

Structure, Union, and Enum in C++

1. Structure (**struct**)

- A **structure** is a user-defined data type that groups different data types together under one name.
- Each member of a structure has its own memory.
- **Syntax:**

```
struct Student {  
    int rollNo;  
    char name[50];  
    float marks;  
};
```

Key Points:

- Members are **stored in separate memory locations**.
- Total memory = sum of sizes of all members.
- Access members using `.` operator:

```
Student s1;  
s1.rollNo = 101;  
s1.marks = 88.5;
```

2. Union (**union**)

- A **union** is also a user-defined data type but with a key difference:
 - **All members share the same memory location.**
- Only **one member can hold a value at a time**.
- **Syntax:**

```
union Data {  
    int intVal;  
    float floatVal;  
    char charVal;  
};
```

Key Points:

- Memory allocated = size of the **largest member**.
- Changing one member's value affects the others.
- Access:

```
Data d;  
d.intVal = 10;  
d.floatVal = 12.5; // overwrites intVal
```

3. Enumeration (**enum**)

- An **enum** is a user-defined type consisting of a set of named integral constants.

- Useful for making code more readable.
- Syntax:

```
enum Color { Red, Green, Blue };
```

Key Points:

- Default values start from 0, 1, 2...
- You can assign custom values:

```
enum Weekday { Mon = 1, Tue, Wed, Thu, Fri, Sat, Sun };
```

- Access:

```
Color c = Green;    // c = 1
Weekday w = Fri;    // w = 5
```

Difference Between Structure and Union

| Feature | Structure | Union |
|---------|---|---|
| Memory | Each member has its own storage | All members share the same memory |
| Size | Sum of all members | Size of the largest member |
| Usage | Used when multiple values needed together | Used when only one value needed at a time |

Summary:

- **Structure** → group different data, all active together.
- **Union** → save memory, only one member active at a time.
- **Enum** → define symbolic names for integral values.

typedef in C++

- The `typedef` keyword in C++ is used to create an **alias (alternative name)** for an existing data type.
- It helps make code more **readable, shorter, and easier to manage**, especially for complex data types.

Syntax

```
typedef existing_type new_name;
```

Examples

1. Basic Usage

```
typedef unsigned int uint;
uint a = 10, b = 20;    // same as: unsigned int a, b;
```

Functions in C++

A **function** in C++ is a block of code that performs a specific task, can be reused, and may return a value.

Types of Functions

1. Library Functions

Predefined functions provided by C++ standard libraries.

Example: `sqrt()`, `strlen()`, `printf()` etc.

2. User-defined Functions

Created by programmers to perform specific tasks.

Example:

```
int add(int a, int b) {  
    return a + b;  
}
```

Advantages of Functions

- Increases code reusability.
- Makes program modular (easy to understand & maintain).
- Reduces code redundancy.
- Easier debugging and testing.

Syntax of a Function

```
returnType functionName(parameter1, parameter2, ...) {  
    // Function body  
    // Statements  
    return value;    // if returnType is not void  
}
```

Example:

```
int square(int num) {  
    return num * num;  
}
```

Function Prototyping

A **function prototype** tells the compiler about the function's name, return type, and parameters **before the function is actually defined**.

- It ensures the compiler checks for correct arguments while calling a function.

Syntax:

```
returnType functionName(parameterType1, parameterType2, ...);
```

Example:

```
#include <iostream>
using namespace std;

// Function prototype (declaration)
int add(int, int);

int main() {
    int result = add(10, 20); // Function call
    cout << "Sum = " << result;
    return 0;
}

// Function definition
int add(int a, int b) {
    return a + b;
}
```

Function Components

Declaration (Prototype) – Declares function.

Definition – Contains actual body of function.

Call – Executes the function.

Types of User-Defined Functions

1. **Function with no arguments and no return value**
2. **Function with arguments but no return value**
3. **Function with no arguments but return value**
4. **Function with arguments and return value**

1. Function with No Arguments and No Return Value

- Function does not take any input (arguments).
- Function does not return any output.

