Hope all of you can remember that

Array can **store data items** of **same data type** only.

So how can we store **data items** of **different data types** under a single name?

Answer is by using C **structure**.

Let us now start our discussion on C structure.

What is C Structure?

- ✓ It is a data type.
- ✓ Structure is usually used for combining a group of related data items having different data types under a single name.
- ✓ For example, it can be used to represent a set of attributes such as roll number, name, total marks, attendance percentage of a student entity.

<u>How to define/declare a structure</u>?

Let us declare a structure to hold the above mentioned information of students i.e. roll number, name, total marks, attendance percentage.

This is a **keyword and is mandatory**. It introduces a **structure declaration**, which is a list of declarations enclosed in braces.

A **structure tag**, which is an optional name that identifies structures of this type, that is, structures having this combination.

```
struct student
{
      int     roll_no;
      char    name[30];
      int     tot_marks;
      float   attn_per;
};
```

Members/elements of structure

Above declaration tells

- "the structure is named **student"** (i.e. **structure tag** is **student**).
- The variables used in a structure are called members.
- It contains 4 members – "roll_no" (an integer quantity), "name" (a string), "tot_marks" (another integer quantity) and "attn_per" (a floating point quantity).

NOTE CAREFULLY and REMEMBER

- The members can be pointers or other structures.
- **Members of a structure do not occupy any memory space until they are associated with structure variables**.
- So after the above declaration, the 4 members of the structure of type **student** do not occupy any memory space.

<u>How to declare structure variables</u>?

Method – 1: Structure tag is used to declare structure variables of that type.

```
struct  student
{
        int roll_no;
        char name[30];
        int tot_marks;
        float attn_per;
};
struct  student  s1, s2;
```

Here, structure tag **"student"** is used to declare variables of that type. Hence s1 and s2 are variables of type **struct student**.

Method – 2: In absence of Structure tag, structure variables are declared along with structure definition (declaration).

```
struct
{
        int roll_no;
        char name[30];
        int tot_marks;
        float attn_per;
} s1, s2;
```

Here, structure tag is omitted and hence structure variable s1, s2 are declared along with the structure declaration.

NOTE

- When we declare a structure variable then memory space is allocated for each of its members.
- The structure elements are always arranged in contiguous memory locations.

How to initialize structure variables?

Case 1: When structure tag is mentioned in declaration of a structure.

struct student s1 = { 1, “Amit”, 890, 91.5 };

Case 2: <u>When structure tag is omitted in declaration of a structure</u>.

```
struct
{
      int roll_no;
      char name[30];
      int tot_marks;
      float attn_per;
} s1 = { 1, "Amit", 890, 91.5 };
```

<u>How to access any member of a structure variable</u>?

1. We can access any member of a structure variable by a **dot (.) operator** as follows:

   s1**.**roll_no

   where **s1** is <u>structure variable</u> and **roll_no** is a <u>member of structure</u>.

2. We can also use arrow (→) operator for accessing members of a structure variables by pointer.

   ptr → roll_no

   where **ptr** is a <u>pointer to structure variable</u> and **roll_no** is a <u>member</u>.

```
struct student
{
      int  roll_no;
      char grade;
};
int main()
{
   struct student s = {1, 'A'}, *ptr;
   ptr=&s;
   printf("\n Roll No: %d Grade: %c", s.roll_no, s.grade);
   printf("\n Roll No: %d Grade: %c", ptr->roll_no, ptr->grade);
  return(0);
}
Output:
Roll No: 1 Grade: A
Roll No: 1 Grade: A
```

User-defined Data Types (typedef)

The typedef feature allows users to define a new data type that is equivalent to some existing data type. Once a user-defined data type is established, and then new variables, arrays, structures etc. can be declared in terms of this new data type.

```
Example 1:
typedef  int age;
age  a, b;
Example 2:
typedef struct
 {
      int roll_no;
      char grade;
} record;
record   s1, s2;
```

Some Features of STRUCTURE are discussed below with examples:

1. We can assign the values of a structure variable to another structure variable of the same type using the assignment operator. However, member-wise assignment is also possible.

