Structure: -

The **structure** in C is a user-defined data type that can be used to group items of possibly different types into a single type. The **struct keyword** is used to define the structure in the C programming language. The items in the structure are called its **member** and they can be of any valid data type.

Structures (also called structs) are a way to group several related variables into one place. Each variable in the structure is known as a member of the structure. Unlike an array, a structure can contain many different data types (int, float, char, etc.).

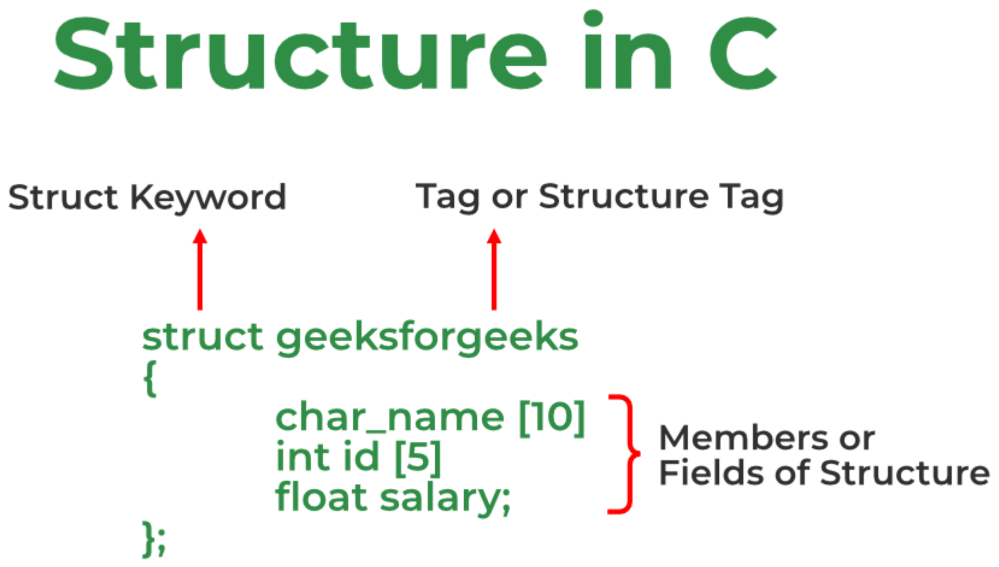
- **a structure is a collection of elements of the different data type.**
- **The structure is used to create user-defined data type in the C programming language. As the structure used to create a user-defined data type, the structure is also said to be "user-defined data type in C".**
- **In other words, a structure is a collection of non-homogeneous elements. Using structure we can define new data types called user-defined data types that holds multiple values of the different data type. The formal definition of structure is as follows...**

Structure is a collection of different type of elements under a single name that acts as user defined data type in C.

Generally, structures are used to define a record in the c programming language. Structures allow us to combine elements of a different data type into a group. The elements that are defined in a structure are called members of structure.

A **struct** is a keyword that creates a user-defined datatype in the C programming language. A **struct** keyword creates datatypes that can be used to group items of possibly different types and same types into a single datatype.

For example, if an administration wants to create a file for their students to handle their information, they can construct a structure by using the struct keyword because as we know that a student's name is a character value and its roll number and marks are integer values. Hence, it is a problem to store them in the same datatype, but the struct allows you to do the same using it.



Syntax to construct a structure using struct keyword

```
struct Student{
    int roll_no;
    char name[30];
};
```

Here, the **struct** is a keyword, *Student* is a tag name, *roll_no* and *name* are its members.

Using the **struct** keyword, we can create a workspace for any heterogeneous and homogeneous type of data to store and manipulate at a single place.

Here, we are using **designated initializer** to initialize a structure.

C language code to understand how we can initialize a structure?

```
#include <stdio.h>

// Creating a Student named structure
typedef struct Student {
    // name and roll_no are its member
    char name[20];
    int rollno;
} Student;

int main()
{
    // Declaring two Student type variables
    Student s1, s2;

    // Initializing 1st Student
    s1 = (Student){.name = "shubh", .rollno = 1 };
    // Initializing 2nd  Student
    s2 = (Student){.name = "pachori", .rollno = 2 };

    printf("\n%d %s\n", s1.rollno, s1.name);
    printf("\n%d %s\n", s2.rollno, s2.name);

    return 0;
}
```

Output:

```
1 shubh

2 pachori
```

But if we want to input values by the keyboard then we have to use a different method for it.

Here is an example of how we can input values in a structure using a keyboard go through with the below code.

C language code to understand how we can Access a structure members?

```
#include <stdio.h>


int main()
{
struct Student {
    // name and roll_no are its member
    char name[20];
    int rollno;
} Student;

 // Declaring two Student type variables
    Student s1, s2;

    // Data input through keyboard of s1
    printf("Enter Roll Number First Student: ");
    scanf("%d", &s1.rollno);
    printf("Enter Name First Student: ");
    scanf("%s", s1.name);

    // Data input through keyboard of s2
```

```c
    printf("Enter Roll Number Second Student: ");
    scanf("%d", &s2.rollno);
    printf("Enter Name Second Student: ");
    scanf("%s", s2.name);

    printf("\n%d %s\n", s1.rollno, s1.name);
    printf("\n%d %s\n", s2.rollno, s2.name);

    return 0;
}
```

Output:

```
Enter Roll Number First Student: 101
Enter Name First Student: Subh
Enter Roll Number Second Student: 102
Enter Name Second Student: Pranit

101 Subh

102 Pranit
```

```c
/*C program to read and print employee's record using structure*/

#include <stdio.h>

/*structure declaration*/
struct employee{
    char    name[30];
    int     empId;
    float   salary;
};

int main()
{
    /*declare structure variable*/
    struct employee emp;

    /*read employee details*/
    printf("\nEnter details :\n");
    printf("Name ?:");          gets(emp.name);
    printf("ID ?:");            scanf("%d",&emp.empId);
    printf("Salary ?:");        scanf("%f",&emp.salary);

    /*print employee details*/
    printf("\nEntered detail is:");
    printf("Name: %s"   ,emp.name);
    printf("Id: %d"     ,emp.empId);
    printf("Salary: %f\n",emp.salary);
    return 0;
}
```

Output

```
Enter details :
Name ?:Mike
ID ?:1120
Salary ?:76543

Entered detail is:
Name: Mike
Id: 1120
Salary: 76543.000000
```

HOW STRUCTURE MEMBERS ARE STORED IN MEMORY?

