Structure

A structure is a collection of variables under a single name. These variables can be of different types, and each has a name which is used to select it from the structure. A structure is a convenient way of grouping several pieces of related information together.

Difference between C variable, C array and C structure:

- A normal C variable can hold only one data of one data type at a time.
- An array can hold group of data of same data type.
- A structure can hold group of data of different data types
- Data types can be int, char, float, double and long double etc.

Accessing members of a structure

There are two types of operators used for accessing members of a structure.

1. Member operator(.)
2. Structure pointer operator(->)

To access any member of a structure, we use the **member access operator (.)**.

Syntax

```
struct structure_name
{
        data_type member1;
        data_type member2;
        data_type member3;
        ..
        ..
        data_type membern;
};
```

Inside main function

```
struct structure_name variable_list;
```

Another way of creating structure variable is:

```
struct structure_name
{
        data_type member1;
        data_type member2;
        data_type member3;
        ..
        ..
        data_type membern;
}variable_list;
```

E.g

```
struct person
{
        char name[50];
        int id;
        float salary;
};
```

Inside main function:

```
struct person a,b,c[20];
```

or

```
struct person
{
        char name[50];
        int id;
        float salary;
}a,b,c[20];
```

In both cases 2 variable a,b and array c having 20 elements of type struct person are created.

E.g.

```
#include <stdio.h>
#include<conio.h>
struct distance
{
   int feet;
   float inch;
}d1,d2,sum;
int main()
{
        printf("1st distance\n");
        printf("Enter feet: ");
        scanf("%d",&d1.feet);  /* input of feet for structure variable d1 */
        printf("Enter inch: ");
        scanf("%f",&d1.inch);  /* input of inch for structure variable d1 */
        printf("2nd distance\n");
        printf("Enter feet: ");
        scanf("%d",&d2.feet);  /* input of feet for structure variable d2 */
        printf("Enter inch: ");
        scanf("%f",&d2.inch);  /* input of inch for structure variable d2 */

        sum.feet=d1.feet+d2.feet;
        sum.inch=d1.inch+d2.inch;

        printf("Sum of distances=%d %f",sum.feet,sum.inch); /* printing sum of distance d1
        and d2 */
        getch();
        return 0;
}
```

Passing structure to function in C:

It can be done in below 3 ways.

1. Passing structure to a function by value
2. Passing structure to a function by address(reference)
3. No need to pass a structure – Declare structure variable as global

Passing structure to function in C by value

```
struct student
{
	int id;
	char name[20];
	float percentage;
};

void func(struct student record);

int main()
{
	struct student record;

	record.id=1;
	strcpy(record.name, "Raju");
	record.percentage = 86.5;

	func(record);
	return 0;
}
```

```
void func(struct student record)
{
        printf(" Id is: %d \n", record.id);
        printf(" Name is: %s \n", record.name);
        printf(" Percentage is: %f \n", record.percentage);
}
```

Passing structure to function in C by address

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

void func(struct student *abc);

int main()
{
        struct student record;

        record.id=1;
        strcpy(record.name, "Raju");
        record.percentage = 86.5;

        func(&record);
        return 0;
}
```

```
void func(struct student *pqr)
{
	printf(" Id is: %d \n", pqr->id);
	printf(" Name is: %s \n", pqr->name);
	printf(" Percentage is: %f \n", pqr->percentage);
}
```

Example program to declare a structure variable as global in C:

```
#include<stdio.h>
#include<conio.h>
struct abc
	{
		char name[100];
		float price;
		int pages;
	};
struct abc b[5];
int main()
{
	int i;
	for(i=0;i<2;i++)
	{
		printf("\n Enter name, price and pages ");
		scanf("%s%f%d",b[i].name,&b[i].price,&b[i].pages);
	}
	abc();
	getch();
	return 0;
}
```

```
abc()
{
        int i;
        for(i=0;i<2;i++)
        {
                printf("%s%f%d",b[i].name,b[i].price,b[i].pages);
        }
        return 0;
}
```

Some More Examples

Source Code to Store Information of 10 students

```
#include <stdio.h>
struct student{
   char name[50];
   int roll;
   float marks;
};
int main(){
   struct student s[10];
   int i;
   printf("Enter information of students:\n");
   for(i=0;i<10;++i)
   {
      s[i].roll=i+1;
      printf("\nFor roll number %d\n",s[i].roll);
      printf("Enter name: ");
      scanf("%s",s[i].name);
      printf("Enter marks: ");
```

