## Templates in C++

Templates in C++ allow writing generic code that works with different data types. They enable **code reusability** and **type independence** in **functions** and **classes**.

---

### 1. Function Template (Generic Functions)

A function template allows writing a single function definition that can work with multiple data types.

**Example: Function Template for Finding Maximum**

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

template <typename T> // Template declaration
T findMax(T a, T b) {
   return (a > b) ? a : b;
}

int main() {
   cout << "Max of 10 and 20: " << findMax(10, 20) << endl;
   cout << "Max of 5.5 and 2.3: " << findMax(5.5, 2.3) << endl;
   cout << "Max of 'A' and 'Z': " << findMax('A', 'Z') << endl;
   return 0;
}
```

**Output:**

```
Max of 10 and 20: 20
Max of 5.5 and 2.3: 5.5
Max of 'A' and 'Z': Z
```

**How It Works?**

- <typename T> declares a template with type T.
- findMax(T a, T b) works for **int, float, char**, or any type that supports > operator.

---

### 2. Class Template (Generic Classes)

A class template allows creating a **single class** that can handle different data types.

**Example: Class Template for a Generic Box**

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

template <typename T>
class Box {
private:
   T value;
public:
   Box(T v) { value = v; } // Constructor
   void show() { cout << "Stored Value: " << value << endl; }
};

int main() {
   Box<int> intBox(100);
   Box<double> doubleBox(45.67);
   Box<string> stringBox("Hello Templates");

   intBox.show();
   doubleBox.show();
   stringBox.show();

   return 0;
}
```

**Output:**

```
Stored Value: 100
Stored Value: 45.67
Stored Value: Hello Templates
```

🔷 **How It Works?**

- Box<T> is a template class where T represents **any data type**.
- We create objects like Box<int>, Box<double>, Box<string>.

---

**3. Template with Multiple Parameters**

You can use multiple template parameters.

**Example: Swap Two Different Types**

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

template <typename T1, typename T2>
void swapValues(T1 &a, T2 &b) {
    cout << "Before Swap: " << a << " and " << b << endl;
    T1 temp = a;
    a = b;
    b = temp;
    cout << "After Swap: " << a << " and " << b << endl;
}

int main() {
    int x = 10;
    double y = 5.5;
    swapValues(x, y);
    return 0;
}
```

**Output:**

```
Before Swap: 10 and 5.5
After Swap: 5.5 and 10
```

🔷 **How It Works?**

- T1 and T2 allow swapping **different types**.

---

**4. Specialized Templates (Template Specialization)**

Sometimes, we need different behavior for a specific data type.

**Example: Specialized Template for char***

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

template <typename T>
void show(T data) {
    cout << "Generic Data: " << data << endl;
}

// Specialization for char*
template <>
void show<char*>(char* data) {
    cout << "String Data: " << data << endl;
}

int main() {
    show(100);
    show(45.67);
    char str[] = "Specialized";
    show(str);
    return 0;
}
```

**Output:**

```
Generic Data: 100
Generic Data: 45.67
String Data: Specialized
```

🔷 **How It Works?**

- The generic template works for int and double.
- The specialized version runs **only** for char*.

---