Data Structure through C

Difference Between Structure and Union in C

In C programming, both structures and unions are used to group different types of data under a single name, but they behave in different ways. The main difference lies in how they store data.

Parameter	**Structure**	**Union**
Definition	A structure is a user-defined data type that groups different data types into a single entity.	A union is a user-defined data type that allows storing different data types at the same memory location.
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union
Size	The size is the sum of the sizes of all members, with padding if necessary.	The size is equal to the size of the largest member, with possible padding.

Memory Allocation	Each member within a structure is allocated unique storage area of location.	Memory allocated is shared by individual members of union.
Data Overlap	No data overlap as members are independent.	Full data overlap as members shares the same memory.
Accessing Members	Individual member can be accessed at a time.	Only one member can be accessed at a time.

Structures

A [structure in C](#) is a collection of variables, possibly of different types, under a single name. Each member of the structure is allocated its own memory space, and the size of the structure is the sum of the sizes of all its members.

Syntax

```
struct name {
    member1 definition;
    member2 definition;
    ...
    memberN definition;
};
```

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

```
union Data {
    int i;
    double d;
    char c;
};

int main() {

    // Create a union variable
    union Data data;

    // Store an integer in the union
    data.i = 100;
    printf("%d
", data.i);

    // Store a double in the union (this will
        // overwrite the integer value)
    data.d = 99.99;
    printf("%.2f
", data.d);

    // Store a character in the union (this will
        // overwrite the double value)
    data.c = 'A';
    printf("%c
```

```
", data.c);

        printf("Size: %d", sizeof(data));

    return 0;
}
```

Output

100

99.99

A

Size: 8

Similarities Between Structure and Union

Structures and unions are also similar in some aspects listed below:

- *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 or selection operator, which has one of the highest precedences, is used for accessing member variables inside both the user-defined datatypes.*

Unions

A **union in C** is similar to a structure, but with a key difference: all members of a union share the same memory location. This means only one member of the union can store a value at any given time. The size of a union is determined by the size of its largest member.

```
union name {
    member1 definition;
    member2 definition;
    ...
    memberN definition;
};

#include <stdio.h>

union Data {
   int i;
   double d;
   char c;
};

int main() {

   // Create a union variable
   union Data data;

   // Store an integer in the union
```

```
    data.i = 100;
    printf("%d
", data.i);

    // Store a double in the union (this will
        // overwrite the integer value)
    data.d = 99.99;
    printf("%.2f
", data.d);

    // Store a character in the union (this will
        // overwrite the double value)
    data.c = 'A';
    printf("%c
", data.c);

        printf("Size: %d", sizeof(data));

    return 0;
}
```

Output

100

99.99

A

Size: 8

C typedef

*The **typedef** is a keyword that is used to provide existing data types with a new name. The C typedef keyword is used to redefine the name of already existing data types. When names of datatypes become difficult to use in programs, typedef is used with user-defined datatypes, which behave similarly to defining an alias for commands.*

Let's take a look at an example:

```
#include <stdio.h>


typedef int Integer;


int main() {


    // n is of type int, but we are using
        // alias Integer
    Integer n = 10;


    printf("%d", n);
    return 0;
}
```

Output

10

Syntax of typedef

typedef *existing_type new_type;*

where,

- ***existing_type:*** *The type that we want to alias (e.g., int, float, struct, etc.).*
- ***new_type:*** *The new alias or name for the existing type.*

After this declaration, we can use the ***alias_name*** *as if it were the real* ***existing_name*** *in out C program.*

Examples of typedef in C

The below examples illustrate the use of typedef for different purposes in our C program:

Define an Alias for Built-in Data Type

```
#include <stdio.h>


// Defining an alias using typedef
typedef long long ll;


int main() {
```

```
    // Using typedef alias name to declare variable
    ll a = 20;


    printf("%lld", a);
    return 0;
}
```

Output

20

Explanation: *In this code, typedef is used to define an alias* ***ll*** *for the* ***long long data type****. The variable* ***a*** *is declared using* ***ll*** *as* ***a*** *shorthand, and its value is printed using printf. This makes the code more readable and concise by using the alias instead of the full type of name.*