Always, contiguous(adjacent) memory locations are used to store structure members in memory. Consider below example to understand how memory is allocated for structures.

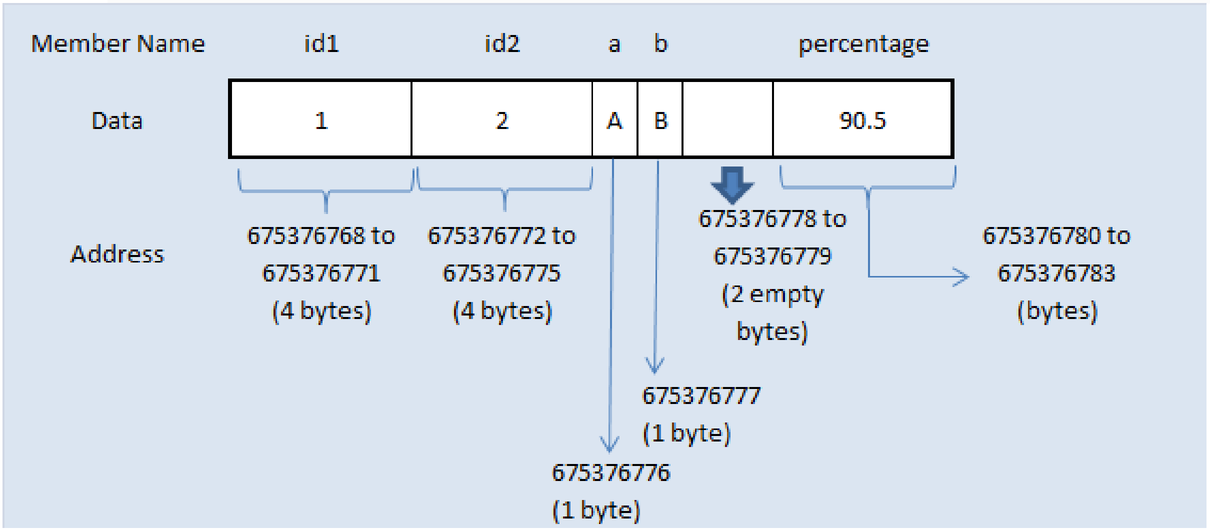
EXAMPLE PROGRAM FOR MEMORY ALLOCATION IN C STRUCTURE:

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

struct student
{
       int id1;
       int id2;
       char a;
       char b;
       float percentage;
};

int main()
{
    int i;
    struct student record1 = {1, 2, 'A', 'B', 90.5};

    printf("size of structure in bytes : %d\n",
                           sizeof(record1));

    printf("\nAddress of id1        = %u", &record1.id1 );
    printf("\nAddress of id2        = %u", &record1.id2 );
    printf("\nAddress of a          = %u", &record1.a );
    printf("\nAddress of b          = %u", &record1.b );
    printf("\nAddress of percentage = %u",&record1.percentage);

    return 0;
}
```

OUTPUT:

```
size of structure in bytes : 16
Address of id1 = 675376768
Address of id2 = 675376772
Address of a = 675376776
Address of b = 675376777
Address of percentage = 675376780
```

- There are 5 members declared for structure in above program. In 32 bit compiler, 4 bytes of memory is occupied by int datatype. 1 byte of memory is occupied by char datatype and 4 bytes of memory is occupied by float datatype.
- Please refer below table to know from where to where memory is allocated for each datatype in contiguous (adjacent) location in memory.



Member Name
id1
id2
a
b
percentage
Data
1
2
A
B
90.5
Address
675376768 to 675376771 (4 bytes)
675376772 to 675376775 (4 bytes)
675376778 to 675376779 (2 empty bytes)
675376780 to 675376783 (bytes)
675376777 (1 byte)
675376776 (1 byte)

Datatype	Memory allocation in C (32 bit compiler)		
	From Address	To Address	Total bytes
int id1	675376768	675376771	4
int id2	**675376772**	**675376775**	**4**
char a	675376776		1
char b	**675376777**		**1**
Addresses 675376778 and 675376779 are left empty. (Do you know why? Please refer below, the next topic C - Structure padding)			2
float percentage	675376780	675376783	4

Array of Structure in C Programming

A structure in C programming language is used to store set of parameters about an object/entity. We sometime want to store multiple such structure variables for hundreds or objects then Array of Structure is used.

Both structure variables and in-built data type gets same treatment in C programming language. C language allows us to create an array of structure variable like we create array of integers or floating point value. The syntax of declaring an array of structure, accessing individual array elements and array indexing is same as any in-built data type array.

Declaration of Structure Array in C

```
struct Employee {
    char name[50];
    int age;
    float salary;
}employees[1000];
or
struct Employee employees[1000];
```

In above declaration, we are declaring an array of 1000 employees where each employee structure contains name, age and salary members. Array employees[0] stores the information of 1st employee, employees[1] stores the information of 2nd employee and so on.

Accessing Structure Fields in Array

We can access individual members of a structure variable as

array_name[index].member_name

For Example

```
employees[5].age
```

C Program to Show the Use of Array of Structures

In below program, we are declaring a structure "employee" to store details of an employee like name, age and salary. Then we declare an array of structure employee named "employees" of size 10, to store details of multiple employees.

```
#include <stdio.h>

struct employee {
 char name[100];
 int age;
 float salary;
};

int main(){
   struct employee employees[10];
   int counter, index, count, totalSalary;

   printf("Enter Number of Employees\n");
   scanf("%d", &count);

   /* Storing employee detaisl in structure array */
   for(counter=0; counter<count; counter++){
       printf("Enter Name, Age and Salary of Employee\n");
       scanf("%s %d %f", &employees[counter].name,
           &employees[counter].age, &employees[counter].salary);
   }

   /* Calculating average salary of an employee */
   for(totalSalary=0, index=0; index<count; index++){
       totalSalary += employees[index].salary;
   }


   printf("Average Salary of an Employee is %f\n",
       (float)totalSalary/count);

   return 0;
}
```

Output

```
Enter Number of Employees
3
Enter Name, Age and Salary of Employee
Jack 30 100
Enter Name, Age and Salary of Employee
Mike 32 200
Enter Name, Age and Salary of Employee
Nick 40 300
Average Salary of an Employee is 200.000000
```

Array Within Structure in C with Examples

In C programming, we can have **structure members** of **type arrays**. Any structure having an array as a structure member is known as an **array within the structure**. We can have as many members of the type array as we want in C structure.

To access each element from an array within a structure, we write **structure variables followed by dot (.) and then array name along with index**.

Example: Array Within Structure

