Structures

Concept:

In real life, there are situations, where we need to store data items that are logically related, but contain dissimilar type of information.

e.g.

1. Employee record is a collection of data items such as employee code, employee name, salary, address etc. All this data is of dissimilar type.
2. Book record is a collection of data items such as Book id, name, price, author etc. All this data is of dissimilar type.
3. Student record is a collection of data items such as Student id, name, marks, percentage etc. All this data is of dissimilar type.

In order to handle such situations, C provides a data type, called Structures.

Since ordinary variable and arrays are unable to store such kind of information.

Ordinary variable can store only one value at a time.

Arrays can store multiple values but of similar type.

Definition:

Structure is User defined data type which can store more than one value of dissimilar data type.

Structure is a collection of logically related data items of different data types grouped together under a single name.

Structure is analogous to records. As records contain different fields, the data items that make up a structure are called its members or fields.

Structure Declaration:

Syntax:

```
struct structure_name
{
  Data_type   member1;
  Data_type   member2;
  ……..
};
```

e.g.

```
struct   book
{ int  book_id;
  char name[10];
```

```
    int pages;
    float price;
};
```

Now, We have only created the format of structure, But we haven't created variables of structure which can store the actual values. By creating Template of structure i.e. declaring a structure one can not get memory.

When we declare variable for that structure memory gets allocated to store values. Memory allocated for one variable is equal to addition of no of bytes required to store all the members in that structure.

There are 3 ways to declare structure variable:

1. Structure variable declaration in structure template.

```
struct book
{
        int book_id;
        char name[10];
        int pages;
        float price;
}b1,b2;
```

b1 and b2 are two variables of type structure book. b1 will get 18 contiguous bytes in memory. . b2 will get 18 contiguous bytes in memory.

2. Structure variable declaration anywhere in the program.

```
struct book
{       int book_id;
        char name[10];
        int pages;
        float price;
};
void main()
{
        struct book b1,b2;
}
```

b1 and b2 are two variables of type structure book. b1 will get 18 contiguous bytes in memory. . b2 will get 18 contiguous bytes in memory.

3. Array of Structure variables.

```
struct   book
{
        int  book_id;
        char name[10];
        int pages;
        float price;
}b[10];
```

b is array of structure book. b will get 180 bytes (10 elements * 18 bytes each) in memory.

OR

```
struct   book
{
        int  book_id;
        char name[10];
        int pages;
        float price;
};
void main()
{       struct book b[10];
}
```

b is array of structure book. b will get 180 bytes (10 elements * 1b bytes each) in memory.

Accessing Structure Members:

Individual members of structure can be accessed using dot (.) operator. This operator is called structure member operator.

Syntax:

structure_variable_name.structure_member;

e.g.

b1.book_id = 1;

//b2.name = “Let Us C”;

b2.pages = 200;

Examples:

1. Write a program to store student information using structure and display records (student record contain student id, name, marks in 3 subject, average marks).

Solution:-

```
#include<stdio.h>
#include<conio.h>
struct stud
{
  int roll;
  char name[10];
  int marks[3];
 float avg;
};

void main()
{
  int i,j,sum;
  struct stud s[3];
  clrscr();
  for(i=0;i<3;i++)
  {
    printf("\n");
    printf("enter the roll of student %d:-",i+1);
    scanf("%d",&s[i].roll);
    printf("enter the name of student %d:-",i+1);
    scanf("%s",s[i].name);
    sum=0;
    for(j=0;j<3;j++)
    {
     printf("enter the marks of subj %d:- ",j+1);
     scanf("%d",&s[i].marks[j]);
     sum=sum+s[i].marks[j];
    }
    s[i].avg=(float)sum/3.0;
  }
  for(i=0;i<3;i++)
  {
    printf("roll of student %d is %d\n",i+1,s[i].roll);
    printf("name of student %d is %s\n",i+1,s[i].name);
```

```
    printf("avg of marks of student %dis %f \n",i+1,s[i].avg);

  }
 getch();
}
```

2. Write a program to store student information using structure and display records in descending order of student's average marks.(student record contain student id, name, marks in 3 subject, average marks).

```
#include<stdio.h>
#include<conio.h>
struct stud
{
  int roll;
  char name[10];
  int marks[3];
  float avg;
};

void main()
{
  int i,j,sum;
  struct stud s[3],temp;
  clrscr();
  for(i=0;i<3;i++)
  {
    printf("\n");
    printf("enter the roll of student %d:-",i+1);
    scanf("%d",&s[i].roll);
    printf("enter the name of student %d:-",i+1);
    scanf("%s",s[i].name);
    sum=0;
    for(j=0;j<3;j++)
    {
     printf("enter the marks of subj %d:- ",j+1);
     scanf("%d",&s[i].marks[j]);
     sum=sum+s[i].marks[j];
    }
    s[i].avg=(float)sum/3.0;
  }
  for(i=0;i<3;i++)
  {
    printf("roll of student %d is %d\n",i+1,s[i].roll);
    printf("name of student %d is %s\n",i+1,s[i].name);
    printf("avg of marks of student %dis %f \n",i+1,s[i].avg);

  }
//logic for sorting
for(i=0;i<3;i++)
{
  for(j=i+1;j<3;j++)
  {
            if(s[i].avg  <  s[j].avg)
            {
```

```
                    temp= s[i];
                    s[i] = s[j];
                    s[j] = temp;
            }
  }

}

printf("\n Student Records in descending order of avg marks" );
for(i=0;i<3;i++)
{
  printf("Information of %d  student",i+1);
  printf("\n Student id is %d",s[i].roll);
  printf("\n Student Name");
  puts(s[i].name);

printf("\n Students average marks %f",s[i].avg);
}
  getch();
}
```

Passing structures to Functions:

Structures can be passed to a function as a parameter and also can be returned by functions.

Two ways:

1. Passing individual structure element(passing individual member of structure)
2. Passing structure as a whole(pass all members at a time, pass structure variable)

Union

A union is a way of providing an alternate way of describing the same memory area. In this way, you could have a struct that contains a union, so that the "static", or similar portion of the data is described first, and the portion that changes is described by the union. The idea of a union could be handled in a different way by having 2 different structs defined, and making a pointer to each kind of struct. The pointer to struct "a" could be assigned to the value of a buffer, and the pointer to struct "b" could be assigned to the same buffer, but now a->somefield and b->someotherfield are both located in the same buffer. That is the idea behind a union. It gives different ways to break down the same buffer area.

Syntax:-

```
union data
{
char a;
int x;
float f;
} u;
```

To access the fields of a union, use the dot operator(.) just as you would for a structure. When a value is assigned to one member, the other member(s) get whipped out since they share the same memory.
Eg:-printf(“%c”,u.a);

Difference between Union and structure:
The difference between structure and union in c is:

1. union allocates the memory equal to the maximum memory required by the member of the union but structure allocates the memory equal to the total memory required by the members.
2. In union, one block is used by all the member of the union but in case of structure, each member have their own memory space.
3. union is best in the environment where memory is less as it shares the memory allocated. But structure cannot be implemented in shared memory.

4. As memory is shared, ambiguity is more in union, but less in structure.

5. self referential union cannot be implemented in any data structure ,but self referential structure can be implemented.

Difference in their Usage:
While structure enables us treat a number of different variables stored at different in memory , a union enables us to treat the same space in memory as a number of different variables. That is a Union offers a way for a section of memory to be treated as a variable of one type on one occasion and as a different variable of a different type on another occasion.