Example:

```
#include <iostream>
using namespace std;

void displayMessage() { // no arguments, no return value
    cout << "Hello! This is a simple function." << endl;
}

int main() {
    displayMessage(); // function call
    return 0;
}
```

2. Function with Arguments but No Return Value

- Function takes input values (arguments).
- Function does not return any output.

Example:

```
#include <iostream>
using namespace std;

void printSum(int a, int b) {    // arguments, no return value
    cout << "Sum = " << (a + b) << endl;
}

int main() {
    printSum(10, 20);    // function call with arguments
    return 0;
}
```

3. Function with No Arguments but Return Value

- Function does not take any input.
- Function returns a value.

Example:

```
#include <iostream>
using namespace std;

int getNumber() {    // no arguments, return value
    return 100;    // returns an integer
}

int main() {
    int num = getNumber();    // function call
    cout << "Number = " << num << endl;
    return 0;
}
```

4. Function with Arguments and Return Value

- Function takes input values (arguments).
- Function also returns a value.

Example:

```
#include <iostream>
using namespace std;

int multiply(int x, int y) {    // arguments and return value
    return x * y;
}

int main() {
    int result = multiply(5, 4);    // function call
}
```

```

    cout << "Product = " << result << endl;
    return 0;
}

```

Call by Value and Call by Reference in C++

1. Call by Value

- In this method, **a copy of actual arguments** is passed to the function.
- Changes made inside the function **do not affect the original values**.
- Safe but less efficient for large data.

Example:

```

#include <iostream>
using namespace std;

void modify(int x) {
    x = x + 10;    // changes only local copy
    cout << "Inside function: " << x << endl;
}

int main() {
    int num = 5;
    modify(num);   // call by value
    cout << "Outside function: " << num << endl;
    return 0;
}

```

Output:

```

Inside function: 15
Outside function: 5

```

2. Call by Reference

- In this method, **the address (reference) of actual arguments** is passed to the function.
- Changes made inside the function **directly affect the original values**.
- Efficient when working with large data structures.

Example:

```

#include <iostream>
using namespace std;

void modify(int &x) {    // & makes it reference
    x = x + 10;          // changes original value
    cout << "Inside function: " << x << endl;
}

int main() {
    int num = 5;
    modify(num);         // call by reference
    cout << "Outside function: " << num << endl;
    return 0;
}

```

}

Output:

Inside function: 15
Outside function: 15

Comparison Table

| Feature | Call by Value | Call by Reference |
|-------------------------|--|---|
| Argument Passed | Copy of the value | Reference (address) of the variable |
| Effect on Original Data | No change in actual variable | Changes affect the actual variable |
| Memory Usage | More (because copies are created) | Less (works on original data) |
| Safety | Safer (original values remain unchanged) | Less safe (accidental changes possible) |
| Use Case | Small data, when original data should remain unchanged | Large data, when modifications are required |

Function Overloading in C++

Function Overloading in C++ is the ability to define **multiple functions with the same name but different parameter lists**.

- Compiler decides which function to call based on the **number and type of arguments**.
- It is a form of **compile-time polymorphism**.

Rules of Function Overloading

1. Functions must differ in **number of arguments OR type of arguments**.
2. Return type **alone cannot distinguish functions**.
3. All overloaded functions must have the **same function name**.

Example

```
#include <iostream>
using namespace std;

int add(int a, int b) {          // 2 int parameters
    return a + b;
}

double add(double a, double b) { // 2 double parameters
    return a + b;
}
```

```

int add(int a, int b, int c) {    // 3 int parameters
    return a + b + c;
}

int main() {
    cout << "Sum (int): " << add(5, 10) << endl;
    cout << "Sum (double): " << add(2.5, 3.7) << endl;
    cout << "Sum of 3 ints: " << add(1, 2, 3) << endl;
    return 0;
}

```

Output:

```

Sum (int): 15
Sum (double): 6.2
Sum of 3 ints: 6

```

Default Arguments in C++

Default arguments in C++ allow you to **assign default values to function parameters**.

