we should specifically mention the values at the time of declaration of enumerated data type.

```
enum days {Sunday = 1, Monday, Tuesday = 20, Wednesday, Thursday, Friday, Saturday = 30};
```

In this declaration, we defined values of `Sunday` as `1` and `Tuesday` as `20` then what about other non-assigned constant's value? Yes the value of next constants will be just a one increment of previous constant's value. The above declaration will assign the values as following:

```
Monday  = Sunday +1 = 2
Tuesday = 20 (it is already assigned)
Wednesday = Tuesday +1 = 21
Thursday = Wednesday +1 = 22
Friday = Thursday + 1 = 23
Saturday = 30 (it is already assigned, if not assigned, its value would be 24)
```

Example 10.12

Write a program to illustrate enumeration with different user-defined values to the members.

```
#include<stdio.h>
int main (void)
{
enum colors {red=25, blue, white=254, green=45, pink, purple, black=0,  };
typedef enum colors wallcolor; // can use typedef keyword same as in structure
wallcolor homewall;
homewall=red;
printf ("\n wallcolor=%d", homewall);
homewall = blue;
printf ("\n wallcolor=%d", homewall);
homewall=white;
printf ("\n wallcolor=%d", homewall);
homewall=green;
printf ("\n wallcolor=%d", homewall);
homewall=pink;
printf ("\n wallcolor=%d", homewall);
homewall=purple;
printf ("\n wallcolor=%d", homewall);
homewall=black;
printf ("\n wallcolor=%d", homewall);
return 0;
}
```

Output:

```
wallcolor=25
wallcolor=26
wallcolor=254
wallcolor=45
wallcolor=46
wallcolor=47
wallcolor=0
```

Example 10.13

Write a program using enumeration to find streams offered in bachloer of engineering by Pulchowk Engineering Campus.

```
#include<stdio.h>
#include<conio.h>
int main (void)
{
enum stream {bce, bex, bct, bel, bme,bag, bar, bin}pulchowk;
```

```
do
{
printf("Enter index number to find stream in pulchowk campus:");
scanf("%d", &pulchowk);
switch(pulchowk)
   {
    case bce:
         puts("Bachelor of Civil Engineering");
         break;
    case bex:
         puts("Bachelor of Electronics Engineering");
         break;
    case bct:
         puts("Bachelor of Computer Engineering");
         break;
    case bel:
         puts("Bachelor of Electrical Engineering");
         break;
    case bme:
         puts("Bachelor of Mechanical Engineering");
         break;
    case bag:
         puts("B.E. in Agricultural not available");
         break;
    case bar:
         puts("Bachelor Architectural Engineering");
         break;
    case bin:
         puts("B.E. in Industrial not available");
         break;
    default:
         puts("Wrong Stream Index Entered");
         break;
   }
}while(pulchowk>=0 && pulchowk <=7);
return 0;
}
```

Output:

```
Enter index number to find stream in pulchowk campus:1
Bachelor of Electronics Engineering
Enter index number to find stream in pulchowk campus:3
Bachelor of Electrical Engineering
Enter index number to find stream in pulchowk campus:0
Bachelor of Civil Engineering
Enter index number to find stream in pulchowk campus:2
Bachelor of Computer Engineering
Enter index number to find stream in pulchowk campus:7
B.E. in Industrial not available
Enter index number to find stream in pulchowk campus:8
Wrong Stream Index Entered
```

However enumerated data types hold the integer values of predefined integer constants in the declaration, latest C compiler makes it possible to hold other integer constants too outside the defined range.

Example 10.14

Write a program to define a structure of representing BCT student's `name`, `rollno`, `marks_c`, `status{fail, pass}`. Pass the structure to a function by reference. Return the structure from the function to main program after assigning values to structure members. Display the values in main program (Use enumerated data types to represent status & structure pointer).

```
#include<stdio.h>
#include<stdlib.h>
typedef struct
   {
    char name[40];
    int rollno;
    float marks_c;
    enum remarks{fail,pass}status;
   }stream;
stream *std(stream *bct); //function prototyping
int main(void)
       {
       stream bct70;
       stream *ptrbct=std(&bct70);
       printf("\n Name is: %s", ptrbct->name);
       printf("\n Roll No is: %d", ptrbct->rollno);
       printf("\n Marks_c is: %3.2f", ptrbct->marks_c);
       printf("\n Status is: %s", ptrbct->status?"Pass":"Fail");
       return 0;
       }

   stream *std(stream *bct) //function starts
       {
       char value[20];
       printf("enter name:");
       gets(bct->name);
       printf("enter rollno:");
       gets(value);
       bct->rollno=atoi(value);
       printf("marks in c:");
       gets(value);
       bct->marks_c=atof(value);
       if(bct->marks_c>40 && bct->marks_c<100)
           bct->status = pass;
       else
           bct->status = fail;
       return bct;
      }
```

```
Output:
enter name:  Michael Jaction
enter rollno:502
marks in c: 88.5

Name is: Michael Jaction
 Roll No is: 502
 Marks_c is: 88.50
 Status is: Pass
```

10.14 Self Referential Structure

A structure which has at least one member of type `pointer to the same structre` is known as self referential structure. Let us consider the following example:

```
struct List
 {
 int data;
 struct List *next;
 } ;
```

Here, `List` is a structure which has two members: one is `data` of type `int`, another is `next` which is pointer of parent type structure (i.e. `List`). Thus, the member `next` stores address of structure variable of type `List` (i.e. it points to the variable of the same structure). Here, `List` is a self referential structure.

Generally, the self referential structure is used to implement linked list where there are a number of nodes. Each node has two parts: first part is `data (i.e. information)` and second part is `address (i.e. pointer field)` which points to the next node in the list. The address field of the node should be pointer as it should point to next node. The detail about linked list will be discussed later sections of this chapter.