# Structures:-

Structure is a collection of elements of dissimilar data types.

(Or)

A collection of one or more variables typically of different types, grouped together under a single name.

**General Format of declaration:**

```
struct tag
{
  datatype variable1;
  datatype variable2;
  ……..
};
```

Structure declaration always starts with '**struct'** keyword. Tag refers to the name of the structure.

The difference between arrays and structures is arrays can hold a collection of elements of similar data type, whereas structures can hold a collection of elements of different data types.

Ex:-

```
struct motor
{
  float volts;
  float amps;
  int phases;
  float rmp;
};
```

```
struct book
{
  char b_name [30];
  char author[30];
  int pages;
  folat price;
};
```

The variables which declared and defined in the structure are known as members of the structures.

```
sturct motor a, b, c;
```

The above statement declares and lets storage for three variables of each of the type struct motor.

```
struct motor *m;
```

It declares a pointer to an object of type struct motor.

**Features of structures:-**

- In structure declaration tag name is optional, which means a structure can be declared without any name called anonymous structures.

Ex: - struct
{
int x;
char y;
};

- The size of a structure variable is sum of its size occupied by each member of the structure.

Ex: - struct emp
{
int empid;
char name[20];
float sal;
} e;
Sizeof (e) = (20+4+4) =28/26.

- If one or more members of a structure are pointer to the same structure, the structure sis known as self-referential structure.

Ex: - struct student
{
int id;
struct student *next;
};

- The operators used with structure are dot(.) and arrow (->) operators.
- The values of a structure variable can be assigned to another structure variable of the same type using the assignment operator.

```
#include<stdio.h>
#include<conio.h>
main ()
{
    struct book
    {
        char bname[10];
        char aname [10];
        int pages;
```

```
        float price;
    }
    struct book p1, p2, p3;
    p1= {"cds","yswanth","800","400.25"};
    strcpy (p2.bname, p1.b name);
    strcpy (p2.aname, p1.aname);
    p2.pages=p1. pages;
    p2 price=p1.price;
    p3=p2;
    printf ("%s%s%d%f", p1.bname, p1.aname, p1.pages, p1.price);
    printf ("%s%s%d%f", p2.bname, p2.aname, p2.pages, p2.price);
    printf ("%s%s%d%f", p3.bname, p3 aname, p3.pages, p3.price);
}
```

## Accessing Members of A Structure:-

We can access the members of a structure by using structure variables. Structure variables are normal variables and pointer variables. Dot (.) operator is used for normal variables and arrow (->) operator is used for pointers.

Declaring structure variables for struct motor:

```
struct motor a;
struct motor p [20];
struct motor *m;
```

then

| | |
|---|---|
| a. volts | p[i].volts |
| a. amps | p[i].amps |
| a. phases | p[i].phases |
| a. rpm | p[i].rpm |

```
(*m).volts
(*m).amps
(*m).phase
(*m).rpm
```

Note: - why parentheses? Because '**.**' Operator has higher precedence than unary '*****'. Instead of this we may write in the following format also.

```
m→ volts
m→ amps
m→ phases
m→ rmp
```

//Write a 'c' program that use structures and display book details.

```
# include < stdio.h>
# include <conio.h>
void main ()
{
        struct book
       {
          char b_name[20];
          char b_author [20];
          int  pages;
          float price;
        };
       sturct  book b1, b2;
printf ("enter details of first book \n");
scanf ("%s %s %d %f", &b1.b_name, &b1.b_auhtor, &b1.pages,&b1.price);
printf ("enter the details of second book \n"0;
scanf ("%s %s %d %f", &b2.b_name, &b2.b_author, &b2.pages,&b2.price);
printf ("First book details \n");
printf ("\t name = %s \n\t author =%s \n\t pages =%d\n\t price =%f\n",
        b1.b_name, b1.b_author, b1.pages, b1.price);
printf ("Second book details \n");
printf ("\t name = %s \n\t author =%s \n\t pages =%d\n\t price =%f\n",
        b2.b_name, b2.b_author, b2.pages, b2.price);
getch():
}
```

//Write a program to display the size of structure elements by using size of () operator

```
main ()
{
  struct book
    {
      char name [20];
```

```
    int pages;
    folat price;
  };
      struct book b;
      clrscr ();
      printf ("\n size of structure elements \n");
      printf ("name =%d", size of (b.name));
      printf ("pages =%f", size of (b.pages));
      pritnf ("price =%f", size of (b.price));
      printf ("total bytes =%d", size of (b));
      getch ();
  }
```

output:-

size of structure elements:

name: 20

pages: 1

price: 1

total bytes=22

## Initialization of structures:

```
struct book
{
  char name [30];
  int pages;
  float price;
};
```

struct bookl m= {"c language", 250, 300};

(Or)

m.name= "c language";

```
m.pages=250;

m.price=300;
```

Operations on struct**:-**

Copy/ align

```
struct motor p, q;

p=q;
```

Get address of structures

```
struct motor p;

struct motor *m;

m=&p;
```

**Structure within structure** :- (Nested structures)

One structure can be nested in another structure. They are known as nested structure.