```
struct student
{
 char name[20];
 int roll;
 float marks[5]; /* This is array within structure
*/
};
```

In above example structure named `student` contains a member element `marks` which is of type array. In this example `marks` is array of floating point numbers.

If we declare structure variable as:

```
struct student s;
```

To access individual marks from above structure **s**, we can write `s.marks[0], s.marks[1], s.marks[2],` and so on.

C Program: Array Within Structure

```
#include<stdio.h>

struct student
{
 char name[20];
 int roll;
 float marks[5]; /* This is array within structure
*/
};

int main()
{
 struct student s;
 int i;
 float sum=0, p;

 printf("Enter name and roll number of
students:\n");
 scanf("%s%d",s.name,&s.roll);
 printf("\nEnter marks obtained in five different
subject\n");

 for(i=0;i< 5;i++)
 {
  printf("Enter marks:\n");
  scanf("%f",&s.marks[i]);
  sum = sum + s.marks[i];
 }

p = sum/5;
printf("Student records and percentage is:\n");
printf("Name : %s\n", s.name);
printf("Roll Number : %d\n", s.roll);
printf("Percentage obtained is: %f", p);

return 0;
}
```

Output

The output of the above program is:

```
Enter name and roll number of students:
Neelima
171

Enter marks obtained in five different subject
Enter marks:
78
Enter marks:
87
Enter marks:
92
Enter marks:
89
Enter marks:
96

Student records and percentage is:
Name: Neelima
Roll Number: 171
Percentage obtained is: 88.40
```

Pointers to Structure in C

The pointers to a structure in C in very similar to the pointer to any in-built data type variable.

The syntax for declaring a pointer to structure variable is as follows:

*struct structure_name *pointer_variable*

For Example

```
struct Employee
{
    char name[50];
    int age;
    float salary;
}employee_one;
struct Employee *employee_ptr;
```

We can use addressOf operator(&) to get the address of structure variable.

```
struct Employee *employee_ptr = &employee_one;
```

To access the member variable using a pointer to a structure variable we can use arrow operator(->). We can access member variable by using pointer variable followed by an arrow operator and then the name of the member variable.

To access a member variable of structure using variable identifier we use dot(.) operator whereas we use arrow(->) operator to access member variable using structure pointer.

```
employee_ptr->salary;
```

C Program to print the members of a structure using Pointer and Arrow Operators

```
#include <stdio.h>


struct employee {
 char name[100];
 int age;
 float salary;
 char department[50];
};

int main(){
   struct employee employee_one, *ptr;

   printf("Enter Name, Age, Salary and Department of Employee\n");
   scanf("%s %d %f %s", &employee_one.name, &employee_one.age,
       &employee_one.salary, &employee_one.department);


   /* Printing structure members using arrow operator */
   ptr = &employee_one;
   printf("\nEmployee Details\n");
   printf(" Name : %s\n Age : %d\n Salary = %f\n Dept : %s\n",
       ptr->name, ptr->age, ptr->salary, ptr->department);

   return 0;
}
```

Output

```
Enter Name, Age, Salary and Department of Employee
Jack 30 1234.5 Sales

Employee Details
 Name : Jack
 Age : 30
 Salary = 1234.500000
 Dept : Sales
```

```
using namespace std;
```

```
struct employee {
    string ename;
    int age, phn_no;
    int salary;
};

// Function to display details of all employees
void display(struct employee emp[], int n)
{
    cout << "Name\tAge\tPhone Number\tSalary\n";
    for (int i = 0; i < n; i++) {
        cout << emp[i].ename << "\t" << emp[i].age << "\t"
             << emp[i].phn_no << "\t" << emp[i].salary
             << "\n";
    }
}

// Driver code
int main()
{
    int n = 3;
    // Array of structure objects
    struct employee emp[n];

    // Details of employee 1
    emp[0].ename = "Chirag";
    emp[0].age = 24;
    emp[0].phn_no = 1234567788;
    emp[0].salary = 20000;

    // Details of employee 2
    emp[1].ename = "Arnav";
    emp[1].age = 31;
    emp[1].phn_no = 1234567891;
    emp[1].salary = 56000;

    // Details of employee 3
    emp[2].ename = "Shivam";
    emp[2].age = 45;
    emp[2].phn_no = 1100661111;
    emp[2].salary = 30500;

    display(emp, n);

    return 0;
}
```

Output

```
Name     Age     Phone Number     Salary

Chirag     24     1234567788     20000
```

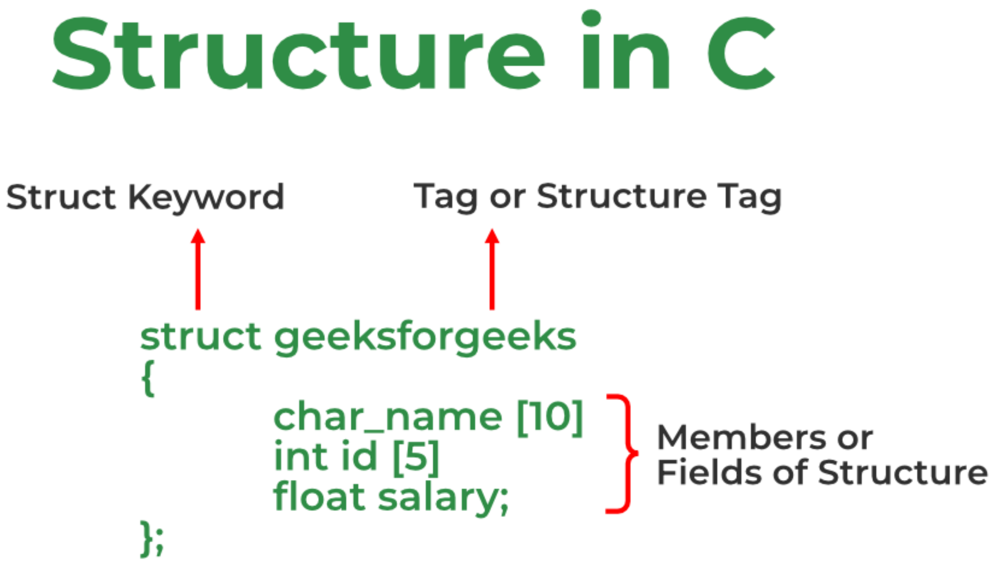
```
Arnav     31     1234567891     56000
Shivam     45     1100661111     30500
```

How to Pass or Return a Structure To or From a Function in C?

A structure is a user-defined data type in C. A structure collects several variables in one place. In a structure, each variable is called a member. The types of data contained in a structure can differ from those in an array (e.g., integer, float, character).

Syntax:

