| **Main Idea** | **Explanation** | **Examples** | **Key Terms** |
| --- | --- | --- | --- |
| **What are Templates?** | Templates are blueprints for creating generic classes or functions. They allow code reuse for different data types. | template<typename T> T add(T a, T b) { return a + b; } | Generic programming, type parameter |
| **Function Templates** | Enable writing generic functions that work for any data type. | template<typename T> T max(T a, T b) { return (a > b) ? a : b; } | Type inference, type-safe functions |
| **Class Templates** | Allow defining classes that operate with any type. | template<typename T> class Stack { T arr[100]; int top = -1; void push(T x) { arr[++top] = x; } }; | Generic class, data abstraction |
| **Template Specialization** | Provides specific implementations of a template for particular data types. | template<> class Stack<bool> { void push(bool x) { /\* custom logic \*/ } }; | Specialization, customization |
| **Non-Type Template Args** | Templates that accept values (e.g., integers) as parameters. | template<int N> int array[N]; | Compile-time constants, flexibility |
| **Variadic Templates** | Allow defining templates that accept a variable number of arguments. | template<typename... Args> void printAll(Args... args) { (cout << ... << args) << endl; } | Pack expansion, flexible arguments |
| **Template Metaprogramming** | Using templates for computations or decisions at compile time. | template<int N> struct Factorial { static const int value = N \* Factorial<N - 1>::value; }; template<> struct Factorial<0> { static const int value = 1; }; | Compile-time logic, constexpr |
| **SFINAE** | "Substitution Failure Is Not An Error" allows enabling/disabling templates based on conditions. | template<typename T> auto enableIfInt(T x) -> typename enable\_if<is\_integral<T>::value, T>::type { return x; } | Overload resolution, enable\_if |
| **Template Constraints (C++20)** | Adds constraints to templates using concepts. | template<typename T> requires std::is\_integral\_v<T> T add(T a, T b) { return a + b; } | Concepts, requires, std::is\_integral |
| **Type Deduction** | Automatic inference of types for template arguments based on function call arguments. | add(5, 3.2); // Deduce T as double | Deduction, type safety |
| **Static Members in Templates** | Static members are shared among all instances of a template instantiated with the same type. | template<typename T> struct Counter { static int count; }; template<typename T> int Counter<T>::count = 0; | Shared data, independent per type |
| **Template Instantiation** | Process by which the compiler generates specific code for a template based on its arguments. | Stack<int> s1; Stack<float> s2; // Instantiates Stack for int and float | Explicit instantiation, on-demand |
| **CRTP (Curiously Recurring Template Pattern)** | Design pattern where a class uses itself as a template parameter to its base class. | template<class Derived> class Base { void interface() { static\_cast<Derived\*>(this)->implementation(); } }; class Derived : public Base<Derived> { void implementation() { /\*...\*/ } }; | Self-referencing templates, design pattern |
| **Template Aliases (C++11)** | Define a shorthand for long template declarations using using. | template<typename T> using Vec = std::vector<T>; Vec<int> v; | Alias templates, cleaner syntax |