Structure

Basics of Structure

An array is a collection of data items, all having the same data type and accessed using a common name and an integer index into the collection.

A *struct* is also a collection of data items, except with a struct the data items can have different data types, and the individual *fields* within the struct are accessed *by name* instead of an integer index.

Structs are very powerful for bundling together data items that collectively describe a thing, or are in some other way related to each other.

For example: You want to store some information about a person: his/her name, citizenship number and salary. You can easily create different variables *name, citNo, salary* to store these information separately.

However, in the future, you would want to store information about multiple persons. Now, you'd need to create different variables for each information per person: *name1, citNo1, salary1, name2, citNo2, salary2*

You can easily visualize how big and messy the code would look. Also, since no relation between the variables (information) would exist, it's going to be a daunting task.

A better approach will be to have a collection of all related information under a single name Person, and use it for every person. Now, the code looks much cleaner, readable and efficient as well.

This collection of all related information under a single name Person is a structure.

Structure Definition in C

Syntax of structure:

```
struct structure_name
{
   data_type member1;
   data_type member2;
   .
   .
   data_type memeber;
};
```

struct is a keyword.

structure_name is a tag name of a structure.

member1, member2 are members of structure.

Example:

```
struct person

{
   char name[50];
   int citNo;
   float salary;
};
```

Structure variable declaration

When a structure is defined, it creates a user-defined type but, no storage or memory is allocated.

For the above structure of a person, variable can be declared as:

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

int main()
{
   struct person person1, person2, person3[20];
   return 0;
}
```

Another way of creating a structure variable is:

```
struct person
{
   char name[50];
   int citNo;
   float salary;
} person1, person2, person3[20];
```

Accessing members of a structure

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

Member operator(.)

Any member of a structure can be accessed as:

```
structure_variable_name.member_name
```

Suppose, we want to access salary for variable *person2*. Then, it can be accessed as:

```
person2.salary
```

Example

```
#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;
}
```

Output

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

Nested Structures

Structs can be nested, either with previously defined structs or with new internally defined structs. In the latter case the struct names may not be necessary, but scoping rules still apply. (I.e. if a new struct type is created inside another struct, then the definition is only known within that struct.) For example, in the following code the Date struct is defined independently to the Exam struct, but the score and time structs are definied internally to the Exam struct.

```
struct Date {
    int day, month, year; };

struct Exam {
    int room, nStudents;

    struct Date date;

    struct {
        int hour, minute;
        bool AM;
    } time;

    typedef struct Score{
        int part1, part2, total; } Score;

    Score scores[ 100 ];
};
```

Array of Structures

C Structure is collection of different datatypes (variables) which are grouped together. Whereas, array of structures is nothing but collection of structures. This is also called as structure array in C.

Example

```
#include <stdio.h>

#include <string.h>

struct student

{
    int id;
    char name[30];
    float percentage;
};
int main()
{
    int i;
    struct student record[2];
    // 1st student's record
    record[0].id=1;
    strcpy(record[0].name, "Raju");
    record[0].percentage = 86.5;

    // 2nd student's record
    record[1].id=2;
    strcpy(record[1].name, "Surendren");
    record[1].percentage = 90.5;

    // 3rd student's record
    record[2].id=3;
    strcpy(record[2].name, "Thiyagu");
    record[2].percentage = 81.5;

    for(i=0; i<3; i++)
    {
        printf("Records of STUDENT : %d \n", i+1);
        printf(" Id is: %d \n", record[i].id);
        printf(" Name is: %s \n", record[i].name);
        printf(" Percentage is: %f\n\n",record[i].percentage);
    }
    return 0;
```

```
}
```

Output

```
Records of STUDENT : 1
Id is: 1
Name is: Raju
Percentage is: 86.500000
Records of STUDENT : 2
Id is: 2
Name is: Surendren
Percentage is: 90.500000
Records of STUDENT : 3
Id is: 3
Name is: Thiyagu
Percentage is: 81.500000
```

Structure and Functions

A structure can be passed to any function from main function or from any sub function. Structure definition will be available within the function only.

It won't be available to other functions unless it is passed to those functions by value or by address(reference).

Example

```
#include <stdio.h>

#include <string.h>

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);

}
```

Output

```
Id is: 1
Name is: Raju
Percentage is: 86.500000
```

Structure and Pointers

```
#include <stdio.h>

#include <string.h>

struct student

{

    int id;

    char name[30];

    float percentage;

};

int main()

{

    int i;

    struct student record1 = {1, "Raju", 90.5};

    struct student *ptr;

    ptr = &record1;

    printf("Records of STUDENT1: \n");

    printf("  Id is: %d \n", ptr->id);
```

```
      printf("  Name is: %s \n", ptr->name);

        printf("  Percentage is: %f \n\n", ptr->percentage);

       return 0;

}
```

Output

```
Records of STUDENT1:
Id is: 1
Name is: Raju
Percentage is: 90.500000
```

Union

A **union** is a special data type available in C that allows 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. Unions provide an efficient way of using the same memory location for multiple-purpose.

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 tag] {
   member definition;
   member definition;
   ...
   member definition;
} [one or more union variables];
```

The **union tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional. Here is the way you would define a union type named Data having three members i, f, and str −

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

Now, a variable of **Data** type can store an integer, a floating-point number, or a string of characters. It means 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.

Difference between structure and union

Structure	**Union**
1.The keyword **struct** is used to define a structure	1. The keyword **union** is used to define a union.
2. 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. The smaller members may end with unused slack bytes.	2. 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.
3. Each member within a structure is assigned unique storage area of location.	3. Memory allocated is shared by individual members of union.
4. The address of each member will be in ascending order This indicates that memory for each member will start at different offset values.	4. The address is same for all the members of a union. This indicates that every member begins at the same offset value.
5 Altering the value of a member will not affect other members of the structure.	5. Altering the value of any of the member will alter other member values.
6. Individual member can be accessed at a time	6. Only one member can be accessed at a time.
7. Several members of a structure can initialize at once.	7. Only the first member of a union can be initialized.