```
struct geeksforgeeks
{
    char name[20];
    int roll_no;
    char branch[20];
}
```



How to Pass a structure as an argument to the functions?(Passing structure to a function):-

When passing structures to or from functions in C, it is important to keep in mind that the entire structure will be copied. This can be expensive in terms of both time and memory, especially for large structures. The passing of structure to the function can be done in two ways:

- By passing all the elements to the function individually.
- By passing the entire structure to the function.

Example 1: Using Call By Value Method

- C

```c
// C program to pass structure as an argument to the
// functions using Call By Value Method
#include <stdio.h>

struct car {
    char name[30];
    int price;
};

void print_car_info(struct car c)
{
    printf("Name : %s", c.name);
    printf("\nPrice : %d\n", c.price);
}

int main()
{
    struct car c = { "Tata", 1021 };
    print_car_info(c);
    return 0;
}
```

Output

```
Name : Tata

Price : 1021
```

Example 2: Using Call By Reference Method

- C

```c
// C program to pass structure as an argument to the
// functions using Call By Reference Method

#include <stdio.h>

struct student {
    char name[50];
    int roll;
    float marks;
};

void display(struct student* student_obj)
{
    printf("Name: %s\n", student_obj->name);
    printf("Roll: %d\n", student_obj->roll);
    printf("Marks: %f\n", student_obj->marks);
}
int main()
```

```
{
    struct student st1 = { "Aman", 19, 8.5 };

    display(&st1);

    return 0;
}
```

Output

```
Name: Aman

Roll: 19

Marks: 8.500000
```

How to Return a Structure From functions?

We can return a structure from a function using the **return** Keyword. To return a structure from a function the return type should be a structure only.

- C

```
// C program to return a structure from a function
#include <stdio.h>

struct student {
    char name[20];
    int age;
    float marks;
};

// function to return a structure
struct student get_student_data()
{
    struct student s;

    printf("Enter name: ");
    scanf("%s", s.name);
    printf("Enter age: ");
    scanf("%d", &s.age);
    printf("Enter marks: ");
    scanf("%f", &s.marks);

    return s;
}

int main()
{
    // structure variable s1 which has been assigned the
    // returned value of get_student_data
    struct student s1 = get_student_data();
    // displaying the information
    printf("Name: %s\n", s1.name);
    printf("Age: %d\n", s1.age);
    printf("Marks: %.1f\n", s1.marks);

    return 0;
}
```

Output:

```
Enter name: Krishna

Enter age: 21

Enter marks: 99

Name: Krishna

Age: 21

Marks: 99.0
```

Structure Pointer(Pointer to Structure):-

The **structure pointer** points to the address of a memory block where the Structure is being stored. Like a pointer that tells the address of another variable of any data type (int, char, float) in memory. And here, we use a structure pointer which tells the address of a structure in memory by pointing pointer variable **ptr** to the structure variable.

Declare a Structure Pointer

The declaration of a structure pointer is similar to the declaration of the structure variable. So, we can declare the structure pointer and variable inside and outside of the main() function. To declare a pointer variable in C, we use the asterisk (*) symbol before the variable's name.

1. struct structure_name *ptr;

After defining the structure pointer, we need to initialize it, as the code is shown:

Initialization of the Structure Pointer

1. ptr = &structure_variable;

We can also initialize a Structure Pointer directly during the declaration of a pointer.

1. struct structure_name *ptr = &structure_variable;

As we can see, a pointer **ptr** is pointing to the address structure_variable of the Structure.

Access Structure member using pointer:

There are two ways to access the member of the structure using Structure pointer:

1. Using (*) asterisk or indirection operator and dot (.) operator.
2. Using arrow (->) operator or membership operator.

Program to access the structure member using structure pointer and the dot operator

Let's consider an example to create a Subject structure and access its members using a structure pointer that points to the address of the Subject variable in C.

Pointer.c

```
#include <stdio.h>

// create a structure Subject using the struct keyword
struct Subject
{
   // declare the member of the Course structure
   char sub_name[30];
   int sub_id;
   char sub_duration[50];
   char sub_type[50];
};

int main()
{
   struct Subject sub; // declare the Subject variable
   struct Subject *ptr; // create a pointer variable (*ptr)
   ptr = ⊂ /* ptr variable pointing to the address of the structure variable sub */

   strcpy (sub.sub_name, " Computer Science");
   sub.sub_id = 1201;
   strcpy (sub.sub_duration, "6 Months");
   strcpy (sub.sub_type, " Multiple Choice Question");

   // print the details of the Subject;
   printf (" Subject Name: %s\t ", (*ptr).sub_name);
         printf (" \n Subject Id: %d\t ", (*ptr).sub_id);
      printf (" \n Duration of the Subject: %s\t ", (*ptr).sub_duration);
        printf (" \n Type of the Subject: %s\t ", (*ptr).sub_type);

   return 0;

}
```

Output:

```
Subject Name:  Computer Science
 Subject Id: 1201
 Duration of the Subject: 6 Months
 Type of the Subject:  Multiple Choice Question
```

In the above program, we have created the **Subject** structure that contains different data elements like sub_name (char), sub_id (int), sub_duration (char), and sub_type (char). In this, the **sub** is the structure variable, and ptr is the structure pointer variable that points to the address of the sub variable like ptr = &sub. In this way, each *ptr is accessing the address of the Subject structure's member.

Structure pointer - Declaration, Accessing of Structure members

The objects to a structure can also be a pointer same like we can create a pointer of int. To achieve this which we will need following declaration:

```
struct tagname *structPtrVariable;
```

To access the elements inside the structure we should be using the following syntax

```
structPtrVariable->x = 'A' // here '.' is replace by '->'
structPtrVariable->y = 20;
structPtrVariable->z = 10.20f;
```

Above we have created two objects for the `struct tagname`. Those two objects have independent memory allocated for each of them. Both of them can be compared with the normal declaration of the variables for example:

```
int a;
int *a;
```

Structures play very important role in big systems where we need to combine several data's into a set and need to capture multiples of that set, perform operations and many more.

It even helps on the smaller systems.

A small piece of code will help understand the use of structures better.

