**Structures in C++**

* In C++, a struct (short for structure) is a user-defined data type that allows for the **grouping of related variables of different data types under a single name.**
* This enhances code organization and data encapsulation.
* Unlike an array, a structure can contain many different data types: int , string , bool , etc. .
* Each variable in the structure is known as a member of the structure.

Key characteristics of C++ structures:a

* **User-defined data type:**

A struct creates a new data type that can be used to declare variables.

* **Grouping of members:**

It groups multiple variables, known as members or fields, which can be of various data types (e.g., int, float, string, or even other structures).

* **Default access specifier:**

*Members of a struct are****public****by default*, meaning they can be accessed directly from outside the struct.

* **Memory allocation:**

Defining a struct does not allocate memory; memory is allocated only when a variable of that struct type is created.

**Syntax**

**struct** StructureName {

DataType member1;

DataType member2;

// ... more members

};

**Example:**

#include <iostream>

#include <string>

using namespace std;

// Define a structure named 'Person'

struct Person {

string name;

int age;

double salary;

};

int main() {

// Create a variable of type 'Person'

Person person1, Person2;

// Access and assign values to members using the dot operator

person1.name = "Alice";

person1.age = 30;

person1.salary = 50000.0;

person2.name = "Jeeva";

person2.age = 25;

person2.salary = 75000.0;

// Access and print member values

cout << "Name: " << person1.name << endl;

cout << "Age: " << person1.age << endl;

cout << "Salary: " << person1.salary << endl;

cout << "Name: " << person2.name << endl;

cout << "Age: " << person2.age << endl;

cout << "Salary: " << person2.salary << endl;

return 0;

}

**Member Functions**

In C structures, functions were not allowed inside the structure but in latest C++ ( V11+ ), we can declare the function inside the structure. They are called **member functions** while the variables are called **data members (member data)**. C++ structure is way more similar to [C++ classes](https://www.geeksforgeeks.org/cpp/c-classes-and-objects/) as compared to C structures.

**Example:**

#include <iostream>

using namespace std;

struct Point {

int x, y;

// Member function

int sum(){

return x + y;

}

};

int main() {

// Point s = { 5, 8 }; Way 1

// Way 2

Point s;

s.x = 5;

s.y = 8;

// Call member function using (.) operator

cout << s.sum();

return 0;

}

**Constructors and Destructors**

C++ structures also support other class components such as constructor, destructor, access specifiers, etc.

#include <iostream>

using namespace std;

// Class like structure

struct Point {

private:

int x, y;

public:

// Constructors

Point() {

x = 100;

y = 200;

}

Point(int a, int b) {

x = a;

y = b;

}

// Member function

void show() {

cout << x << " " << y << endl;

}

// Destructor

~Point() {

cout << "Destroyed Point Variable" << endl;

}

};

int main() {

// Creating Point variables using constructors

Point s1;

Point s2(99, 1001);

s1.show();

s2.show();

return 0;

}

**Nested Structure**

**Nested structure** in C++ refers to a structure that is defined inside another structure. Just as structure members are declared within a structure, one structure can be declared as a member inside another structure.

#include <iostream>

using namespace std;

// Define inner structure

struct inner {

int a, b;

};

// Define the outer structure that

// contains inner structure

struct outer {

// Nested structure

inner in;

int x, y;

};

int main() {

outer obj = {{1, 2}, 10, 20};

cout << "Inner: " << obj.in.a << " "

<< obj.in.b << endl;

cout << "Outer: " << obj.x << " "

<< obj.y<<endl;

outer obj1;

obj1.in.a = 100;

obj1.in.b = 200;

obj1.x = 300;

obj1.y = 400;

cout << "Inner: " << obj1.in.a << " "

<< obj1.in.b << endl;

cout << "Outer: " << obj1.x << " "

<< obj1.y << endl;

return 0;

}

**Array of structures**

In C++, an array of structures allows for the storage and organization of multiple instances of a custom data type (a structure) in a contiguous block of memory. This combines the benefits of structures (grouping related data of different types) with the efficiency of arrays (storing multiple elements of the same type sequentially).

