# Chapter 12: Templates and Generic Programming

**Why use templates?** Templates solve the problem of code duplication in a type-safe way. Before templates, if you needed a function to find the maximum of two `int`s and another for two `double`s, you had to write two nearly identical functions. Templates allow you to write the generic algorithm *once*, and the compiler generates the specific versions for you as needed. This is **compile-time polymorphism**—polymorphism that is resolved during compilation, resulting in highly efficient code.

## 12.1 Function and Class Templates

Templates are blueprints that the compiler uses to generate code.

### Function Templates

A function template is a blueprint for a family of functions. The `typename` (or `class`) keyword introduces a placeholder for a type that will be specified when the function is called.

```
// A function template for finding the maximum of two values
template <typename T>
T my_max(T a, T b) {
    return (a > b) ? a : b;
}

// How it's used:
int i = my_max(5, 10);          // Compiler sees two ints, instantiates my_max<int>
double d = my_max(3.14, 2.71);  // Compiler sees two doubles, instantiates my_max<double>
```

### Class Templates

Similarly, a class template is a blueprint for a family of classes. This is the foundation of all STL containers.

```
// A class template for a generic Pair
template <typename T1, typename T2>
class Pair {
public:
    Pair(T1 first, T2 second) : first_(first), second_(second) {}

    T1 get_first() const { return first_; }
    T2 get_second() const { return second_; }
private:
    T1 first_;
    T2 second_;
};

// How it's used:
Pair<int, double> p1(1, 2.5);
Pair<std::string, int> p2("hello", 42);
```

### Template Instantiation and Specialization

**What is instantiation?** When you use a template with a specific type (e.g., `my_max(5, 10)`), the compiler generates a concrete, non-template version of the code for that type (e.g., `int my_max(int, int)`). This process is called **template instantiation**.

**Why specialize?** Sometimes, the generic algorithm is wrong for a specific type. The classic example is comparing C-style strings (`const char*`). The generic `my_max` would compare the pointer addresses, not the string content. To fix this, you can provide a **template specialization**—a custom, hand-written implementation for that one specific type.

```
#include <cstring>

// Generic template
template <typename T>
T my_max(T a, T b) { return (a > b) ? a : b; }

// Template specialization for const char*
```

```cpp
template <>
const char* my_max<const char*>(const char* a, const char* b) {
    // Provide a correct implementation using strcmp
    return (std::strcmp(a, b) > 0) ? a : b;
}
```

#### Theory: The Two-Phase Translation Model

**Why are template errors so long?** This is why. The compiler checks template code in two phases. 1. **Phase 1 (Definition Time):** When the template is first defined, the compiler checks for syntax errors that **do not** depend on the template parameters (e.g., missing semicolons, mismatched parentheses). 2. **Phase 2 (Instantiation Time):** When you use the template with a concrete type (e.g., `my_max<MyClass>(a, b)`), the compiler substitutes `MyClass` for `T` and compiles the code again. Now it checks for errors that **do** depend on the type (e.g., does `MyClass` support the `>` operator?). An error found here is deep inside the instantiation process, and the compiler often prints the entire chain of template instantiations that led to the error.

## 12.2 The STL Ecosystem: A Triumph of Generic Programming

The Standard Template Library (STL) is a masterclass in generic programming. Its design brilliantly decouples data structures from the algorithms that operate on them.

1. **Containers:** Class templates that own and manage data (e.g., `std::vector`, `std::list`, `std::map`).
2. **Algorithms:** Function templates that perform operations on data (e.g., `std::sort`, `std::find`, `std::for_each`).
3. **Iterators:** The "glue" that connects them. Iterators are objects that act like pointers, providing a uniform interface for algorithms to traverse containers without needing to know the container's internal details.

**Why is this design so powerful?** `std::sort` doesn't know what a `std::vector` is. It only knows how to work with a pair of iterators that meet certain requirements. Because `std::vector` provides these iterators, `std::sort` works on it. This means you can write a new container, provide the correct iterator interface, and all the standard algorithms will work with your new container for free. It is the ultimate expression of code reuse.

#### Insight: The Zero-Overhead Principle

This is a core philosophy of C++: **"You don't pay for what you don't use."** Templates are a key example. All the work of template instantiation and type resolution happens at **compile time**. The resulting machine code is just as fast as if you had written the specialized code by hand. The call to `my_max(5, 10)` compiles down to the exact same machine instructions as a specific `int max_int(int, int)` function. The powerful abstractions of templates exist only in your source code, not as runtime overhead.

---

## Projects for Chapter 12

### Project 1: The Generic `Pair` Class

- **Problem Statement:** Create a class template `Pair` that can hold two values of potentially different types (`template <typename T1, typename T2>`). It should have a constructor to initialize the two members, a `get_first()` method, and a `get_second()` method. In `main`, create an instance of `Pair<int, double>` and an instance of `Pair<std::string, bool>`. Print the contents of both pairs to demonstrate that your template works with different types.
- **Core Concepts to Apply:** Class templates, multiple template parameters.

### Project 2: The `my_max` Specialization

- **Problem Statement:** Write a generic function template `my_max(T a, T b)` that returns the greater of two values. Then, write a full template specialization for `const char*` that uses `std::strcmp` to correctly compare the content of C-style strings. In `main`, demonstrate that your function works correctly for `int`s, `double`s, and `const char*` literals.
- **Core Concepts to Apply:** Function templates, template specialization for pointer types.

### Project 3: The Generic Search Algorithm

- **Problem Statement:** Write your own generic search algorithm. It should be a function template `template <typename Iterator, typename T> bool contains(Iterator begin, Iterator end, const T& value)` that takes two iterators defining a range and a value to search for. The function should loop from `begin` to `end` and return `true` if the value is found, and `false` otherwise. In `main`, demonstrate that your single function works correctly on both a `std::vector<int>` and a `std::list<std::string>`.

- **Core Concepts to Apply:** Function templates, working with iterators, decoupling algorithms from containers.