Generic Programming with C++

Building Reusable and Type-Safe Code

Author: [Your Name]

Date: November 10, 2025

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
void sortIntArray(int arr[], int n) {
    for (int i = 0; i < n-1; i++)
        for (int j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1])
                std::swap(arr[j], arr[j+1]);
}

void sortDoubleArray(double arr[], int n) {
    /* duplicate code */
}

void sortStringArray(
    std::string arr[], int n) {
    /* duplicate code */
}
```

Generic approach

```
// One template for all types
template <typename T>
void bubbleSort(T arr[], int n) {
    for (int i = 0; i < n-1; i++)
        for (int j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1])
                std::swap(arr[j], arr[j+1]);
}
```

- ✅ Works with all comparable types
- ✅ Single implementation
- ✅ Type-safe
- ✅ No code duplication

Function Templates - Usage

```cpp
// Works with any type that supports operator>
int intArr[] = {64, 34, 25, 12, 22};
bubbleSort(intArr, 5);  // T = int

double dblArr[] = {3.14, 2.71, 1.41, 1.73};
bubbleSort(dblArr, 4);  // T = double

std::string names[] = {"Zoe", "Alice", "Bob", "Charlie"};
bubbleSort(names, 4);   // T = std::string

// Even works with custom types (if they have operator>)
struct Person {
    std::string name;
    int age;
    bool operator>(const Person& other) const { return age > other.age; }
};

Person people[] = {{"Alice", 30}, {"Bob", 25}, {"Charlie", 35}};
bubbleSort(people, 3);  // Sorts by age!
```

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 Parameters: Types and Values

Templates can accept both **type** and **value** parameters:

```
// Type parameters (typename or class)
template <typename T>        // T is a type parameter
class Container { T data; };

template <class T>           // 'class' means the same as 'typename'
class Box { T item; };

// Non-type parameters (compile-time constants)
template <typename T, int N>  // N is a non-type parameter
class Array {
    T elements[N];            // Array size known at compile time
public:
    constexpr int size() const { return N; }
};

// Usage
Array<int, 5> arr1;      // T=int, N=5
Array<double, 10> arr2;  // T=double, N=10
```

Non-Type Template Parameters

Allowed non-type parameter types:

- Integral types: `int`, `char`, `bool`, `size_t`, `enum`
- Pointer/reference types
- `std::nullptr_t`
- C++20: Floating point, literal class types

Integer parameter

```
template <int Size>
class Buffer {
    char data[Size];
};
```

Multiple non-type parameters

```
template <typename T, size_t Rows, size_t Cols>
class Matrix {
    T data[Rows][Cols];
public:
    T& at(size_t r, size_t c) {
        return data[r][c];
    }
    constexpr size_t rows() const {
        return Rows;
    }
    constexpr size_t cols() const {
        return Cols;
    }
};

Matrix<double, 3, 3> identity;  // 3x3 matrix
```

Template Parameters - Advanced Examples

```cpp
// Pointer to function as template parameter
template <int (*Func)(int)>
class FunctionWrapper {
public:
    int call(int x) { return Func(x); }
};

int square(int x) { return x * x; }
FunctionWrapper<square> wrapper;
wrapper.call(5);  // 25

// C++17: auto for non-type parameters
template <auto Value>
class Constant {
public:
    static constexpr auto value = Value;
};

Constant<42> int_const;        // Value is int
Constant<3.14> double_const;   // Value is double
Constant<'A'> char_const;      // Value is char

// C++20: Class types as non-type parameters
struct Point { int x, y; };
template <Point P>
class PointHolder {
    static constexpr int x = P.x;
    static constexpr int y = P.y;
};
```

Template Parameter Defaults and Variadic

```cpp
// Default template parameters
template <typename T = int, int N = 10>
class Array {
    T data[N];
};

Array<> arr1;           // T=int, N=10 (uses defaults)
Array<double> arr2;     // T=double, N=10
Array<char, 20> arr3;   // T=char, N=20

// Variadic template parameters (parameter pack)
template <typename... Types>
class Tuple;  // Can accept any number of type parameters

Tuple<int, double, std::string> t1;  // 3 types
Tuple<int> t2;                        // 1 type
Tuple<> t3;                           // 0 types