Define an Alias for a Structure

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


// Using typedef to define an alias for structure
typedef struct Students {
    char name[50];
    char branch[50];
```

```
    int ID_no;
} stu;


int main() {


        // Using alias to define structure
    stu s;
    strcpy(s.name, "Geeks");
    strcpy(s.branch, "CSE");
    s.ID_no = 108;


    printf("%s\n", s.name);
    printf("%s\n", s.branch);
    printf("%d", s.ID_no);
    return 0;
}
```

Output

```
Geeks
CSE
108
```

Explanation: *In this code, typedef is used to define an alias* ***stu*** *for the* *structure* ***Students****. The alias simplifies declaring variables of this structure type, such as* ***st****. The program then initializes and prints the values of the structure members name, branch, and ID_no. This approach enhances code readability by using the alias instead of the full* ***struct students*** *declaration.*

Define an Alias for Pointer Type

```
#include <stdio.h>


// Creating alias for pointer
typedef int* ip;


int main() {
    int a = 10;
    ip ptr = &a;


    printf("%d", *ptr);
    return 0;
}
```

Output

10

Define an Alias for Array

```
#include <stdio.h>


// Here 'arr' is an alias
typedef int arr[4];


int main() {
    arr a = { 10, 20, 30, 40 };


    for (int i = 0; i < 4; i++)
        printf("%d ", a[i]);


    return 0;
}
```

Output

10 20 30 40

typedef vs #define

The #define preprocessor can also be used to create an alias but there are some primary [differences between the typedef and #define](#) in C:

1. *#define is capable of defining aliases for values as well, for instance, you can define 1 as ONE, 3.14 as PI, etc. Typedef is limited to giving symbolic names to types only.*
2. *Preprocessors interpret #define statements, while the compiler interprets typedef statements.*
3. *There should be no semicolon at the end of #define, but a semicolon at the end of typedef.*
4. *In contrast with #define, typedef will actually define a new type by copying and pasting the definition values.*

C Structure

*In C, a **structure** 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 a structure. The items in the structure are called its **member** and they c#include <stdio.h>*

```
// Defining a structure
struct A {
    int x;
};

int main() {
```

```
    // Creating a structure variable
    struct A a;

        // Initializing member
        a.x = 11;

    printf("%d", a.x);
    return 0;
}
```

an be of any valid data type.

Output

11

Explanation: *In this example, a structure **A** is defined to hold an integer member **x**. A variable **a** of type **struct A** is created and its member **x** is initialized to **11** by accessing it using dot operator. The value of **a.x** is then printed to the console.*

*Structures are used when you want to store a collection of different data types, such as integers, floats, or even other structures under a single name. To understand how structures are foundational to building complex data structures, the **C Programming Course Online with Data Structures** provides practical applications and detailed explanations.*

Syntax of Structure

There are two steps of creating a structure in C:

1. *Structure Definition*
2. *Creating Structure Variables*

Structure Definition

A structure is defined using the ***struct*** *keyword followed by the structure name and its members. It is also called a structure* ***template*** *or structure* ***prototype****, and no memory is allocated to the structure in the declaration.*

```
struct structure_name {
    data_type1 member1;
    data_type2 member2;
    ...
};
```

- ***structure_name:*** *Name of the structure.*
- ***member1, member2, ...:*** *Name of the members.*
- ***data_type1****,* ***data_type2****, ...: Type of the members.*

Be careful not to forget the semicolon at the end.

Creating Structure Variable

After structure definition, we have to create variable of that structure to use it. It is similar to the any other type of variable declaration:

```
struct strcuture_name var;
```

We can also declare structure variables with structure definition.