```
#include <stdio.h>

struct tagname {
       char Char;
       int  Int;
       float Dec;
};
```

```c
int main()
{
        struct tagname StructObj;
        struct tagname *ptrStructObj=&StructObj;

        StructObj.Char='H';
        ptrStructObj->Int=927;
        ptrStructObj->Dec=911.0f;

        printf("%C\n",StructObj.Char);
        printf("%d\n",ptrStructObj->Int);
        printf("%f",ptrStructObj->Dec);
        printf("\n");

        return 0;
}
```

Output

```
H
927
911.000000
```

Structure Within Structure (Nested Structure):-

C provides us the feature of nesting one structure within another structure by using which, complex data types are created. For example, we may need to store the address of an entity employee in a structure. The attribute address may also have the subparts as street number, city, state, and pin code. Hence, to store the address of the employee, we need to store the address of the employee into a separate structure and nest the structure address into the structure employee.

Consider the following program.:-

```c
#include<stdio.h>
struct address
{
   char city[20];
   int pin;
   char phone[14];
};
struct employee
{
   char name[20];
   struct address add;
};
```

```
void main ()
{
    struct employee emp;
   printf("Enter employee information?\n");
    scanf("%s %s %d %s",emp.name,emp.add.city, &emp.add.pin, emp.add.p
hone);
   printf("Printing the employee information....\n");
    printf("name: %s\nCity: %s\nPincode: %d\nPhone: %s",emp.name,emp.ad
d.city,emp.add.pin,emp.add.phone);
}
```

Output

```
Enter employee information?
Arun

Delhi
110001
1234567890

Printing the employee information....

name: Arun
City: Delhi
Pincode: 110001
Phone: 1234567890
```

The structure can be nested in the following ways.

1. **By separate structure**
2. **By Embedded structure**

1) Separate structure

Here, we create two structures, but the dependent structure should be used inside the main structure as a member. Consider the following example.

```
struct Date
{
   int dd;
  int mm;
   int yyyy;
};
struct Employee
{
   int id;
  char name[20];
   struct Date doj;
}emp1;
```

As you can see, doj (date of joining) is the variable of type Date. Here doj is used as a member in Employee structure. In this way, we can use Date structure in many structures.

2) Embedded structure

The embedded structure enables us to declare the structure inside the structure. Hence, it requires less line of codes but it can not be used in multiple data structures. Consider the following example.

```
struct Employee
{
  int id;
  char name[20];
  struct Date
   {
     int dd;
     int mm;
     int yyyy;
   }doj;
}emp1;
```

Accessing Nested Structure

We can access the member of the nested structure by Outer_Structure.Nested_Structure.member as given below:

1. e1.doj.dd
2. e1.doj.mm
3. e1.doj.yyyy

C Nested Structure example

Let's see a simple example of the nested structure in C language.

```
#include <stdio.h>
#include <string.h>
struct Employee
{
  int id;
  char name[20];
  struct Date
   {
     int dd;
     int mm;
     int yyyy;
   }doj;
}e1;
```

```
int main( )
{
  //storing employee information
  e1.id=101;
  strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
  e1.doj.dd=10;
  e1.doj.mm=11;
  e1.doj.yyyy=2014;

  //printing first employee information
  printf( "employee id : %d\n", e1.id);
  printf( "employee name : %s\n", e1.name);
  printf( "employee date of joining (dd/mm/yyyy) : %d/%d/%d\n", e1.doj.dd,e1.doj
.mm,e1.doj.yyyy);
  return 0;
}
```

Output:

```
employee id : 101
employee name : Sonoo Jaiswal
employee date of joining (dd/mm/yyyy) : 10/11/2014
```

Union in C

Union can be defined as a user-defined data type which is a collection of different variables of different data types in the same memory location. The union can also be defined as many members, but only one member can contain a value at a particular point in time.

Union is a user-defined data type, but unlike structures, they share the same memory location.

Let's understand this through an example.

```
struct abc
{
  int a;
  char b;
}
```

The above code is the user-defined structure that consists of two members, i.e., 'a' of type **int** and 'b' of type **character**. When we check the addresses of 'a' and 'b', we found that their addresses are different. Therefore, we conclude that the members in the structure do not share the same memory location.

When we define the union, then we found that union is defined in the same way as the structure is defined but the difference is that union keyword is used for defining the union data type, whereas the struct keyword is used for defining the structure. The union contains the data members, i.e., 'a' and 'b', when we check the addresses of both the variables then we found that both have the same addresses. It means that the union members share the same memory location.

Let's have a look at the pictorial representation of the memory allocation.

The below figure shows the pictorial representation of the structure. The structure has two members; i.e., one is of integer type, and the another one is of character type. Since 1 block is equal to 1 byte; therefore, 'a' variable will be allocated 4 blocks of memory while 'b' variable will be allocated 1 block of memory.

The below figure shows the pictorial representation of union members. Both the variables are sharing the same memory location and having the same initial address.

In union, members will share the memory location. If we try to make changes in any of the member then it will be reflected to the other member as well. Let's understand this concept through an example.

```
union abc
{
   int a;
char b;
}var;
int main()
{
  var.a = 66;
  printf("\n a = %d", var.a);
  printf("\n b = %d", var.b);
}
```

In the above code, union has two members, i.e., 'a' and 'b'. The 'var' is a variable of union abc type. In the **main()** method, we assign the 66 to 'a' variable, so var.a will print 66 on the screen. Since both 'a' and 'b' share the memory location, **var.b** will print '**B**' (ascii code of 66).

Deciding the size of the union

The size of the union is based on the size of the largest member of the union.

Let's understand through an example.

```
union abc{
int a;
char b;
float c;
double d;
};
int main()
{
  printf("Size of union abc is %d", sizeof(union abc));
  return 0;
}
```

As we know, the size of int is 4 bytes, size of char is 1 byte, size of float is 4 bytes, and the size of double is 8 bytes. Since the double variable occupies the largest memory among all the four variables, so total 8 bytes will be allocated in the memory. Therefore, the output of the above program would be 8 bytes.

Accessing members of union using pointers

We can access the members of the union through pointers by using the (->) arrow operator.

Let's understand through an example.