**Array of Structures in C++**

An array of structures combines both an array and a structure to handle complex data neatly. Instead of creating many separate variables of structure, we use an array of structures. Each element in this array is a structure on its own and since the array elements are stored continuously in memory, it allows for quick access and makes our program more efficient and straightforward.

// C++ program to demonstrate the use of array of structure.

#include <cstring>

#include <iostream>

using namespace std;

// defining struct coordinates

struct Coordinates {

int x;

int y;

};

int main()

{

// Declare an array of structures

int size = 3;

Coordinates vertices[size];

// Assign values to elements in the array

vertices[0].x = 2;

vertices[0].y = 4;

vertices[1].x = 3;

vertices[1].y = 6;

vertices[2].x = 4;

vertices[2].y = 8;

Coordinates vertices1[size];

for(int i=0; i<3; i++)

{

cout<<”Input x and y coordinates \n”;

cin >> vertices1[i].x >> vertices1[i].y;

}

// Displaying the values stored in the array of

// structures

for (int i = 0; i < size; ++i) {

cout << "Coordinates of ( x , y ) " << i + 1

<< ": (" << vertices[i].x << ", "

<< vertices[i].y << ")" << endl;

}

for (int i = 0; i < size; ++i) {

cout << "Coordinates of ( x , y ) " << i + 1

<< ": (" << vertices1[i].x << ", "

<< vertices1[i].y << ")" << endl;

}

return 0;

}

**Array Within a Structure in C++**

An array within a structure simply means that we can create one or more arrays inside a structure as structure member which can be useful when we want to associate a collection of items within a single [structure.](https://www.geeksforgeeks.org/structures-in-cpp/)

// C++ program to demonstrate the use of array within

// structure

#include <iostream>

#include <string>

using namespace std;

// Definingstructure

struct Student {

string name;

int grades[5]; // Array to store grades for 5 subjects

};

int main()

{

// Creating an instance of Student

Student student1;

// Initializing the student's name

student1.name = "Jack";

// Initializing the student's grades

student1.grades[0] = 85;

student1.grades[1] = 92;

student1.grades[2] = 76;

student1.grades[3] = 81;

student1.grades[4] = 90;

// Output the student's information

cout << "Student Name: " << student1.name << endl;

cout << "Grades: ";

for (int grade : student1.grades) {

cout << grade << " ";

}

cout << endl;

return 0;

}

**Difference Between Array of Structures and Array Within a Structure in C++**

| **Feature** | **Array of Structures** | **Array of Structures** |
| --- | --- | --- |
| **Definition** | It is declared as an array where each element is a structure. | A structure contains an array as one of its members. |
| **Syntax** | struct Point { int x; int y; };  Point points[3]; | struct Person { char name[50];int grades[5]; };  Person person1; |
| **Memory Allocation** | Memory is allocated for every structure element separately. | Memory is allocated for the entire structure, including the array within it. |
| **Memory Efficiency** | It can be a memory-saver if structures have a fixed size and minimal unused space. | It may have higher memory overhead, especially if arrays within structures vary in size. |
| **Code Readability** | Enhances code readability by providing a clear structure for related data. | Improves code readability by encapsulating arrays within the context of a single structure. |
| **Accessing Data** | Enhances code readability by providing a clear structure for related data. | Accessing data involves indexing the structure and then the array within it. |

**Union**

A union in C++ is a **user-defined data type that allows different data types to be *stored in the same memory location*.** *Unlike a struct, where each member occupies its own distinct memory space, all members of a union share the same starting address in memory.*

Key characteristics of unions:

* **Memory Sharing:** All members of a union share the same memory location.
* **Size Determination:** The size of a union is determined by the size of its largest member.
* **Single Active Member:** Only one member of a union can be "active" at any given time. Assigning a value to one member will overwrite the data of any previously active member in that shared memory space. This can lead to data loss if not managed carefully.
* **Memory Efficiency:** Unions are primarily used for memory optimization, especially when dealing with situations where only one type of data needs to be stored at a time within a specific context, but the exact type might vary.
* **Syntax:** Unions are declared using the **union** keyword, similar to structs.

union union\_Name {

type1 member1;

type2 member2;

.

