What is a structure?

A structure is a user defined data type in C. A structure creates a data type that can be used to group items of possibly different types into a single type.

- Structure is a way to group variables.
- Structure is a collection of dissimilar elements.
- Defining structure means creating new data types.
- A structure contains a number of data types grouped together. These data types may or may not be of the same type.
- Defining a structure: In general it is defined with the syntax name **struct** as follows

```
struct structure_name
{
        Data type variable1;
        Data type variable2;
}
```

- ***How to create a structure?*** (Declaring a Structure)
  'struct' keyword is used to create a structure. Following is an example.

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

This statement defines a new data type called **struct book**.
It consist of a character variable called **name**,
a float variable called **price** and
an integer variable called **pages**.

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 ;
```

Structure initialization

How to initialize structure members?

Structure members **cannot be** initialized with declaration. For example the following C program fails in compilation.

```
struct Point
{
    int x = 0; // COMPILER ERROR: cannot initialize members here
    int y = 0; // COMPILER ERROR: cannot initialize members here
};
```

Like any other variable or array a structure variable can also be initialized:

```
Struct record
{
        char name[30];
        int age;
        int weight;
}
```

Static struct record student1={“rajan”, 18, 62}

Here student1 is of record structure and the name,age and weight are initialised as “rajan”, 18 and 62 respectively.

Another example:

```
struct book
    {
        char name[10] ;
        float price ;
        int pages ;
    } ;
    struct book b1 = {"Basic", 130.00, 550 }; // Structure members can
    be initialized using curly braces
    struct book b2 = {"Physics", 150.80, 800 };
```

Another example:

```
struct Point
    {
        int x, y;
    };

    int main()
    {
        struct Point p1 = {0, 1};
    }
```

// A valid initialization. member x gets value 0 and y gets value 1. The order of declaration is followed.

Note: In C++, the struct keyword is optional before in declaration of a variable. In C, it is mandatory.

- ***How to access structure elements?***
  Structure members are accessed using dot (.) operator.

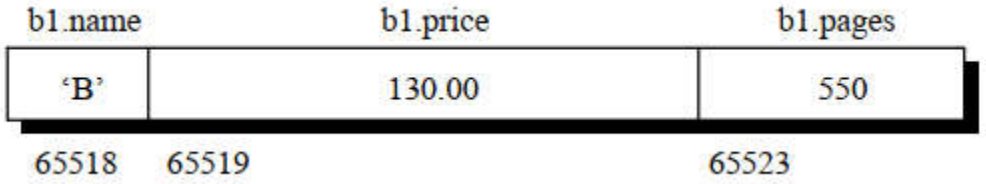
```
#include<stdio.h>
struct Point
    {
        int x, y;
    };

int main()
{
    struct Point p1 = {0, 1};
    p1.x = 20;      // Accessing members of point p1
    printf ("x = %d, y = %d", p1.x, p1.y);
    return 0;
}
```

- **Accessing Structure Elements**
  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.

- 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

- Write a c program to read biodata of students showing name, place, pin, phoneNo and grade.

structure1.c

```
#include<stdio.h>
void main()
{   struct biodata
    {       char name[30];
            char place[40];
            int pin;
            long int phone;
            char grade;
    };
struct biodata student[50];
int i,n;
printf("\n no of students");
scanf("%d",&n);
for(i=0;i<n;i++)  {
scanf("%s",student[i].name);
scanf("%s",student[i].place);
scanf("%d",student[i].pin);
scanf("%ld",student[i].phone);
scanf("%c",student[i].grade);            }
}
```

- **Additional Features of Structures**

  (a) The values of a structure variable can be assigned to another structure variable of the same type using the assignment operator.

  (b) One structure can be nested within another structure. Using this facility complex data types can be created.

  (c) Like an ordinary variable, a structure variable can also be passed to a function.

  (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'.

copy structure.c

- The values of a structure variable can be assigned to another structure variable of the same type using the assignment operator.

```
main( )
{
        struct employee
        {
        char name[10] ;
        int age ;
        float salary ;
        } ;
                struct employee e1 = { "Sanjay", 30, 5500.50 } ;
                struct employee e2, e3 ;
                strcpy ( e2.name, e1.name ) ;    /* piece-meal copying */
                e2.age = e1.age ;
                e2.salary = e1.salary ;
                e3 = e2 ;                        /* copying all elements at one go */
        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 ) ;
}
```

nested structure.c

```
main( )
{
struct address
{
char phone[15] ;
char city[25] ;
int pin ;
} ;
struct emp
{
char name[25] ;
struct address a ;
} ;
struct emp e = { "amit", "9411776858", "dehradun",
248001 };
printf ( "\n name = %s phone = %s", e.name,
e.a.phone ) ;
printf ( "\n city = %s pin = %d", e.a.city, e.a.pin ) ;
}
```

- One structure can be nested within another structure. Using this facility complex data types can be created.

Output:

```
name = amit          phone = 9411776858
city = dehradun      pin = 248001
-------------------------------
```

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

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

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

- Like an ordinary variable, a structure variable can also be passed to a function.