```
#include <stdio.h>
union abc
{
   int a;
   char b;
};
int main()
{
   union abc *ptr; // pointer variable declaration
   union abc var;
   var.a= 90;
   ptr = &var;
   printf("The value of a is : %d", ptr->a);
   return 0;
}
```

In the above code, we have created a pointer variable, i.e., *ptr, that stores the address of var variable. Now, ptr can access the variable 'a' by using the (->) operator. Hence the output of the above code would be 90.

Why do we need C unions?

Consider one example to understand the need for C unions. Let's consider a store that has two items:

- Books
- Shirts

Store owners want to store the records of the above-mentioned two items along with the relevant information. For example, Books include Title, Author, no of pages, price, and Shirts include Color, design, size, and price. The 'price' property is common in both items. The Store owner wants to store the properties, then how he/she will store the records.

Initially, they decided to store the records in a structure as shown below:

```
struct store
{
   double price;
   char *title;
   char *author;
   int number_pages;
   int color;
  int size;
  char *design;
};
```

The above structure consists of all the items that store owner wants to store. The above structure is completely usable but the price is common property in both the items and the rest of the items are individual. The properties like price, *title, *author, and number_pages belong to Books while color, size, *design belongs to Shirt.

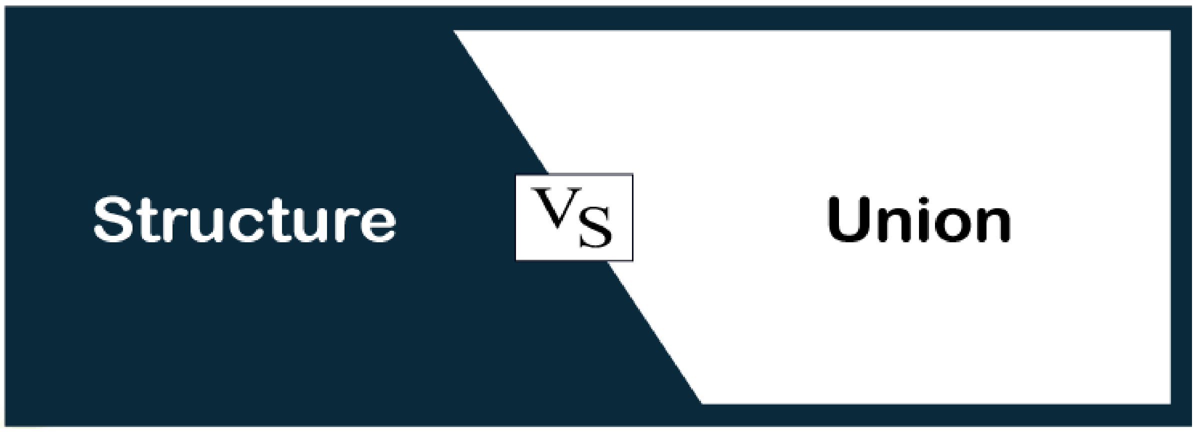
Let's see how can we access the members of the structure.

```
int main()
{
   struct store book;
   book.title = "C programming";
book.author = "Paulo Cohelo";
book.number_pages = 190;
book.price = 205;
printf("Size is : %ld bytes", sizeof(book));
return 0;
}
```

In the above code, we have created a variable of type **store**. We have assigned the values to the variables, title, author, number_pages, price but the book variable does not possess the properties such as size, color, and design. Hence, it's a wastage of memory. The size of the above structure would be 44 bytes.

Difference between Structure and Union:-

Structure and **union** both are *user-defined* data types in the C/C++ programming language.



What is a structure (struct)?

Structure (struct) is a user-defined data type in a programming language that stores different data types' values together. The **struct** keyword is used to define a structure data type in a program. The struct data type stores one or more than one data element of different kinds in a variable.

Suppose that you want to store the data of employee in your C/C++ project, where you have to store the following different parameters:

- Id
- Name
- Department
- Email Address

One way to store 4 different data by creating 4 different arrays for each parameter, such as id[], name[], department[], and email[]. Using array id[i] represents the id of the ith employee. Similarly, name[i] represents the name of ith employee (name). Array element department[i] and email[i] represent the ith employee's department and email address.

The advantage of using a separate array for each parameter is simple if there are only a few parameters. The disadvantage of such logic is that it is quite difficult to manage employee data. Think about a scenario in which you have to deal with 100 or even more parameters associated with one employee. It isn't easy to manage 100 or even more arrays. In such a condition, a **struct** comes in.

Syntax of struct

```
struct [structure_name]
{
    type member_1;
    type member_2;
    . . .
    type member_n;
};
```

Example

```
struct employee
{
    int id;
    char name[50];
    string department;
    string email;
};
```

What is a Union?

In "c," programming union is *a user-defined data type* that is used to store the different data type's values. However, in the union, one member will occupy the memory at once. In other words, we can say that the size of the union is equal to the size of its largest data member size. Union offers an effective way to use the same memory location several times by each data member. The **union** keyword is used to define and create a union data type.

Syntax of Union

```
union [union name]
    {
type member_1;
    type member_2;
    . . .
    type member_n;
    };
```

Example

```
union employee
{
    string name;
    string department;
    int phone;
    string email;
};
```

Example of Structure and Union showing the difference in C

Let's see an example of structure (struct) and union in c programming, illustrating the various differences between them.