```
        scanf("%f",&s[i].marks);
        printf("\n");
    }
    printf("Displaying information of students:\n\n");
    for(i=0;i<10;++i)
    {
     printf("\nInformation for roll number %d:\n",i+1);
     printf("Name: ");
     puts(s[i].name);
     printf("Marks: %.1f",s[i].marks);
    }
   return 0;
}
```

E.g.

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

struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main( )
{
   struct Books Book1;        /* Declare Book1 of type Book */
   struct Books Book2;        /* Declare Book2 of type Book */
```

```
   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* book 2 specification */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* print Book1 info */
   printf( "Book 1 title : %s\n", Book1.title);
   printf( "Book 1 author : %s\n", Book1.author);
   printf( "Book 1 subject : %s\n", Book1.subject);
   printf( "Book 1 book_id : %d\n", Book1.book_id);

   /* print Book2 info */
   printf( "Book 2 title : %s\n", Book2.title);
   printf( "Book 2 author : %s\n", Book2.author);
   printf( "Book 2 subject : %s\n", Book2.subject);
   printf( "Book 2 book_id : %d\n", Book2.book_id);
   return 0;
}
```

E.g.

```
struct abc
	{
		char name[100];
		float price;
		int pages;
	};
	struct abc b[2];
	abc(struct abc s);
int main()
{
	int i;
	for(i=0;i<2;i++)
	{
		printf("\n Enter name, price and pages ");
		scanf("%s%f%d",b[i].name,&b[i].price,&b[i].pages);
	}
	xyz(b);
	getch();
	return 0;
}
xyz(struct abc st)
{
	int i;
	for(i=0;i<2;i++)
	{
		printf("%s%f%d",b[i].name,b[i].price,b[i].pages);
	}
	return 0;
}
```

Union

A **union** is a special data type available in C that enables you to store 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.

- Union and structure in C are same in concepts, except allocating memory for their members.
- Structure allocates storage space for all its members separately.
- Whereas, Union allocates one common storage space for all its members. So that Unions provide an efficient way of using the same memory location for multi-purpose.
- We can access only one member of union at a time. We can't access all member values at the same time in union. But, structure can access all member values at the same time. This is because, Union allocates one common storage space for all its members. Whereas Structure allocates storage space for all its members separately.
- Many union variables can be created in a program and memory will be allocated for each union variable separately.

E.g.

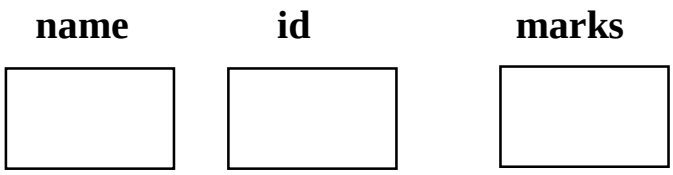
```
union Data
{
  int id;
  float marks;
  char  name[20];
};

int main( )
{
  union Data abc;
  printf( "Memory size occupied by data : %d\n", sizeof(abc));
  return 0;
}
```

Now, a variable of **Data** type can store an integer, a floating-point number, or a string of characters. This means that a single variable i.e. same memory location can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in above example Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by character string. Following is the example which will display total memory size occupied by the above union:

When the above code is compiled and executed, it produces the following result:

Memory size occupied by data : 20



20 Bytes **2 Bytes** **4 Bytes**

Memory Allocation in case of structure

20 Bytes

Memory Allocation in case of Union

E.g.

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

union Data
{
   int i;
   float f;
   char  str[20];
};

int main( )
```

```
{
   union Data data;

   data.i = 10;
   data.f = 220.5;
   strcpy(data.str, "C Programming");
   char s;

   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

Output:

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

Here values of **i** and **f** members of union gets corrupted because final value assigned to the variable has occupied the memory location and this is the reason that the value if **str** member is getting printed very well.

Now let's look into the same example once again where we will use one variable at a time which is the main purpose of having union:

```
#include <stdio.h>
#include <string.h>
union Data
{
   int i;
   float f;
```

```
   char  str[20];
};
int main( )
{
   union Data data;

   data.i = 10;
   printf( "data.i : %d\n", data.i);
   data.f = 220.5;
   printf( "data.f : %f\n", data.f);

   strcpy( data.str, "C Programming");
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

Here, all the members are getting printed very well because one member is being used at a time.