```
/* Passing structure variable to a function */
        struct book
        {
        char name[25] ;
        char author[25] ;
        int callno ;
        } ;
                main( )
                {
                struct book b1 = { "programming in C", "gehu", 101 } ;
                display ( b1 ) ;
                }

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

- It can be realized that to pass individual elements would become more tedious. A better way would be to pass the entire structure variable at a time.

```
/* program of a structure pointer. */

main( )
{
        struct book
        {
        char name[25] ;
        char author[25] ;
        int callno ;
        } ;
        struct book b1 = { "programming in C", "gehu", 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 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'.

- ***What is an array of structures?***
  Like other primitive data types, we can create an array of structures.

```
#include<stdio.h>
struct Point
{
    int x, y;
};

int main()
{
struct Point arr[10];           // Create an array of structures
arr[0].x = 10;                  // Access array members
arr[0].y = 20;

printf("%d %d", arr[0].x, arr[0].y);
return 0;
}
```

Output:`10 20`

```
/* Usage of an array of structures */
main( )
{
    struct book
    {
    char name[10] ;
    float price ;
    int pages ;
    };
    struct book b[2];
    int i ;
            for ( i = 0 ; i<2 ; i++ )
            {
            printf ( "\nEnter name, price and pages " ) ;
            scanf ( "%s %f %d", &b[i].name, &b[i].price, &b[i].pages ) ;
            }
    for ( i = 0 ; i <2 ; i++ )
    printf ( "\n%s %f %d", b[i].name, b[i].price, b[i].pages ) ;
}
```

array of structures.c

What is a structure pointer?

Like primitive types, we can have pointer to a structure. If we have a pointer to structure, members are accessed using arrow (->) operator.

```
#include<stdio.h>
struct Point
{
    int x, y;
};

int main()
{
struct Point p1 = {1, 2};
struct Point *p2 = &p1;                 // p2 is a pointer to structure p1
printf("%d %d", p2->x, p2->y);          // Accessing structure members using
                                            structure pointer
return 0;
}
```

Output: `1 2`

- **Limitations of C Structures**
- In C language, Structures provide a method for packing together data of different types. A Structure is a helpful tool to handle a group of logically related data items. However, C structures have some limitations.
- The C structure does not allow the struct data type to be treated like built-in data types:We cannot use operators like +,- etc. on Structure variables.

For example, consider the following code:

```
struct number
{
    float x;
};
int main()
{
    struct number n1,n2,n3;
    n1.x=4;
    n2.x=3;
    n3=n1+n2;
    return 0;
}
/*Output:
prog.c: In function 'main':
prog.c:10:7: error:
invalid operands to binary + (have 'struct number' and 'struct number')
n3=n1+n2;     */
```

Union in C

- Like Structures, union is a user defined data type. In union, all members share the same memory location.

```
#include <stdio.h>
union test              // Declaration of union is same as structures
{
    int x, y;
};

int main()
{
    union test t;
    t.x = 2;
    printf("x=%d, y=%d\n", t.x,t.y);
    t.y = 10;
    printf("x=%d, y=%d\n", t.x,t.y);
    return 0;
}
```

OUTPUT:

```
x=2, y=2
x=10, y=10
----------------------------
```

Difference between Structure and Union in C

A structure is a user-defined data type available in C that allows to combining data items of different kinds. Structures are used to represent a record.

Defining a structure: To define a structure, you must use the **struct** statement. The struct statement defines a new data type, with more than or equal to one member.

```
struct [structure name]
{
        member definition;
        member definition;
        member definition;
};
```

- A union is a special data type available in C that allows storing different data types in the same memory location.
- You can define a union with many members, but only one member can contain a value at any given time.
- Unions provide an efficient way of using the same memory location for multiple purposes.

Defining a Union: To define a union, you must use the **union** statement in the same way as you did while defining a structure.

The union statement defines a new data type with more than one member for your program.

- The format of the union statement is as follows:

```
union [union name]
{
    member definition;
    member definition;
    member definition;
};
```

- **Similarities between Structure and Union**
- Both are user-defined data types used to store data of different types as a single unit.
- Their members can be objects of any type, including other structures and unions or arrays. A member can also consist of a bit field.
- Both structures and unions support only assignment = and sizeof operators. The two structures or unions in the assignment must have the same members and member types.
- A structure or a union can be passed by value to functions and returned by value by functions. The argument must have the same type as the function parameter. A structure or union is passed by value just like a scalar variable as a corresponding parameter.
- **'.'** operator is used for accessing members.

Differences

	STRUCTURE	UNION
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is **greater than or equal to the sum of sizes of its members.**	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of **union is equal to the size of largest member.**
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.