

# Generic Programming with C++

Building Reusable and Type-Safe Code

# What is Generic Programming?

- Write code once, use with multiple types
- Focus on algorithms and data structures independent of specific types
- Achieve code reusability without sacrificing type safety or performance
- Core feature: **Templates**

# Function Templates

```
// Traditional approach - separate functions for each type
int max(int a, int b) { return (a > b) ? a : b; }
double max(double a, double b) { return (a > b) ? a : b; }

// Generic approach - one template for all types
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}

// Usage
int i = max(10, 20);           // T = int
double d = max(3.14, 2.71);    // T = double
```

# Class Templates

```
template <typename T>
class Stack {
private:
    std::vector<T> elements;

public:
    void push(const T& elem) {
        elements.push_back(elem);
    }

    T pop() {
        T elem = elements.back();
        elements.pop_back();
        return elem;
    }

    bool empty() const {
        return elements.empty();
    }
};
```

# Using Class Templates

```
// Stack of integers
Stack<int> intStack;
intStack.push(7);
intStack.push(42);
std::cout << intStack.pop(); // 42

// Stack of strings
Stack<std::string> stringStack;
stringStack.push("Hello");
stringStack.push("World");
std::cout << stringStack.pop(); // "World"

// Stack of custom types
Stack<MyClass> objectStack;
```

# Template Specialization

Provide specific implementations for certain types:

```
// Primary template
template <typename T>
class Container {
public:
    void process() {
        std::cout << "Generic processing\n";
    }
};

// Specialization for bool
template <>
class Container<bool> {
public:
    void process() {
        std::cout << "Special bool processing\n";
    }
};
```

# Concepts (C++20)

Define requirements for template parameters:

```
#include <concepts>

// Define a concept
template <typename T>
concept Numeric = std::is_arithmetic_v<T>;

// Use the concept
template <Numeric T>
T multiply(T a, T b) {
    return a * b;
}

// This works
auto result1 = multiply(5, 10);           // OK
auto result2 = multiply(3.5, 2.0);       // OK

// This fails at compile time
// auto result3 = multiply("hello", "world"); // Error
```

# Type Traits

Query and manipulate types at compile time:

```
#include <type_traits>

template <typename T>
void process(T value) {
    if constexpr (std::is_integral_v<T>) {
        std::cout << "Processing integer: " << value << "\n";
    } else if constexpr (std::is_floating_point_v<T>) {
        std::cout << "Processing float: " << value << "\n";
    } else {
        std::cout << "Processing other type\n";
    }
}

process(42);           // "Processing integer: 42"
process(3.14);         // "Processing float: 3.14"
process("text");       // "Processing other type"
```

# Variadic Templates

Accept variable number of arguments:

```
// Base case
void print() {
    std::cout << "\n";
}

// Recursive case
template <typename T, typename... Args>
void print(T first, Args... rest) {
    std::cout << first << " ";
    print(rest...); // Recursively call with remaining args
}

// Usage
print(1, 2.5, "hello", 'c', true);
// Output: 1 2.5 hello c 1
```

# SFINAE (Substitution Failure Is Not An Error)

Enable/disable templates based on conditions:

```
#include <type_traits>

// Only enabled for integral types
template <typename T>
typename std::enable_if<std::is_integral<T>::value, T>::type
square(T value) {
    return value * value;
}

// Only enabled for floating-point types
template <typename T>
typename std::enable_if<std::is_floating_point<T>::value, T>::type
square(T value) {
    return value * value * 1.0;
}

square(5);           // Calls integral version
square(2.14);        // Calls floating point version
```

# Real-World Example: Generic Algorithms

```
template <typename Iterator, typename Predicate>
Iterator find_if(Iterator first, Iterator last, Predicate pred) {
    while (first != last) {
        if (pred(*first)) {
            return first;
        }
        ++first;
    }
    return last;
}

// Usage with lambda
std::vector<int> vec = {1, 2, 3, 4, 5};
auto it = find_if(vec.begin(), vec.end(),
    [](int x) { return x > 3; });

if (it != vec.end()) {
    std::cout << "Found: " << *it << "\n"; // Found: 4
}
```

# Benefits of Generic Programming

- ✓ **Code Reusability** - Write once, use with many types
- ✓ **Type Safety** - Compile-time type checking
- ✓ **Performance** - No runtime overhead, inline optimization
- ✓ **Maintainability** - Single implementation to maintain
- ✓ **Flexibility** - Adapt to new types without modification

## Common Pitfalls

- ⚠ **Cryptic error messages** - Template errors can be complex
- ⚠ **Code bloat** - Each instantiation generates code
- ⚠ **Compilation time** - Templates increase compile time
- ⚠ **Header-only** - Templates must be in headers

**Solutions:** Use concepts (C++20), forward declarations, explicit instantiation, and precompiled headers

# Best Practices

1. **Use concepts** (C++20) to constrain templates
2. **Provide clear documentation** about template requirements
3. **Test with various types** including edge cases
4. **Consider explicit instantiation** for common types
5. **Use meaningful template parameter names**
6. **Leverage the standard library** (STL) as reference

# Standard Library Examples

The C++ Standard Library heavily uses generic programming:

```
std::vector<int>           // Dynamic array of integers
std::map<string, int>      // Key-value pairs
std::sort(vec.begin(), vec.end()) // Generic sorting
std::accumulate(v.begin(), v.end(), 0) // Generic accumulation
std::transform(in.begin(), in.end(), out.begin(), func) // Transform
```

All of these work with any compatible type!

# Resources

- **C++ Reference:** <https://en.cppreference.com/>
- **C++ Core Guidelines:** <https://isocpp.github.io/CppCoreGuidelines/>
- **Books:**
  - "C++ Templates: The Complete Guide" by Vandevorde & Josuttis
  - "Effective Modern C++" by Scott Meyers
- **Practice:** Implement your own generic containers and algorithms

# Summary

- **Templates** enable generic programming in C++
- Write **type-safe**, **reusable**, and **efficient** code
- Modern C++ (C++20) makes generics more accessible with **concepts**
- The **STL** is a prime example of generic programming
- Master templates to write professional C++ code

# Thank You!

Questions?