// Mixed: regular + variadic
template <typename First, typename... Rest>
class TypeList {
    // First is guaranteed, Rest are optional
};

TypeList<int, double, char> list;  // First=int, Rest={double, char}
```

Recording Template Parameters

Storing and querying template parameters:

```
// Technique 1: Store as member type/value
template <typename T, int N>
class Array {
public:
    using value_type = T;              // Record type
    static constexpr int size = N;     // Record value
};

// Query the recorded parameters
Array<int, 5>::value_type x = 42;      // x is int
constexpr int sz = Array<int, 5>::size;  // sz = 5

// Technique 2: Template template parameters
template <template <typename> class Container>
class Adapter {
    Container<int> int_container;
    Container<double> double_container;
};

Adapter<std::vector> adapter;  // Uses std::vector internally
```

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";
    }
};
```

CRTP (Curiously Recurring Template Pattern)

A pattern where a class inherits from a template instantiation of itself:

```
// Base class template - takes derived class as parameter
template <typename Derived>
class Base {
public:
    void interface() {
        // Static polymorphism - no virtual functions!
        static_cast<Derived*>(this)->implementation();
    }

    void common_functionality() {
        std::cout << "Common code in base\n";
        static_cast<Derived*>(this)->implementation();
    }
};

// Derived class inherits from Base<Derived>
class Derived : public Base<Derived> {
public:
    void implementation() {
        std::cout << "Derived implementation\n";
    }
};
```

CRTP - Benefits and Use Cases

Why use CRTP?

- ✅ **Compile-time polymorphism**

  No virtual function overhead

- ✅ **Code reuse**

  Share implementation across derived classes

- ✅ **Type-safe**

  Errors caught at compile time

- ✅ **Zero runtime cost**

  Calls are inlined

Use case: Comparable mixin

```
template <typename T>
class Comparable {
public:
    bool operator!=(const T& other) const {
        return !(static_cast<const T&>(*this)
                 == other);
    }
    bool operator>(const T& other) const {
        return other <
               static_cast<const T&>(*this);
    }
    bool operator<=(const T& other) const {
        return !(static_cast<const T&>(*this)
                 > other);
    }
    bool operator>=(const T& other) const {
        return !(static_cast<const T&>(*this)
                 < other);
    }
};

class MyInt : public Comparable<MyInt> {
    int value;
public:
    MyInt(int v) : value(v) {}
    bool operator==(const MyInt& o) const {
        return value == o.value;
    }
    bool operator<(const MyInt& o) const {
        return value < o.value;
```

CRTP - Advanced Example

Static Polymorphism vs Virtual Functions:

Traditional (runtime cost)

```
class Animal {
public:
    virtual void speak() = 0;
    // Runtime dispatch via vtable
};

class Dog : public Animal {
public:
    void speak() override {
        std::cout << "Woof!\n";
    }
};

class Cat : public Animal {
public:
    void speak() override {
        std::cout << "Meow!\n";
    }
};
```

❌ Virtual table overhead

CRTP (zero overhead)

```
template <typename Derived>
class AnimalCRTP {
public:
    void speak() {
        static_cast<Derived*>(this)
            ->speak_impl();
    }
    void describe() {
        std::cout << "This animal says: ";
        speak();  // Compile-time!
    }
};

class Dog : public AnimalCRTP<Dog> {
public:
    void speak_impl() {
        std::cout << "Woof!\n";
    }
};

class Cat : public AnimalCRTP<Cat> {
public:
    void speak_impl() {
```

CRTP - Practical Example: Object Counter

```cpp
// Count instances of any class using CRTP
template <typename T>
class ObjectCounter {
private:
    static inline int count = 0;  // C++17 inline static

protected:
    ObjectCounter() { ++count; }
    ObjectCounter(const ObjectCounter&) { ++count; }
    ~ObjectCounter() { --count; }

public:
    static int get_count() { return count; }
};

// Any class can track its instances
class Widget : public ObjectCounter<Widget> {
    // Widget implementation
};

class Gadget : public ObjectCounter<Gadget> {
    // Gadget implementation
};

// Usage
Widget w1, w2, w3;
Gadget g1, g2;
std::cout << Widget::get_count();  // 3
std::cout << Gadget::get_count();  // 2
// Each type has its own counter!
```