   We can pass a **structure variable to a function**.

   We can **have a pointer to a structure**.

   Example:

```
#include <stdio.h>
#include <string.h>

struct student
{
   char name[30];
   float per;
};
```

```
int main()
{
   void display1(struct student);             //display1 function declaration
   void display2(struct student *);           //display2 function declaration

   struct student s1 = {"Ajoy Das",89.2};  //s1 is declared & initialized
   struct student s2, s3;                 //Structure variables s2 & s3 declared

   /* Member-wise assignments from structure variable s1 to structure variable s2  */

   strcpy(s2.name, s1.name);
   s2.per = s1.per;

   /*Assignment of values from structure variable s2 to structure variable s3 */

   s3 = s2;

 /* display1 function is called with structure variable s2 as argument */

   display1(s2);

/* display2 function is called with a pointer to s3 as argument */

   display2(&s3);
   return(0);
}

void display1(struct student s)
{
   printf("\n Name       : %s", s.name);   //name member is accessed by dot operator
   printf("\n Percentage : %6.2f", s.per); //per member is accessed by dot operator
}

void display2(struct student *ps)
{
   printf("\n Name      :  %s", ps–>name);    //name member is accessed by arrow operator
   printf("\n Percentage : %6.2f", ps–>per); //per member is accessed by arrow operator
}
```

2. One structure can be nested within another.

Example:

```
#include <stdio.h>
#include <string.h>

struct address
{
   char street[20];
   char city[10];
   long int pin;
};
struct emp
{
   char name[30];
   struct address a;          //address structure is nested within emp structure
};

int main()
{
   struct emp e={"Ajoy Das","12 Hakimpara","Siliguri",734401};
   printf("\n Name      : %s", e.name);
   printf("\n Street    : %s", e.a.street);
   printf("\n City      : %s", e.a.city);
   printf("\n Pin       : %ld", e.a.pin);
   return(0);
}
```

[NOTE: To access **name** member using a structure variable e of type **struct emp**, we have used **dot** operator and written e**.**name .
Now **observe carefully** how members (street, city, pin) of **address** structure is accessed using structure variable e of type **struct emp**.]

3. We can use an array of structures.

[We use structure to group together related data items, possibly of different data type, under a single name. For example, **struct student** is used to group roll, name and marks of a student. Now declaring a structure variable we can store values of its members for a student. NOW if we want to store values of roll, name & marks for 3 students, we need to declare 3 structure variables.

BUT if it is to be done for 100 students!!

We have to **declare** an **array of structure** where **each element of the array** will represent **a structure variable**.]

```
#include <stdio.h>
#include <string.h>
struct student
{
        int roll;
        char name[20];
        int marks;
};

int main()
{
        int i;
        struct student s[3];                //s is an array of struct student
        for(i = 0; i < 3; i++)
        {
                printf("\n Enter roll no : ");
                scanf("%d", &s[i].roll);
                printf("\n Enter Name of student : ");
                gets(s[i].name);
                printf("\n Enter Marks : ");
                scanf("%d", &s[i].marks);
        }
        for(i = 0;i < 3;i++)
        {
                printf("\n%d\t%s\t%d",  s[i].roll,  s[i].name,  s[i].marks);
        }
        return(0);
 }
```

Compare array and structure:

Array	Structure
An array is a collection of data items of **same** data type.	A structure is a collection of data items of **different** data types.
Array declaration is simple.	Structure declaration is complicated than array declaration since a structure must be defined in terms of its individual members.
There is no keyword.	The keyword **struct** is used.
An array name represents the base address i.e. the address of the first element of the array.	A **structure name** is known as **tag name**. It is used to declare a structure variable of its type.

Example –
An array declaration
int n[10];
means an array n can hold 10 integer data items.
	Example –
	A structure declaration
	struct account {
	int acc_no;
	float balance;
	};
	means structure is name account with 2 members – integer quantity(acc_no) & floating-point quantity (balance).