Example:

```
    main()
{
    struct address
    {
            int door no;
            char street[15];
            char city[15];
            long  int pincode;
    };
    struct emp
    {
            char name[];
            struct address p;
    };
    struct emp s={"John",12,"J street","Banglore",786215};
    printf("%s %d %s %s %d", s.name, s.door no, s.street, s.city, s.pincode);
    getch();
}
```

Output:- John 12 j street Banglore 786215

## Array of structures:-

```
#include<stdio.h>
#include<conio.h>
void main()
{
        struct  book
        {
                char bname[];
                char aname[];
                int  pages;
                float price;
        }b1[20];
        int i;
        printf("enter 10 book details");
        for(i=0;i<10;i++)
        {
scanf("%s %s %d %f",&b1[i].bname,&b1[i].aname,&b[i].pages,&b[i].price);
        }
        printf("entered book details are");
        for(i=0;i<10;i++)
        {
printf("bookname=%s\n author=%s\n pages=%d\n price=%f\n", b1[i].bname,
b1[i].aname, b1[i].pages, b1[i].price);
        }
        getch();
}
```

## Pointer to structure:-

Pointer is used to store the address of another variable. That variable may be of any type. As we declare pointer for int, float, char, in the same way we can declare for structures also. In this case, the starting address of the member variable can be accessed. This type of pointers is called as pointer to structures.

Example:-

```
#include<stdio.h>
#include<conio.h>
void main()
{
```

```
        struct  book
                {
                        char bname[];
                        char aname[];
                        int  pages;
                        float price;
                };
        struct book b={"cds","kamthane",500,399.00};
        struct book  *p;
        p=&b;
        clrscr();
        printf("%s %s %d %f", b.bname, b.aname, b.pages, b.price);
        printf("%s %s %d %f", p->bname, p->aname, p->pages, p->price);
        getch();
}
```

Output:-

CDS kamthane 500 399.000000

CDS kamthane 500 399.000000

## Structures and functions:-

Like ordinary variables a structure variable can also be passed to a function. We may either pass individual structure members or pass whole structure variable at one time.

**Example1:-**

// passing individual structure members to a function

```
#include<stdio.h>
#include<conio.h>
void show(char *,char *,int,float);
void main()
{
        struct book
        {
                char bname[20];
                char aname[20];
                int pages;
                float price;
```

```
        };
        struct book s={“java”,”steve”,”1000”,”450.00”};
        show(s.bname,s.aname,s.pages,s.price);
        getch();
}
void show(char *p,char *q,int m,float n)
{
        printf(“%s %s %d %f”,p,q,m,n);
}
```

Output:-

Java Steve 1000 450.000000

**Example 2:-**

// passing whole structure variable at once to a function

```
#include<stdio.h>
#include<conio.h>
struct book
        {
                char bname[20];
                char aname[20];
                int pages;
                float price;
        };
void show(struct book);
void main()
{
        struct book  k={“java”,”steve”,”1000”,”450.00”};
        show(k);
        getch();
}
void show(struct book)
{
        printf(“%s %s %d %f”, k.bname, k.aname, k.pages, k.price);
}
```

Output:-

Java Steve 1000 450.000000

## Typedef:-

- The purpose of typedef is to assign alternative names to existing types
- Type def is a keyword in 'c'
- A typedef declaration does not reserve storage.
- The names you define using typedef are not new data types, but synonyms for the data types or combination of data types they represent.

Example:-

```
typedef int LENGTH;
        LENGTH length,weight,height;
```

This is same as

```
        int  length,weight,height;
```

Usage of typedef in structures:-

```
struct  new
    {
        int  var1;
        int var2;
        float  var3;
    };
```

To create a variable of struct new type we need struct key word as

```
struct new p;
```

A typedef can be used to eliminate the need for struct keyword in 'c'

Ex:

```
typedef struct new TYPE;
```

We can now create a variable of this type with

```
TYPE p;
```

**Advantages of type def:-**

- It provides a mean to make a program more portable. Instead of having to change a type everywhere it appears throughout the programs source files, only a single typedef statement needs to be changed
- A typedef can make a complex declaration easier to understand.