Concepts (C++20)

Define requirements for template parameters:

```cpp
#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:

```cpp
#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"
```

constexpr if (C++17)

Compile-time conditional branches - discarded branches are not compiled:

```
// Before C++17: Required function overloading or SFINAE
template <typename T>
auto get_value_old(T t) -> std::enable_if_t<std::is_pointer_v<T>,
                                            std::remove_pointer_t<T>> {
    return *t;  // Dereference pointer
}

template <typename T>
auto get_value_old(T t) -> std::enable_if_t<!std::is_pointer_v<T>, T> {
    return t;   // Return value directly
}

// With C++17 constexpr if: Much simpler!
template <typename T>
auto get_value(T t) {
    if constexpr (std::is_pointer_v<T>) {
        return *t;  // Only compiled if T is a pointer
    } else {
        return t;   // Only compiled if T is not a pointer
    }
}
```

constexpr if - Advanced Examples

```cpp
// Example 1: Different processing based on container type
template <typename Container>
auto process_container(const Container& c) {
    if constexpr (std::is_same_v<Container, std::string>) {
        return c.length();  // String-specific operation
    } else if constexpr (requires { c.size(); }) {
        return c.size();    // Generic container
    } else {
        return std::distance(c.begin(), c.end());  // Iterator-based
    }
}

// Example 2: Optimized serialization
template <typename T>
void serialize(const T& value, std::ostream& os) {
    if constexpr (std::is_arithmetic_v<T>) {
        os.write(reinterpret_cast<const char*>(&value), sizeof(T));
    } else if constexpr (requires { value.serialize(os); }) {
        value.serialize(os);  // Custom serialization
    } else {
        os << value;  // Fallback to stream operator
    }
}
```

constexpr if vs Runtime if

```cpp
template <typename T>
void compare(T value) {
    // Runtime if - both branches must be valid C++
    if (std::is_integral_v<T>) {
        // This would cause compile error if T is not integral!
        // int x = value + 1;  // Error if T is std::string
    }

    // constexpr if - discarded branch is not compiled
    if constexpr (std::is_integral_v<T>) {
        int x = value + 1;     // OK: only compiled for integral types
        std::cout << "Integer: " << x << "\n";
    } else if constexpr (std::is_floating_point_v<T>) {
        double x = value * 2.0;  // Only compiled for floating-point
        std::cout << "Float: " << x << "\n";
    } else {
        std::cout << "Other: " << value << "\n";
    }
}

compare(42);        // Only integer branch compiled
compare(3.14);      // Only float branch compiled
compare("hello");   // Only other branch compiled
```

Compile-Time Computation

Move computations from runtime to compile time for zero-cost abstractions:

constexpr variables

```
constexpr int factorial(int n) {
    return n <= 1 ? 1 : n * factorial(n-1);
}

constexpr int val = factorial(5);  // 120
// Computed at compile time!
// No runtime overhead

constexpr double pi = 3.14159265359;
constexpr double area(double r) {
    return pi * r * r;
}
constexpr double circle_area = area(10.0);
```

Template metaprogramming

```
// Fibonacci at compile time
template <int N>
struct Fibonacci {
    static constexpr int value =
        Fibonacci<N-1>::value +
        Fibonacci<N-2>::value;
};

template <>
struct Fibonacci<0> {
    static constexpr int value = 0;
};

template <>
struct Fibonacci<1> {
    static constexpr int value = 1;
};

// Computed at compile time
constexpr int fib10 = Fibonacci<10>::value;
// fib10 = 55, no runtime cost!
```

Compile-Time Computation - Advanced

constexpr functions (C++14+)

```cpp
// Complex compile-time computation
constexpr int pow(int base, int exp) {
    int result = 1;
    for (int i = 0; i < exp; ++i) {
        result *= base;
    }
    return result;
}

constexpr int lookup_table[5] = {
    pow(2, 0),  // 1
    pow(2, 1),  // 2
    pow(2, 2),  // 4
    pow(2, 3),  // 8
    pow(2, 4)   // 16
};
// All computed at compile time!