Unions:

A **union** may contain many members of different data types, but only one member may be stored at a time in a union variable.

Example: Consider the following declaration

```
union item
{       int x;
        float y;
        char z;
} code;
```

Here **code** is a **union variable** of type **union item**. The union contains three members, each with a different data type. However we can use only one of them at a time. This is due to the fact that only one location is allocated for a union variable, irrespective of its size. In this declaration, maximum space required by member **y** and it is 4 bytes in case of Turbo C compiler. So **size** of union variable **code** will be 4 bytes so that it can hold an integer quantity (requires 2 bytes) or a floating point number (requires 4 bytes) or a character value (requires 1 byte) at any one time.

[NOTE: Unions follow the <u>same syntax as structure</u>.]

Compare structure and Union:

Structure	**Union**
Every member has its own storage location.	All members use the same location.
Keyword **struct** is used.	Keyword **union** is used.
All members may be initialized.	Only its first member may be initialized. Other members can be initialized by either assigning values or reading from the keyboard.
Different interpretations of the same memory location are not possible.	Different interpretations of the same memory location are possible.
Consumes more space compared to union.	Conservation of memory is possible.
All members are active at a time.	Only one member is active at a time.

Example –
struct clothes
{
    char colour[15];
    int size;
} shirt;
Here **shirt** is a **structure variable** of type **struct clothes**.
Assuming integer requires 2 bytes & char requires 1 byte, the size of structure variable shirt is 15 + 2 = 17 bytes.
	Example –
	union clothes
	{
	    char colour[15];
	    int size;
	} shirt;
	Here **shirt** is a **union variable** of type **union clothes**.
	Assuming integer requires 2 bytes & char requires 1 byte, the size of union variable shirt is 15 bytes.

Review Question 1:

Consider the following definition of union and structure.

```
union result
{
    int marks;
    char grade;
};
```

```
struct  res
{
    char name[15];
    int age;
    union result r;
}data;
```

Assuming integer requires 2 bytes & character requires 1 byte, state the output of the following statements:

```
printf("\n size of union = %d", sizeof(data.r));
printf("\n size of structure = %d", sizeof(data));
```

==

Enumeration:

- It is a data type similar to structure or a union.
- Its members are constants that are written as identifiers.
- These constants represent values that can be assigned to corresponding enumeration variables.

Example:

We can define an enumeration data type color as follows:

enum color { black, blue, cyan, red, green};

where **enum** is a required keyword.

color is a name of enumerated data type

black, **blue**, … , **green** are the individual identifiers that may be assigned to variables of this type.

We can declare enumeration variables **foreground** and **background** of type **color** as follows:

enum color foreground, background;

The compiler automatically assigns 0 to first constant, 1 to second constants, and so on **unless explicitly specified**.

So by the above definition, black has a value 0, blue has value 1, cyan 2, …..green 4. However we can define it as follows:

enum color { black =−1, blue, cyan, red, green};

So here black has a value −1, blue 0, ……, green has a value 3.

<u>Example Program</u>: Write a C program using enum data type.

```
#include <stdio.h>
#include <string.h>

enum day {sun, mon, tue, wed, thu, fri, sat};
struct emp
{ char name[20];
  enum day d;
};
```

```
int main()
{
  struct emp e1,e2;
  strcpy(e1.name, "Sachin");
  strcpy(e2.name, "Sourav");
  e1.d=tue;
  e2.d=sat;
  printf("\n%s takes off on day %d", e1.name, e1.d);
  printf("\n%s takes off on day %d", e2.name, e2.d);
  return(0);
}
```

Output :
Sachin takes off on day 2
Sourav takes off on day 6

Advantages:

a) It provides a way to associate constant values with names like **#define**.
b) It has an added advantage over **#define** that the constant values can be automatically generated.

Disadvantage:
There is no way to use the enumerated values directly in input/output functions like printf() and scanf().