## Bit fields:-

- A bitfield is a common idiom used in computer programming to compactly store a value as a short series of bits
- A bitfield is most commonly used to represent integral types of known fixed bit-width

Ex:

If we want to store an employee's information, each employee can have the following data

- Be a male or female
- Be single, married , divorced or widowed
- Have one of the eight different hobbies

According to the above data we need one bit to store gender, two bit to store marital status, three for hobby. We need six bits to pack all the data.

To do this using bit fields, we declare the following structure.

```
struct emp
{
        unsigned gender : 1 ;
        unsigned mar_stat : 2;
        unsigned hobby : 3;
};
```

The colon (:) in the above declaration tells the compiler that we are talking about bitfields and the number after it tells how many bits are to be allotted for that field.

```
#include<stdio.h>
#define male 0
#define female 1
#define single 0
#define married 1
#define divorced 2
#define widowed 3

main()
{
        struct emp
```

```
{
unsigned gender:1 ;
unsigned mar_stat:2;
unsigned hobby:3;
};
struct emp d;
d.gender = male;
d.mar_stat = divorced;
d.hobby = 5;
printf("gender=%d\n", d.gender);
printf("marital status=%d\n", d.mar_stat);
printf("bytes occupied by d=%d\n", sizeof(d));
getch();
}
```

Output:-

```
Gender=0
Marital status=2
Bytes occupied by d=2
```

## Enumerated data types:-

- An enumerated data type is a data type consisting of a set of named values called elements, members or enumerators of the type
- enum is the keyword used for declaration of enumerated data types
- the enumerator names are usually identifiers that behave as constants in the language

ex

red, green, black, blue belongs to colour enum

club, diamond, heart, spade belongs to play card enum

married, single, divorced, widowed belongs to marital status enum

- ✓ The following declaration declares the data type and specifies its possible values. These values are called 'enumerators'

```
enum cardsuit
{ CLUBS, DIAMONDS, HEARTS, SPADES, };
```

- ✓ The following statement declares variables of type enum card suit

```
enum cardsuit play1, play2;
```

Now we can give values to these variables

```
play1=CLUBS;

play2=SPADES;
```

- ✓ Internally the compiler treats the enumerator as integers. Each value on the list of permissible values corresponds to an integer, starting with 0. Thus in the above example CLUBS=0, DIAMONDS=1, HEARTS=2, SPADES=3
- ✓ This way of assigning numbers can be overridden by the programmer by initializing the enumerators to different integer values as shown below

```
enum cardsuit
{
        CLUBS=10, DIAMONDS=12, HEARTS=13, SPADES=15
};
```

## UNIONS:

- Unions are derived data types like structures.
- Both structures and unions are used to group a number of different variables together.
- A structure enables us treat a number of different variables stored at different places in memory.
- A union enables us to treat the same space in memory as a number of different variables.
- A union is a collection of variables of different types, us like a structure. However with unions, you can only store information in one field at one time.

**Features/usage**

- Unions access individual bytes of larger type.
- Sharing an area to save storage usage.
- Unions not used nearly as much as structures.
- Size of unions is size of its biggest member.
- Unions most often contain different types if structures.

- Can only initialize first member of union.
- Functions can return union, type.
- Accepting members by using dot (.) for variables and (→) for pointers variables.
- Can align one variables of union to another.
- Can only initialize first member of union.

Ex:-

```
main ( )
{
        union new
        {
                int x;
                float k;
        };
        union new a;
        a.x=3;
        printf (“x=%d\n”,a.x);
        a.k=4.28;
        printf (“k=%f\n”,a.k);
        getch ( );
}
```

Output:

X=3

K=4.280000

**Unions in structures:**

Just as one structure can be nested within another. A union too can be nested in another union. Not only that, there can be a union in a structure, or a structure in a union.

```
# include<stdio.h>
void main ( )
{
        struct one
```

```
{
        int i;
        char c [2];
};
struct two
{
        int j;
        char x [2];
};
union a
{
        struct one k;
        struct two s;
};
union a f;
        f. k. i=250;
        f. s. x[0]='A';
        f. s. x[1]='B';
printf("%d", f.k.i);
printf("%d", f.s.j);
printf("%c", f.k.c[0]);
printf("%c", f.k.c[1]);
printf("%c", f.s.x[0]);
printf("%c", f.s.x[1]);
getch ( );
}
```