**C Structures**

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 **members** and they can be of any valid data type. Applications of structures involve creating data structures Linked List and Tree. Structures in C are also used to represent real world objects in a software like Students and Faculty in a college management software.

**Example:**

#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;

}

**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**](https://www.geeksforgeeks.org/courses/c-Programming-basic-to-advanced?utm_campaign=287_c_structures&utm_medium=gfgcontent_icp&utm_source=geeksforgeeks) 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*** *structure\_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**](https://www.geeksforgeeks.org/c/dot-operator-in-c/). This is applicable when we are using structure variables directly.

*structure\_name . member1;  
structure\_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.

#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 structure

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

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 s = {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](https://www.geeksforgeeks.org/c/c-programming-language-standard/).

***struct*** *structure\_name str = { .member1 = value1, .member2 = value2, .member3 = value3 };*

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

**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](https://www.geeksforgeeks.org/c/typedef-in-c/) 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.

#include <stdio.h>

​

// Defining structure

typedef struct {

int a;

} str1;

​

// Another way of using typedef with structures

typedef struct {

int x;

} str2;

​

int main() {

// Creating structure variables using new names

str1 var1 = { 20 };

str2 var2 = { 314 };

​

printf("var1.a = %d\n", var1.a);

printf("var2.x = %d\n", var2.x);

return 0;

}

**Output**

var1.a = 20

var2.x = 314

**Explanation:** In this code, **str1**and **str2**are defined as aliases for the unnamed structures, allowing the creation of structure variables (**var1**and **var2**) using these new names. This simplifies the syntax when declaring variables of the structure.

**Size of Structures**

Technically, the size of the structure in C should be the sum of the sizes of its members. But it may not be true for most cases. The reason for this is Structure Padding.

**Structure padding** is the concept of adding multiple empty bytes in the structure to naturally align the data members in the memory. It is done to minimize the CPU read cycles to retrieve different data members in the structure.

There are some situations where we need to pack the structure tightly by removing the empty bytes. In such cases, we use**Structure Packing.** C language provides two ways for structure packing:

1. Using #pragma pack(1)
2. Using \_\_attribute((packed))\_\_

To know more about structure padding and packing, refer to this article - [Structure Member Alignment, Padding and Data Packing](https://www.geeksforgeeks.org/c/structure-member-alignment-padding-and-data-packing/).

**Nested Structures**

In C, a [nested structure](https://www.geeksforgeeks.org/c/nested-structure-in-c-with-examples/) refers to a structure that contains another structure as one of its members. This allows you to create more complex data types by grouping multiple structures together, which is useful when dealing with related data that needs to be grouped within a larger structure.

There are two ways in which we can nest one structure into another:

* **Embedded Structure Nesting:**The structure being nested is also declared inside the parent structure.
* **Separate Structure Nesting:**Two structures are declared separately and then the member structure is nested inside the parent structure.

**Accessing Nested Members**

We can access nested Members by using the same ( . ) dot operator two times as shown:

*str\_parent . str\_child . member;*

**Example**

#include <stdio.h>

​

// Child structure declaration

struct child {

int x;

char c;

};

​

// Parent structure declaration

struct parent {

int a;

struct child b;

};

​

int main() {

struct parent p = { 25, 195, 'A' };

​

// Accessing and printing nested members

printf("p.a = %d\n", p.a);

printf("p.b.x = %d\n", p.b.x);

printf("p.b.c = %c", p.b.c);

return 0;

}

**Output**

p.a = 25

p.b.x = 195

p.b.c = A

**Explanation:** In this code, the structure **parent**contains another structure **child**as a member. The **parent**structure is then initialized with values, including the values for the child structure's members.

**Structure Pointer**

A pointer to a structure allows us to access structure members using the [**( -> ) arrow operator**](https://www.geeksforgeeks.org/c/arrow-operator-in-c-c-with-examples/) instead of the dot operator.

#include <stdio.h>

​

// Structure declaration

struct Point {

int x, y;

};

​

int main() {

struct Point p = { 1, 2 };

​

// ptr is a pointer to structure p

struct Point\* ptr = &p;

​

// Accessing structure members using structure pointer

printf("%d %d", ptr->x, ptr->y);

​

return 0;

}

**Output**

1 2

**Explanation:** In this example, **ptr**is a pointer to the structure **Point**. It holds the address of the structure variable **p**. The structure members **x** and **y** are accessed using the **-> operator**, which is used to dereference the pointer and access the members of the structure.

**Self-Referential Structures**

The [self-referential structures](https://www.geeksforgeeks.org/dsa/self-referential-structures/) are those structures that contain references to the same type as themselves i.e. they contain a member of the type pointer pointing to the same structure type.

**Example:**

*struct str {  
int mem1;*

*// Reference to the same type  
struct str\* next;  
};*

Such kinds of structures are used in different data structures such as to define the nodes of linked lists, trees, etc.

**Bit Fields**

[Bit Fields](https://www.geeksforgeeks.org/c/bit-fields-c/) are used to specify the length of the structure members in bits. When we know the maximum length of the member, we can use bit fields to specify the size and reduce memory consumption.

**Syntax**

***struct*** *structure\_name {  
data\_type member\_name: width\_of\_bit-field;  
};*

**Uses of Structure in C**

C structures are used for the following:

1. The structure can be used to define the custom data types that can be used to create some complex data types such as dates, time, complex numbers, etc. which are not present in the language.
2. It can also be used in data organization where a large amount of data can be stored in different fields.
3. Structures are used to create data structures such as trees, linked lists, etc.
4. They can also be used for returning multiple values from a function.