// Also works at runtime if needed
int runtime_val = pow(2, x);
```

consteval (C++20)

```cpp
// MUST be evaluated at compile time
consteval int compile_time_only(int n) {
    return n * n;
}

constexpr int a = compile_time_only(5);
// OK: compile time

// int b = compile_time_only(x);
// Error: x is runtime variable

// Type-safe compile-time strings
consteval auto get_version() {
    return "v1.2.3";
}

// Enforce compile-time evaluation
template <auto Value>
struct CompileTimeValue {
    static constexpr auto value = Value;
};

CompileTimeValue<compile_time_only(10)> ct;
```

Compile-Time vs Runtime

```cpp
// Runtime computation (slow)
int runtime_factorial(int n) {
    int result = 1;
    for (int i = 2; i <= n; ++i)
        result *= i;
    return result;
}

int main() {
    int x = runtime_factorial(10);  // Computed every time program runs
}

// Compile-time computation (zero cost)
constexpr int compile_time_factorial(int n) {
    return n <= 1 ? 1 : n * compile_time_factorial(n-1);
}

int main() {
    constexpr int x = compile_time_factorial(10);  // Computed once at compile time
    // x is a constant in the binary, like writing: const int x = 3628800;
}
```

Benefits:

- ✅ Zero runtime overhead - results baked into binary

Custom Type Traits

Define your own type traits to query custom properties:

```
// Trait to check if a type has a serialize() method
template <typename T, typename = void>
struct has_serialize : std::false_type {};

template <typename T>
struct has_serialize<T, std::void_t<decltype(std::declval<T>().serialize())>>
    : std::true_type {};

// Helper variable template (C++17)
template <typename T>
inline constexpr bool has_serialize_v = has_serialize<T>::value;

// Usage
struct Serializable { void serialize() {} };
struct NotSerializable { };

static_assert(has_serialize_v<Serializable>);      // OK
static_assert(!has_serialize_v<NotSerializable>);  // OK
```

Custom Type Traits - Advanced Example

```cpp
// Trait to detect if a type is a container
template <typename T, typename = void>
struct is_container : std::false_type {};

template <typename T>
struct is_container<T, std::void_t<
    typename T::value_type,
    typename T::iterator,
    decltype(std::declval<T>().begin()),
    decltype(std::declval<T>().end())
>> : std::true_type {};

template <typename T>
inline constexpr bool is_container_v = is_container<T>::value;

// Use in template functions
template <typename T>
auto print(const T& value) {
    if constexpr (is_container_v<T>) {
        for (const auto& elem : value) std::cout << elem << " ";
    } else {
        std::cout << value;
    }
}
```

Variadic Templates

Accept variable number of arguments:

```cpp
// 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
```

std::index_sequence

Generate compile-time integer sequences for unpacking tuples/arrays:

```
#include <utility>
#include <tuple>

// Helper to apply function to tuple elements
template <typename Func, typename Tuple, std::size_t... I>
auto apply_impl(Func&& f, Tuple&& t, std::index_sequence<I...>) {
    return f(std::get<I>(std::forward<Tuple>(t))...);
}

template <typename Func, typename Tuple>
auto apply(Func&& f, Tuple&& t) {
    constexpr auto size = std::tuple_size_v<std::decay_t<Tuple>>;
    return apply_impl(std::forward<Func>(f),
                      std::forward<Tuple>(t),
                      std::make_index_sequence<size>{});
}
```

std::index_sequence - Usage

```cpp
// Example: Sum all elements in a tuple
auto sum = [](auto... args) { return (args + ...); };

std::tuple<int, double, int> values{10, 3.14, 7};
auto result = apply(sum, values);  // result = 20.14

// Example: Print tuple elements
auto print_all = [](auto... args) {
    ((std::cout << args << " "), ...);
};

std::tuple<std::string, int, char> data{"Hello", 42, '!'};
apply(print_all, data);  // Output: Hello 42 !

