Templates

Templates allow writing **generic** and **reusable** code that works with any data type.

1 Function Templates

Function templates allow creating a single function that works with different data types.

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

Usage examples:

```
max(3, 7);         // T is int
max(3.5, 2.1);     // T is double
```

Note: `typename` and `class` are interchangeable in template declarations.

2 Class Templates

Class templates define a blueprint for a class to operate on any data type.

```
template <typename T>
class Box {
    T value;
public:
    void set(T val) { value = val; }
    T get() { return value; }
};
```

Usage examples:

```
Box<int> intBox;
intBox.set(5);

Box<std::string> strBox;
strBox.set("Hello");
```

3 Template Specialization

Template specialization allows customizing behavior for specific types.

```
template <>
class Box<bool> {
    bool value;
public:
    void set(bool val) { value = val; }
    bool get() { return value; }
};
```

4 Non-Type Template Parameters

Templates can also take constant values as parameters.

```
template <typename T, int size>
class Array {
    T arr[size];
};
```

5 Summary of Benefits

Feature	Description
Reusability	Write once, use with any data type
Type Safety	Type checking is enforced at compile time
Performance	No runtime overhead; templates are instantiated at compile time