.

};

**Size of Union**

All the elements of the union are stored in the same memory location. So, if the size of the union is equal to the**size of the largest element**, it would be able to store the largest member, smallest member and all members in between in the same memory location

**Nested Union**

union outer {

union inner {

type member;

};

};

Nested members are then accessed by using **(.) dot operator**.

**outer.inner.member**

**Example**

#include <iostream>

union Data {

int i;

float f;

char str[20];

};

int main() {

Data d;

d.i = 10;

std::cout << "Integer value: " << d.i << std::endl;

d.f = 20.5f; // Overwrites the integer value

std::cout << "Float value: " << d.f << std::endl;

// std::cout << "Integer value after float assignment: " << d.i << std::endl; // This would likely show garbage

// Assigning to str will overwrite float value

// std::strcpy(d.str, "Hello Union");

// std::cout << "String value: " << d.str << std::endl;

return 0;

}

In this example, d.i, d.f, and d.str all occupy the same memory space. When d.f is assigned a value, it overwrites the value previously stored in d.i. Attempting to access d.i after d.f has been modified would result in undefined behavior or a "garbage" value, as the memory now holds the representation of a float.

#include <iostream>

using namespace std;

// Define a structure containing a nested union

struct Employee {

char name[50];

int id;

// Nested union

union pay {

float hourlyRate;

float salary;

} payment;

};

int main() {

struct Employee e1;

e1.id = 101;

// Access nested union member

e1.payment.hourlyRate = 300;

cout << "Employee ID: " << e1.id << "\n";

cout << "Hourly Rate: Rs " << e1.payment.hourlyRate;

return 0;

}

**enum in C++**

In C++, an enum (enumeration) is a user-defined data type that provides a way **to assign names to a set of integral constant**s. It enhances code readability and maintainability by allowing you to use descriptive names instead of raw integer values.

Declaration and Usage:

* **Declaration:** An enumeration is declared using the **enum**keyword, followed by the enum's name and a list of identifiers (enumerators) enclosed in curly braces.

enum Color {

RED, // Default value is 0

GREEN, // Default value is 1

BLUE // Default value is 2

};

NOTE: You can also explicitly assign integer values to enumerators:

enum Status {

SUCCESS = 0,

ERROR = 100,

WARNING = 200

};

* **Usage:** You can declare variables of the enum type and assign them one of the defined enumerators.

Color myColor = RED;

Status currentStatus = SUCCESS;

Benefits of using enum:

* **Readability:**

Replaces "magic numbers" with meaningful names, making the code easier to understand.

* **Maintainability:**

Changes to constant values only require modifying the enum definition, not every instance where the value is used.

* **Type Safety (with enum class):**

C++11 introduced enum class (scoped enums), which provides stronger type safety by preventing implicit conversions to integers and avoiding name conflicts in the global scope.

enum class Direction {

NORTH,

SOUTH,

EAST,

WEST

};

// Usage:

Direction travelDirection = Direction::NORTH;

Example:

#include <iostream>

using namespace std;

// Defining enum

enum fruit {

APPLE, BANANA = 5, ORANGE

};

int main() {

// Creating enum variable

fruit f = BANANA;

cout << f << endl;

// Changing the value

f = ORANGE;

cout << f;

return 0;

}

**Enum Classes**

C++11 introduced enum class, which provides better type safety. It helps in resolving name conflicts by providing scope to the constant names. It also requires manual typecasting to integer values from names.

**Create enum Class**

Enum class can be created just by adding the **class**keyword in the enum delcaration.

#include <iostream>

using namespace std;

// Define the enum class

enum class Day { Sunday = 1, Monday, Tuesday,

Wednesday, Thursday, Friday,

Saturday };

int main() {

// initializing

Day today = Day::Thursday;

// Print the enum

cout << static\_cast<int>(today);

return 0;

}