// Example: Convert array to tuple
template <typename Array, std::size_t... I>
auto array_to_tuple_impl(const Array& arr, std::index_sequence<I...>) {
    return std::make_tuple(arr[I]...);
}

template <typename T, std::size_t N>
auto array_to_tuple(const std::array<T, N>& arr) {
    return array_to_tuple_impl(arr, std::make_index_sequence<N>{});
}
```

Fold Expressions (C++17)

Simplify variadic template operations with fold expressions:

```cpp
// Four types of fold expressions:

// 1. Unary right fold: (args op ...)
template <typename... Args>
auto sum(Args... args) {
    return (args + ...);  // Expands to: arg1 + (arg2 + (arg3 + ...))
}

// 2. Unary left fold: (... op args)
template <typename... Args>
auto sum_left(Args... args) {
    return (... + args);  // Expands to: ((... + arg1) + arg2) + arg3
}

// 3. Binary right fold: (args op ... op init)
template <typename... Args>
auto sum_with_init(Args... args) {
    return (args + ... + 0);  // Starts with 0
}

// 4. Binary left fold: (init op ... op args)
template <typename... Args>
auto sum_left_init(Args... args) {
    return (0 + ... + args);  // Starts with 0
}
```

Fold Expression Examples

```cpp
// Addition
template <typename... Args>
auto sum(Args... args) { return (args + ...); }
sum(1, 2, 3, 4, 5);  // 15

// Multiplication
template <typename... Args>
auto product(Args... args) { return (args * ...); }
product(2, 3, 4);  // 24

// Logical AND
template <typename... Args>
bool all_true(Args... args) { return (args && ...); }
all_true(true, true, false);  // false

// Logical OR
template <typename... Args>
bool any_true(Args... args) { return (args || ...); }
any_true(false, false, true);  // true

// Comma operator - execute all, return last
template <typename... Args>
void print_all(Args... args) {
    (std::cout << ... << args) << '\n';  // Prints all concatenated
}
print_all("Hello", " ", "World", "!");  // Hello World!
```

Fold Expression Shortcuts & Tricks

```cpp
// Print with spaces using comma fold
template <typename... Args>
void print_with_spaces(Args... args) {
    ((std::cout << args << ' '), ...);  // Comma operator trick
}
print_with_spaces(1, 2, 3);  // "1 2 3 "

// Push multiple elements into vector
template <typename T, typename... Args>
void push_all(std::vector<T>& vec, Args&&... args) {
    (vec.push_back(std::forward<Args>(args)), ...);
}

std::vector<int> v;
push_all(v, 1, 2, 3, 4, 5);  // v = {1, 2, 3, 4, 5}

// Check if value is in parameter pack
template <typename T, typename... Args>
bool is_in(T value, Args... args) {
    return ((value == args) || ...);
}
is_in(3, 1, 2, 3, 4);  // true

// Count matching elements
template <typename T, typename... Args>
int count_matches(T value, Args... args) {
    return ((value == args) + ...);
}
count_matches(2, 1, 2, 3, 2, 4, 2);  // 3
```

SFINAE (Substitution Failure Is Not An Error)

Enable/disable templates based on conditions:

```cpp
#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(3.14);   // 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

Template Compilation Performance

Understanding how templates affect build times:

```
// Problem: Each instantiation generates new code
template <typename T>
class Vector {
    // Complex implementation with many methods
    void push_back(const T&);
    void insert(iterator, const T&);
    void resize(size_t);
    // ... 50+ methods
};

// Each type creates a full copy of ALL methods
Vector<int> v1;        // Generates ~50 int-specific methods
Vector<double> v2;     // Generates ~50 double-specific methods
Vector<std::string> v3; // Generates ~50 string-specific methods
// Result: 150+ functions in object code!
```

Compilation Performance - Impact

Why templates slow down compilation:

1. **Template Instantiation** - Compiler generates code for each unique type
2. **Header Inclusion** - Templates must be in headers, parsed repeatedly
3. **Recursive Instantiation** - Complex templates can instantiate deeply
4. **Error Cascading** - One error can trigger hundreds of messages

Example: Compilation time comparison

```
// Non-template: Fast compilation
class IntVector { /* implementation */ };  // Compiled once

// Template: Slower compilation
template <typename T>
class Vector { /* implementation */ };     // Compiled N times

// Usage in 100 files with 10 types each = 1000 instantiations!
```

Optimizing Template Compilation

Technique 1: Explicit Instantiation

```
// header.h
template <typename T>
class MyClass { /* implementation */ };

// Prevent implicit instantiation
extern template class MyClass<int>;
extern template class MyClass<double>;

// implementation.cpp - Compile only once
template class MyClass<int>;
template class MyClass<double>;

// Usage: No recompilation of template body
MyClass<int> obj;  // Uses pre-compiled version
```

Optimizing Template Compilation (cont.)