```
// A simple C program showing differences between Structure and Union
#include <stdio.h>
#include <string.h>
// declaring structure
struct struct_example
{
    int integer;
    float decimal;
    char name[20];
};
// declaraing union
union union_example
{
    int integer;
    float decimal;
    char name[20];
};
void main()
{
    // creating variable for structure and initializing values difference six
    struct struct_example stru ={5, 15, "John"};

    // creating variable for union and initializing values
    union union_example uni = {5, 15, "John"};

    printf("data of structure:\n integer: %d\n decimal: %.2f\n name: %s\n", stru.integer, str
u.decimal, stru.name);
    printf("\ndata of union:\n integer: %d\n" "decimal: %.2f\n name: %s\n", uni.inte
ger, uni.decimal, uni.name);

    // difference five
    printf("\nAccessing all members at a time:");
    stru.integer = 163;
    stru.decimal = 75;
    strcpy(stru.name, "John");
    printf("\ndata of structure:\n integer: %d\n " "decimal: %f\n name: %s\n", stru.integer,
stru.decimal, stru.name);

    uni.integer = 163;
    uni.decimal = 75;
    strcpy(uni.name, "John");
    printf("\ndata of union:\n integeer: %d\n " "decimal: %f\n name: %s\n", uni.inte
ger, uni.decimal, uni.name);

    printf("\nAccessing one member at a time:");
    printf("\ndata of structure:");
    stru.integer = 140;
    stru.decimal = 150;
    strcpy(stru.name, "Mike");

    printf("\ninteger: %d", stru.integer);
    printf("\ndecimal: %f", stru.decimal);
    printf("\nname: %s", stru.name);

    printf("\ndata of union:");
    uni.integer = 140;
    uni.decimal = 150;
    strcpy(uni.name, "Mike");

    printf("\ninteger: %d", uni.integer);
    printf("\ndecimal: %f", uni.decimal);
    printf("\nname: %s", uni.name);

    //difference four
    printf("\nAltering a member value:\n");
    stru.integer = 512;
    printf("data of structure:\n integer: %d\n decimal: %.2f\n name: %s\n", stru.inte
ger, stru.decimal, stru.name);
    uni.integer = 512;
    printf("data of union:\n integer: %d\n decimal: %.2f\n name: %s\n", uni.integer,
uni.decimal, uni.name);

    // difference two and three
    printf("\nsizeof structure: %d\n", sizeof(stru));
    printf("sizeof union: %d\n", sizeof(uni));
}
```

Output

```
data of structure:
 integer: 5
 decimal: 15.00
 name: John

data of union:
 integer: 5
decimal: 0.00
 name:

Accessing all members at a time:
data of structure:
 integer: 163
 decimal: 75.000000
 name: John
```

```
data of union:
 integer: 1852337994
 decimal: 17983765624912253034071851008.000000
 name: John

Accessing one member at a time:
data of structure:
integer: 140
decimal: 150.000000
name: Mike
data of union:
integer: 1701538125
decimal: 69481161252302940536832.000000
name: Mike
Altering a member value:
data of structure:
 integer: 512
 decimal: 150.00
 name: Mike
data of union:
 integer: 512
 decimal: 0.00
 name:

sizeof structure: 28
sizeof union: 20
```

Difference between Structure and Union

Let's summarize the above discussed topic about the Struct and Union in the form of a table that highlight the differences between structure and union:

Struct	Union
The struct keyword is used to define a structure.	The union keyword is used to define union.
When the variables are declared in a structure, the compiler allocates memory to each variables member. The size of a structure is equal or greater to the sum of the sizes of each data member.	When the variable is declared in the union, the compiler allocates memory to the largest size variable member. The size of a union is equal to the size of its largest data member size.
Each variable member occupied a unique memory space.	Variables members share the memory space of the largest size variable.
Changing the value of a member will not affect other variables members.	Changing the value of one member will also affect other variables members.
Each variable member will be assessed at a time.	Only one variable member will be assessed at a time.
We can initialize multiple variables of a structure at a time.	In union, only the first data member can be initialized.

All variable members store some value at any point in the program.	Exactly only one data member stores a value at any particular instance in the program.
The structure allows initializing multiple variable members at once.	Union allows initializing only one variable member at once.
It is used to store different data type values.	It is used for storing one at a time from different data type values.
It allows accessing and retrieving any data member at a time.	It allows accessing and retrieving any one data member at a time.

Enumerated Data Type in C:-

The enum in C is also known as the enumerated type. It is a user-defined data type that consists of integer values, and it provides meaningful names to these values. The use of enum in C makes the program easy to understand and maintain. The enum is defined by using the enum keyword.

The following is the way to define the enum in C:

```
1.  enum flag{integer_const1, integer_const2,.....integter_constN};
```

In the above declaration, we define the enum named as flag containing 'N' integer constants. The default value of integer_const1 is 0, integer_const2 is 1, and so on. We can also change the default value of the integer constants at the time of the declaration.

For example:

```
enum fruits{mango, apple, strawberry, papaya};
```

The default value of mango is 0, apple is 1, strawberry is 2, and papaya is 3. If we want to change these default values, then we can do as given below:

```
enum fruits{
mango=2,
apple=1,
strawberry=5,
papaya=7,
};
```

Enumerated type declaration

As we know that in C language, we need to declare the variable of a pre-defined type such as int, float, char, etc. Similarly, we can declare the variable of a user-defined data type, such as enum. Let's see how we can declare the variable of an enum type.

Suppose we create the enum of type status as shown below:

```
enum status{false,true};
```

Now, we create the variable of status type:

```
enum status s; // creating a variable of the status type.
```

In the above statement, we have declared the 's' variable of type status.

To create a variable, the above two statements can be written as:

```
enum status{false,true} s;
```

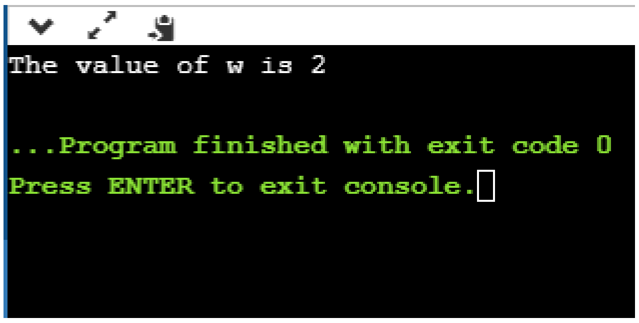
In this case, the default value of false will be equal to 0, and the value of true will be equal to 1.

Let's create a simple program of enum.

```
#include <stdio.h>
 enum weekdays{Sunday=1, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
 int main()
{
 enum weekdays w; // variable declaration of weekdays type
 w=Monday; // assigning value of Monday to w.
 printf("The value of w is %d",w);
   return 0;
}
```

In the above code, we create an enum type named as weekdays, and it contains the name of all the seven days. We have assigned 1 value to the Sunday, and all other names will be given a value as the previous value plus one.

Output



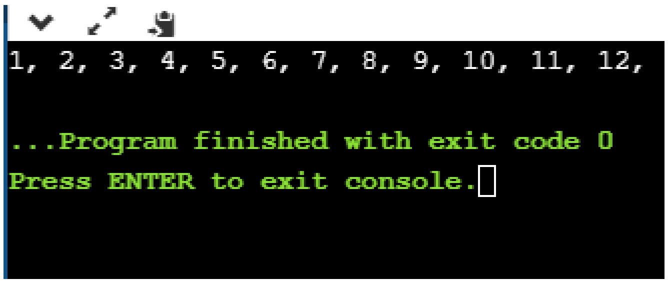
```
The value of w is 2

...Program finished with exit code 0
Press ENTER to exit console.
```

Let's demonstrate another example to understand the enum more clearly.