```
struct structure_name {
    ...
}var1, var2....;
```

Basic Operations of Structure

Following are the basic operations commonly used on structures:

1. Access Structure Members

*To access or modify members of a structure, we use the **(.) dot operator**. This is applicable when we are using structure variables directly.*

```
structure_name . member1;
strcuture_name . member2;
```

*In the case where we have a pointer to the structure, we can also use the **arrow operator** to access the members.*

```
structure_ptr -> member1
structure_ptr -> member2
```

2. Initialize Structure Members

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

```
struct structure_name {
    data_type1 member1 = value1;  // COMPILER ERROR:  cannot initialize
members here
    data_type2 member2 = value2;   // COMPILER ERROR:  cannot initialize
members here
```

```
    ...
};
```

The reason for the above error is simple. When a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created. So there is no space to store the value assigned.

We can initialize structure members in 4 ways which are as follows:

Default Initialization

By default, structure members are not automatically initialized to 0 or NULL. Uninitialized structure members will contain garbage values. However, when a structure variable is declared with an initializer, all members not explicitly initialized are zero-initialized.

```
struct structure_name = {0}; // Both x and y are initialized to 0
```

Initialization using Assignment Operator

```
struct structure_name str;
str.member1 = value1;
....
```

Note: *We cannot initialize the arrays or strings using assignment operator after variable declaration.*

Initialization using Initializer List

```
struct structure_name str = {value1, value2, value3 ....};
```

In this type of initialization, the values are assigned in sequential order as they are declared in the structure template.

Initialization using Designated Initializer List

Designated Initialization allows structure members to be initialized in any order. This feature has been added in the C99 standard.

***struct** structure_name str = { .member1 = value1, .member2 = value2, .member3 = value3 };*

The Designated Initialization is only supported in C but not in C++.

```
#include <stdio.h>


// Defining a structure to represent a student

struct Student {

    char name[50];

    int age;

    float grade;

};


int main() {


    // Declaring and initializing a structure

    // variable

    struct Student s1 = {"Rahul",20, 18.5};
```

```
    // Designated Initializing another stucture
        struct Student s2 = {.age = 18, .name =
        "Vikas", .grade = 22};

    // Accessing structure members
    printf("%s\t%d\t%.2f\n", s1.name, s1.age,
    s1.grade);
    printf("%s\t%d\t%.2f\n", s2.name, s2.age,
    s2.grade);

    return 0;
}
```

Output

```
Rahul 20    18.50
Vikas 18    22.00
```

3. Copy Structure

Copying structure is simple as copying any other variables. For example, s1 is copied into s2 using assignment operator.

s2 = s1;

*But this method only creates a shallow copy of s1 i.e. if the structure **s1** have some dynamic resources allocated by malloc, and it contains pointer to that resource, then only the pointer will be copied to **s2.** If the dynamic resource is also needed, then it has to be copied manually (deep copy)*

```
#include <stdio.h>

#include <stdlib.h>


struct Student {

    int id;

        char grade;

};


int main() {

    struct Student s1 = {1, 'A'};


        // Create a copy of student s1

        struct Student s1c = s1;


    printf("Student 1 ID: %d\n", s1c.id);

    printf("Student 1 Grade: %c", s1c.grade);
```

```
    return 0;
}
```

Output

```
Student 1 ID: 1
Student 1 Grade: A
```

4. Passing Structure to Functions

Structure can be passed to a function in the same way as normal variables. Though, it is recommended to pass it as a pointer to avoid copying a large amount of data.

```
#include <stdio.h>


// Structure definition
struct A {
  int x;
};


// Function to increment values
void increment(struct A a, struct A* b) {
  a.x++;
    b->x++;
}
```

```
int main() {
    struct A a  = { 10 };
        struct A b  = { 10 };

