1. Introduction

In C++, **structure** and **union** are user-defined data types that allow grouping of different types of variables under a single name. They help in organizing data efficiently and improving code readability.

- **Structure:** Stores multiple variables of different types as a single unit.
- **Union:** Stores multiple variables of different types, but only one variable can hold a value at a time.

Both are useful for representing complex real-world entities like student records, employee details, etc.

---

2. Structure in C++

A structure is a collection of variables of **different data types grouped together under one name**. It is defined using the `struct` keyword.

Syntax

```
struct StructureName {
   data_type member1;
   data_type member2;
   ...
};
```

---

3. Declaration of Structure Variables

After defining a structure, you can declare variables of that type.

Example

```
struct Student {
   int roll;
   char name[20];
   float marks;
};

Student s1, s2;
```

`s1` and `s2` are two variables of type `Student`.

---

4. Accessing Structure Members

Members of a structure are accessed using the **dot operator (.)**.

Example

```
s1.roll = 101;
strcpy(s1.name, "Ravi");
s1.marks = 95.5;
```

5. Initialization of Structures

Structures can be initialized at the time of declaration.

```
Student s1 = {101, "Ravi", 95.5};
```

6. Nested Structures

A structure can contain **another structure** as a member.

Example

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

struct Student {
   int roll;
   char name[20];
   Date dob;
};
```

7. Array of Structures

You can create an array of structures to store multiple records.

Example

```
Student students[50];
```

Access individual elements using index:

```
students[0].roll = 101;
```

8. Pointers to Structures

- Pointers can be used to refer to structures
- Access members using the **arrow operator (->)**

Example

```
Student s1 = {101, "Ravi", 95.5};
Student *ptr = &s1;
cout << ptr->roll;
```

9. Functions and Structures

Structures can be passed to functions by:

- **Value:** A copy is passed
- **Reference (pointer):** Original structure can be modified

Example

```
void display(Student s) { cout << s.roll; }
```

---

10. Union in C++

A union is similar to a structure, but **only one member can store a value at a time**. It shares the same memory location for all its members.

- Defined using the union keyword
- Efficient memory usage
- Useful when only one of the variables is required at a time

---

11. Syntax of Union

```
union UnionName {
   int intVar;
   float floatVar;
   char charVar;
};
```

---

12. Declaration and Access

```
union Data {
   int i;
   float f;
   char c;
};

Data d;
d.i = 10;
d.f = 5.5; // overwrites previous value
```

Access members using the **dot operator**.

---

13. Size of Union

The size of a union is **equal to the size of its largest member**. This is because all members share the same memory location.

---

14. Differences Between Structure and Union

Feature	Structure	Union
Memory allocation	Separate for each member	Shared among members
Size	Sum of sizes of all members	Size of largest member
Usage	Store all members simultaneously	Store only one member at a time
Access	Dot operator (.)	Dot operator (.)

---

15. Nested Unions

Unions can also contain other unions or structures as members.

Example

```
union Data {
   int i;
   struct {
      char c;
      float f;
   } inner;
};
```

---

16. Structures vs Classes

- Classes in C++ are similar to structures but provide **encapsulation, access specifiers, constructors, and destructors**
- Structures are primarily used for simple data grouping

---

17. Typedef with Structures/Unions

typedef can be used to create a new type name for structure or union.

Example

```
typedef struct Student {
   int roll;
   char name[20];
} Stu;

Stu s1; // no need to write 'struct Student'
```

---

18. Advantages of Structures

- Organizes data logically
- Simplifies complex data
- Supports arrays and pointers
- Can be nested

---

19. Advantages of Unions

- Efficient memory usage
- Useful for variant data storage
- Simplifies certain programming scenarios

---

20. Applications

Structures:

- Student, Employee, Product records
- Database management
- Game programming

Unions:

- Storing variant data types
- Memory-efficient design
- Embedded systems
- File format parsers

---

21. Common Mistakes

- Accessing multiple union members simultaneously
- Not initializing members of structure/union
- Confusing structures and unions
- Forgetting to use pointers correctly

---

22. Best Practices

- Use structures for grouping related data
- Use unions when memory optimization is required
- Prefer classes for advanced OOP concepts

- Initialize members properly

---

23. Conclusion

Structures and unions are **powerful tools for data organization** in C++.

- **Structures:** Store multiple values simultaneously
- **Unions:** Save memory by storing only one value at a time

Understanding their usage is essential for efficient programming, memory management, and creating real-world applications.