```
#include <stdio.h>
 enum months{jan=1, feb, march, april, may, june, july, august, september, october, nove
mber, december};
int main()
{
// printing the values of months
 for(int i=jan;i<=december;i++)
 {
 printf("%d, ",i);
 }
   return 0;
}
```

In the above code, we have created a type of enum named as months which consists of all the names of months. We have assigned a '1' value, and all the other months will be given a value as the previous one plus one. Inside the main() method, we have defined a for loop in which we initialize the 'i' variable by jan, and this loop will iterate till December.

Output



```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,

...Program finished with exit code 0
Press ENTER to exit console.
```

Why do we use enum?

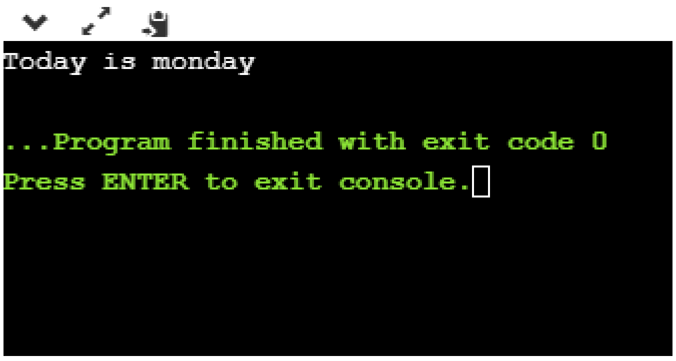
The enum is used when we want our variable to have only a set of values. For example, we create a direction variable. As we know that four directions exist (North, South, East, West), so this direction variable will have four possible values. But the variable can hold only one value at a time. If we try to provide some different value to this variable, then it will throw the compilation error.

The enum is also used in a switch case statement in which we pass the enum variable in a switch parenthesis. It ensures that the value of the case block should be defined in an enum.

Let's see how we can use an enum in a switch case statement.

```
#include <stdio.h>
enum days{sunday=1, monday, tuesday, wednesday, thursday, friday, saturday};
int main()
{
  enum days d;
  d=monday;
  switch(d)
  {
     case sunday:
     printf("Today is sunday");
     break;
     case monday:
     printf("Today is monday");
     break;
     case tuesday:
     printf("Today is tuesday");
     break;
     case wednesday:
     printf("Today is wednesday");
     break;
     case thursday:
     printf("Today is thursday");
     break;
     case friday:
     printf("Today is friday");
     break;
     case saturday:
     printf("Today is saturday");
     break;
  }

  return 0;
}
```

Output



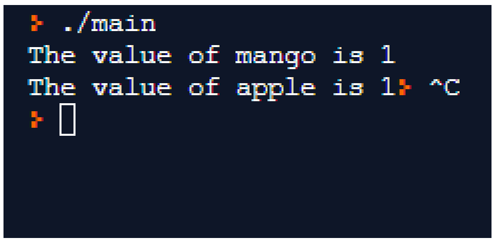
```
Today is monday

...Program finished with exit code 0
Press ENTER to exit console.
```

Some important points related to enum

- The enum names available in an enum type can have the same value. Let's look at the example.

```
#include <stdio.h>

int main(void) {
  enum fruits{mango = 1, strawberry=0, apple=1};
    printf("The value of mango is %d", mango);
    printf("\nThe value of apple is %d", apple);
  return 0;
}
```

Output



```
› ./main
The value of mango is 1
The value of apple is 1› ^C
›
```

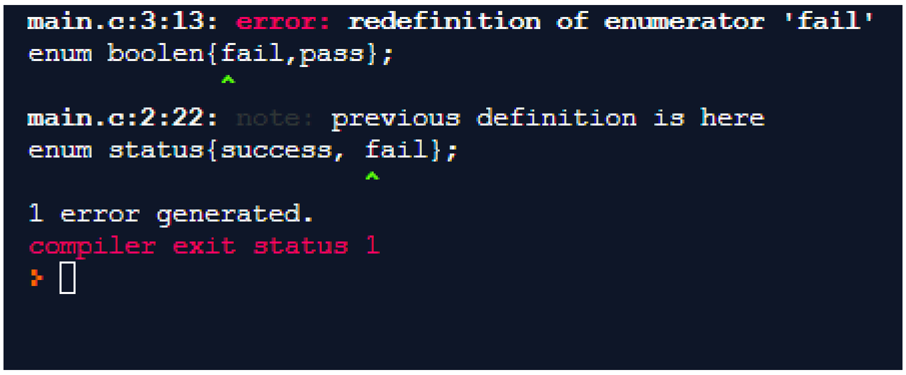
- If we do not provide any value to the enum names, then the compiler will automatically assign the default values to the enum names starting from 0.
- We can also provide the values to the enum name in any order, and the unassigned names will get the default value as the previous one plus one.
- The values assigned to the enum names must be integral constant, i.e., it should not be of other types such string, float, etc.
- All the enum names must be unique in their scope, i.e., if we define two enum having same scope, then these two enums should have different enum names otherwise compiler will throw an error.

Let's understand this scenario through an example.

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

```c
enum status{success, fail};
enum boolen{fail,pass};
int main(void) {

    printf("The value of success is %d", success);
  return 0;
}
```

Output

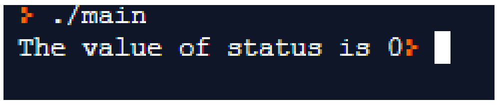


```
main.c:3:13: error: redefinition of enumerator 'fail'
enum boolen{fail,pass};
            ^
main.c:2:22: note: previous definition is here
enum status{success, fail};
                     ^
1 error generated.
compiler exit status 1
>
```

- In enumeration, we can define an enumerated data type without the name also.

```c
#include <stdio.h>
enum {success, fail} status;
int main(void) {
status=success;
printf("The value of status is %d", status);
return 0;
   }
```

Output



```
> ./main
The value of status is 0>
```