        // Passing a by value and b by pointer
        increment(a, &b);

        printf("a.x: %d \tb.x: %d", a.x, b.x);
    return 0;
}
```

Output

a.x: 10   b.x: 11

5. typedef for Structures

The typedef keyword is used to define an alias for the already existing datatype. In structures, we have to use the struct keyword along with the structure name to define the variables. Sometimes, this increases the length and complexity of the code. We can use the typedef to define some new shorter name for the structure.

In C, we can create an array whose elements are of struct type. In this article, we will learn how to access an array of structures in C.

For Example,

Input:

myArrayOfStructs = {{'a', 10}, {'b', 20}, {'A', 9}}

Output:

Integer Member at index 1: 20

Accessing Array of Structure Members in C

We can access the array of structures in a very similar to accessing array elements. To access members of an **array of structures** we can use the array indexing and access individual fields of a struct using the **dot operator**.

Syntax to Access Array of Structure in C

arrayName[index].member;

Here,

- arrayName is the name of the array of struct.
- index is the position of the struct in the array that we want to access, starting from 0.
- member is the name of the member within the struct that we want to access.

C Program to Access Array of Struct Members

The below program demonstrates how we can access an **array of structures** in C.

C

```
// C Program to access the members of array of structure
#include <stdio.h>

// Defining the struct
struct MyStruct {
    int id;
    char name[20];
};

int main()
{
    // Declaring an array of structs
    struct MyStruct myArray[] = {
        { 1, "Person1" },
        { 2, "Person2" },
    };

    // Accessing and printing data using array indexing
    printf("Struct at index 0: ID = %d, Name = %s\n",
           myArray[0].id, myArray[0].name);

    // Modifying the id of the second person
    myArray[1].id = 3;
```

```
    // Accessing and printing the updated information of the
    // second person
    printf("Struct at index 1 after modifying: ID = %d, "
           "Name = %s\n",
           myArray[1].id, myArray[1].name);

    return 0;
}
```

Output

Struct at index 0: ID = 1, Name = Person1

Struct at index 1 after modifying: ID = 3, Name = Person2

Time Complexity: O(1)
Space Complexity: O(1)

A structure pointer is a pointer variable that stores the address of a structure. It allows the programmer to manipulate the structure and its members directly by referencing their memory location rather than passing the structure itself. In this article let's take a look at structure pointer in C.

Let's take a look at an example:

C

```c
#include <stdio.h>

struct A {
    int var;
};

int main() {
    struct A a = {30};

    // Creating a pointer to the structure
    struct A *ptr;

    // Assigning the address of person1 to the pointer
    ptr = &a;

    // Accessing structure members using the pointer
    printf("%d", ptr->var);

    return 0;
}
```

Output

30

Explanation: In this example, ptr is a pointer to the structure A. It stores the address of the structure a, and the structure's member var is accessed using the pointer with the -> operator. This allows efficient access to the structure's members without directly using the structure variable.

Syntax of Structure Pointer

The syntax of structure pointer is similar to any other pointer to variable:

struct struct_name ***ptr_name;**

Here, **struct_name** is the name of the structure, and **ptr_name** is the name of the pointer variable.

Accessing Member using Structure Pointers

There are two ways to access the members of the structure with the help of a structure pointer:

1. Differencing and Using (.) Dot Operator.
2. Using (->) Arrow operator.

Differencing and Using (.) Dot Operator

First method is to first dereference the structure pointer to get to the structure and then use the dot operator to access the member. Below is the program to access the structure members using the structure pointer with the **help of the dot operator.**

C

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

struct Student {
    int roll_no;
    char name[30];
    char branch[40];
    int batch;
};

int main() {

    struct Student s1 = {27, "Geek", "CSE", 2019};

        // Pointer to s1
    struct Student* ptr = &s1;

        // Accessing using dot operator
    printf("%d\n", (*ptr).roll_no);
```

```
    printf("%s\n", (*ptr).name);

    printf("%s\n", (*ptr).branch);

    printf("%d", (*ptr).batch);


    return 0;

}
```

Output

27

Geek

CSE

2019

Using (->) Arrow Operator

C language provides an array operator (->) that can be used to directly access the structure member without using two separate operators. Below is the program to access the structure members using the structure pointer with the **help of the Arrow operator.**

