Templates in C++

---

1. What Are Templates?

Templates in C++ allow the creation of **generic code** that works with **any data type**. Instead of writing separate code for each data type (e.g., `int`, `float`, `double`), templates enable writing a single piece of code that adapts to the data type provided during usage.

---

2. Why Use Templates?

1. **Reusability:**
    - Avoid redundant code by creating one generic version of a function or class.
    - E.g., You don't need separate sorting functions for `int` and `float`.
2. **Flexibility:**
    - Write code that works with **any data type** without prior knowledge of what type will be used.
3. **Maintainability:**
    - Changes in logic affect only the generic template, reducing the scope of bugs.
4. **Efficiency:**
    - STL (Standard Template Library) in C++ is built on templates, offering highly optimized implementations.

---

3. How Templates Work

Templates use placeholders for data types. These placeholders are replaced with the actual data type during compilation.

Syntax of Templates

1. **Template Declaration:**

    ```
    template<typename T>
    // or template<class T>
    ```

    - `T` is a placeholder for a data type (it could be any name, but `T` is common).
2. **Template Function:**

    ```
    template<typename T>
    T getMax(T a, T b) {
        return (a > b) ? a : b;
    }
    ```

3. **Using Templates:**

- For a function:

  ```
  cout << getMax<int>(5, 10); // Output: 10
  cout << getMax<double>(3.5, 2.5); // Output: 3.5
  ```

Types of Templates

1. **Function Templates:**
   - For creating generic functions.
   - Example: Find maximum of two numbers.
2. **Class Templates:**
   - For creating generic classes.
   - Example: Create a stack that works with any data type.
3. **Template Specialization:**
   - Customize templates for specific data types.
   - Example: Handle `char` differently from `int` or `float`.
4. **Variadic Templates:**
   - For creating templates that accept a variable number of arguments.

Examples

1. Function Template

```
template<typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    cout << add<int>(3, 4) << endl;      // Output: 7
    cout << add<double>(3.5, 2.5) << endl; // Output: 6.0
    return 0;
}
```

Common Questions and How to Think

1. **When to Use Templates?**
   - Use templates when you need the same logic to work across multiple data types.
2. **What Are the Limitations?**
   - Templates increase compile-time complexity.

- Error messages can be verbose and hard to understand.

3. **How to Debug Template Code?**
    - Use simple examples to isolate template errors.
    - Add explicit type definitions when possible for testing.
4. **How to Optimize Templates?**
    - Avoid over-complicating templates with unnecessary logic.
    - Use template specialization only when necessary.