- If a value is not provided during the function call, the default value is used.

Rules

1. Default values are assigned in the **function declaration**.
2. Once a parameter has a default value, **all parameters to its right must also have default values**.
3. Default arguments are evaluated from **right to left**.

Example

```

#include <iostream>
using namespace std;

int add(int a, int b = 10, int c = 5) {
    return a + b + c;
}

int main() {
    cout << "Call with 1 argument: " << add(2) << endl;           // 2 + 10 + 5
    cout << "Call with 2 arguments: " << add(2, 3) << endl;       // 2 + 3 + 5
    cout << "Call with 3 arguments: " << add(2, 3, 4) << endl;    // 2 + 3 + 4
    return 0;
}

```

Output:

```

Call with 1 argument: 17
Call with 2 arguments: 10
Call with 3 arguments: 9

```

Comparison: Function Overloading vs Default Arguments

| Feature | Function Overloading | Default Arguments |
|-------------|---|---|
| Definition | Multiple functions with same name but different parameter lists | Single function with parameters having default values |
| Flexibility | More control, can have different logic | Same logic with optional parameters |
| Readability | Multiple function definitions | Single compact function definition |
| Use Case | When different operations are required based on parameter types/count | When some parameters usually have common/default values |

Inline Functions in C++

An **inline function** is a function where the compiler replaces the function call with the actual function code at compile-time.

- It reduces the overhead of function calls.
- Best used for **small and frequently used functions**.

Syntax

```
inline returnType functionName(parameters) {
    // function body
}
```

Example

```
#include <iostream>
using namespace std;

inline int square(int x) {
    return x * x;
}

int main() {
    cout << "Square of 5 = " << square(5) << endl;
    cout << "Square of 5 = " << square(5) << endl;
    return 0;
}
```

Key Points

- Useful for **small functions (1–2 lines)**.
- Compiler may ignore `inline` if the function is large or uses recursion/loops.
- Overuse can increase code size.

Static Variables in Functions

A **static variable inside a function** retains its value between function calls.

- Unlike normal local variables, it is initialized **only once**.
- Scope is within the function, but lifetime is throughout the program.

Example

```
#include <iostream>
using namespace std;

void counter() {
    static int count = 0; // static variable
    count++;
    cout << "Function called " << count << " times" << endl;
}

int main() {
    counter();
    counter();
    counter();
    return 0;
}
```

Output:

```
Function called 1 times
Function called 2 times
Function called 3 times
```

Constant Parameters in Functions

When a function parameter is declared as **const**, the function cannot modify that argument inside its body.

- It ensures the argument is **read-only** within the function.
- Commonly used when passing objects or references for safety.

Syntax

```
void functionName(const dataType parameter);
```

Example

```
#include <iostream>
using namespace std;

void display(const int x) {
    // x = x + 10; // Not allowed (x is constant)
    cout << "Value received = " << x << endl;
}

int main() {
    int num = 50;
    display(num);
    return 0;
}
```

Quick Comparison

| Concept | Meaning |
|-----------------|--|
| Inline Function | Replaces function call with code at compile-time (faster, for small code). |
| Static Variable | Retains its value between function calls, initialized only once. |
| Const Parameter | Makes the received argument read-only inside the function. |

Scope and Lifetime of Variables

Definition: Scope of a variable is the region of the program where the variable is accessible.

Types of Scope: Here is 4 types of scope and lifetime of variable in c++.

1. Local Scope:

Variables declared inside a function or block { }.

Accessible only within that block.

Example:

```
void func() {
    int x = 10; // local variable
    cout << x;
}
// x is not accessible here
```

2. Global Scope:

Variables declared outside all functions.

Accessible by all functions in the file (or other files if `extern` is used).

Example:

```
int y = 20; // global variable

void func() {
    cout << y; // accessible
}
```

3. Block Scope:

Any variable inside { } has block scope.

Example:

```
for (int i = 0; i < 5; i++) {
    int a = i; // block scope
}
// a is not accessible here
```

4. Function Scope:

Labels used for `goto` statements have function scope.