C

```
#include <stdio.h>

#include <string.h>
```

```
struct Student {
    int roll_no;
    char name[30];
    char branch[40];
    int batch;
};

int main() {

    struct Student s1 = {27, "Geek", "CSE", 2019};

        // Pointer to s1
    struct Student* ptr = &s1;

        // Accessing using dot operator
    printf("%d\n", ptr->roll_no);
    printf("%s\n", ptr->name);
    printf("%s\n", ptr->branch);
    printf("%d", ptr->batch);
```

```
    return 0;
}
```

Output

27

Geek

CSE

2019

Explanation: In this code, a **struct Person** is defined with name and age as members. A pointer ptr is used to store the address of person1. The arrow operator (**->**) is used to access and modify the members of the structure via the pointer, updating the **name** and **age** of person1, and printing the updated values.

An array of structures in C is a data structure that allows us to store multiple records of different data types in a contiguous memory location where each element of the array is a structure. In this article, we will learn how to pass an array of structures from one function to another in C.

Passing an Array of Struct to Functions in C

We can pass an **array of structures** to a **function** in a similar way as we pass an array of any other data type i.e. by passing the array as the pointer to its first element.

Syntax to Pass Array of Struct to Function in C

// using array notation

returnType functionName(struct structName **arrayName[]**, dataType arraySize);

// using pointer notation

returnType functionName(struct structName ***arrayName**, dataType arraySize));

Here,

- **functionName** is the name of the function.
- **structName** is the name of the struct.
- ***arrayName** is a pointer to the array of structures.
- **arraySize** is the size of an array of structures.

C Program to Pass Array of Struct to Function

The below program demonstrates how we can pass an array of structures to a function in C.

C

```
// C Program to pass array of structures to a function

#include <stdio.h>


// Defining the struct

struct MyStruct {

    int id;

    char name[20];
```

```
};

// Function to print the array of structures
void printStructs(struct MyStruct* array, int size)
{
    for (int i = 0; i < size; i++) {
        printf("Struct at index %d: ID = %d, Name = %s\n",
               i, array[i].id, array[i].name);
    }
}

int main()
{
    // Declaring an array of structs
    struct MyStruct myArray[] = {
        { 1, "P1" },
        { 2, "P2" },
    };

    // Passing the array of structures to the function
```

```
    printStructs(myArray, 2);


    return 0;
}
```

Output

Struct at index 0: ID = 1, Name = P1

Struct at index 1: ID = 2, Name = P2

Structure Pointer in C

A structure pointer is a pointer variable that stores the address of a structure. It allows the programmer to manipulate the structure and its members directly by referencing their memory location rather than passing the structure itself. In this article let's take a look at structure pointer in C.

Let's take a look at an example:

```
#include <stdio.h>

struct A {
  int var;
};
```

```
int main() {
  struct A a = {30};

  // Creating a pointer to the structure
  struct A *ptr;

  // Assigning the address of person1 to the pointer
  ptr = &a;

  // Accessing structure members using the pointer
  printf("%d", ptr->var);

  return 0;
}
```

Output

30

Explanation: In this example, ptr is a pointer to the structure A. It stores the address of the structure a, and the structure's member var is accessed using the pointer with the -> operator. This allows efficient access to the structure's members without directly using the structure variable

Syntax of Structure Pointer

The syntax of structure pointer is similar to any other pointer to variable:

*struct struct_name **ptr_name;***

Here, **struct_name** is the name of the structure, and **ptr_name** is the name of the pointer vs variable

Accessing Member using Structure Pointers

There are two ways to access the members of the structure with the help of a structure pointer:

1. Differencing and Using (.) Dot Operator.
2. Using (->) Arrow operator.

Differencing and Using (.) Dot Operator

First method is to first dereference the structure pointer to get to the structure and then use the dot operator to access the member. Below is the program to access the structure members using the structure pointer with the **help of the dot operator.**

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

struct Student {
  int roll_no;
  char name[30];
  char branch[40];
  int batch;
};

int main() {
```

```
  struct Student s1 = {27, "Geek", "CSE", 2019};
  // Pointer to s1
  struct Student* ptr = &s1;

 // Accessing using dot operator
  printf("%d\n", (*ptr).roll_no);
  printf("%s\n", (*ptr).name);
  printf("%s\n", (*ptr).branch);
  printf("%d", (*ptr).batch);

  return 0;
}
```

Output

27

Geek

CSE

2019

Using (->) Arrow Operator

C language provides an array operator (->) that can be used to directly access the structure member without using two separate operators. Below is the program to access the structure members using the structure pointer with the **help of the Arrow operator.**

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

struct Student {
  int roll_no;
  char name[30];
  char branch[40];
  int batch;
};

int main() {

  struct Student s1 = {27, "Geek", "CSE", 2019};
  // Pointer to s1
  struct Student* ptr = &s1;

 // Accessing using dot operator
  printf("%d\n", ptr->roll_no);
  printf("%s\n", ptr->name);
  printf("%s\n", ptr->branch);
  printf("%d", ptr->batch);

  return 0;
}
```

Output

27

Geek

CSE

2019

Explanation: In this code, a **struct Person** is defined with name and age as members. A pointer ptr is used to store the address of person1. The arrow operator (**->**) is used to access and modify the members of the structure via the pointer, updating the **name** and **age** of person1, and printing the updated values.

Sure! Let me explain **self-referential pointers in C** in a simple and easy way.

🔁 What is a Self-Referential Pointer?

A **self-referential pointer** is a pointer that **points to a structure of the same type** as the one in which it is declared.

It is commonly used in **linked lists, trees, and other data structures**.

📦 Basic Example:

```
struct Node {
    int data;
    struct Node* next;  // <-- Self-referential pointer
```

```
};
```

- Here, struct Node* next; is a pointer that points to **another variable of the same type struct Node**.
- This is what makes it **self-referential**.

💡 Why Use It?

Self-referential pointers help you **link multiple structures together**.

Example: In a **linked list**, each node points to the next node using a self-referential pointer.

🧪 Full Example with Linked List Node:

```
#include <stdio.h>

#include <stdlib.h>


struct Node {

    int data;

    struct Node* next;  // Self-referential pointer

};


int main() {
```

```
    struct Node* head = NULL;
    struct Node* second = NULL;

    // Allocate memory
    head = (struct Node*)malloc(sizeof(struct Node));
    second = (struct Node*)malloc(sizeof(struct Node));

    // Set data and link
    head->data = 10;
    head->next = second;

    second->data = 20;
    second->next = NULL;

    // Print values
    printf("First node data: %d\n", head->data);
    printf("Second node data: %d\n", head->next->data);

    return 0;
}
```

🔁 In Summary:

Term	Meaning
struct Node* next;	Self-referential pointer
Purpose	To connect similar structures
Used in	Linked lists, trees, stacks, queues

Sure! Let's understand **Nested Structures in C** in a simple and easy way.

🧩 What is a Nested Structure?

A **nested structure** means having a **structure inside another structure**.

It's like putting one box inside another box.

✅ Syntax:

```
struct Outer {
    int x;
    struct Inner {
        int y;
    } inner;
```

```
};
```

Or if the inner structure is defined separately:

```
struct Inner {
    int y;
};

struct Outer {
    int x;
    struct Inner inner;
};
```

💡 **Example with Code:**

```
#include <stdio.h>

struct Address {
    char city[20];
    int pincode;
```

```
};

struct Student {
    char name[20];
    int age;
    struct Address addr;  // Nested structure
};

int main() {
    struct Student s = {"Srinjoy", 20, {"Kolkata", 700001}};

    // Accessing nested members
    printf("Name: %s\n", s.name);
    printf("Age: %d\n", s.age);
    printf("City: %s\n", s.addr.city);
    printf("Pincode: %d\n", s.addr.pincode);

    return 0;
}
```

🧠 How to Access Nested Members?

```
s.addr.city       // Access city inside Address inside Student
s.addr.pincode    // Access pincode
```

🔁 Summary Table:

Concept	Meaning
Nested Structure	Structure inside another structure
Access Syntax	outer.inner.member
Use Cases	Student info, employee with address, etc.