Technique 2: Type Erasure / Non-templated Base

```
// Non-templated base class contains most implementation
class VectorBase {
protected:
    void* data_;
    size_t size_;
    void resize_impl(size_t new_size, size_t elem_size);
    void push_impl(const void* elem, size_t elem_size);
};

// Thin template wrapper - minimal code generation
template <typename T>
class Vector : private VectorBase {
public:
    void push_back(const T& value) {
        push_impl(&value, sizeof(T));  // Delegates to non-templated code
    }
    T& operator[](size_t i) { return static_cast<T*>(data_)[i]; }
};
// Result: Most code compiled once, not per-type!
```

Optimizing Template Compilation (cont.)

Technique 3: Precompiled Headers (PCH)

```
// stdafx.h or pch.h - Include heavy templates once
#include <vector>
#include <string>
#include <map>
#include <memory>
#include <algorithm>
// Compiled once, reused across all translation units

// Technique 4: Forward Declarations
template <typename T> class MyClass;  // Declare only

// Use pointers/references without full definition
void process(MyClass<int>* ptr);     // OK
void process(MyClass<int>& ref);     // OK
// void process(MyClass<int> val);   // ERROR: needs full definition
```

Technique 5: C++20 Modules (future)

```
// Replace headers with modules - faster compilation
import std.vector;    // No text preprocessing
import my_templates;  // Compiled module, not header
```

Compilation Performance - Real Numbers

Typical compilation time impact:

Code Type	Files	Build Time	Code Size
Non-template	100	10s	500 KB
Basic templates	100	25s	1.2 MB
Heavy STL usage	100	60s	3.5 MB
Complex metaprogramming	100	180s+	8+ MB

Mitigation strategies:

- Use explicit instantiation for common types (2-3x faster)
- Enable precompiled headers (3-5x faster)
- Minimize template depth and recursion

Common Pitfalls

⚠️ **Cryptic error messages** - Template errors can be complex
⚠️ **Code bloat** - Each instantiation generates code
⚠️ **Compilation time** - Templates increase compile time
⚠️ **Debugging difficulty** - Template code harder to step through

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

Pitfall 1: Template Error Messages

```
template <typename T>
void process(T value) {
    value.nonexistent_method();  // Typo!
}

std::vector<int> vec;
process(vec);

// Error: 50+ lines of template instantiation stack trace
// error: 'class std::vector<int>' has no member named 'nonexistent_method'
// in instantiation of 'void process(T) [with T = std::vector<int>]'
// ... dozens more lines ...
```

Solution: Use concepts for clearer errors

```
template <typename T>
concept HasMethod = requires(T t) { t.nonexistent_method(); };

template <HasMethod T>
void process(T value) {
    value.nonexistent_method();
}

process(vec);  // Clear error: constraint not satisfied!
```

Pitfall 2: Accidental Template Instantiation

```
// header.h
template <typename T>
class HeavyClass {
    // 1000+ lines of complex code
};

// In 50 different .cpp files:
#include "header.h"
HeavyClass<int> obj;  // Each file re-compiles the template!
```

Solution: Explicit instantiation

```
// header.h
template <typename T>
class HeavyClass { /* ... */ };

extern template class HeavyClass<int>;  // Declare only

// implementation.cpp
template class HeavyClass<int>;  // Define once

// Now all files use pre-compiled version
```

Pitfall 3: Template Argument Deduction Issues

```
template <typename T>
void print(T a, T b) {  // Both parameters must be same type
    std::cout << a << " " << b << "\n";
}

print(1, 2.5);  // Error! T = int or double?

// Problem: Cannot deduce T
template <typename T>
T max(T a, T b) { return a > b ? a : b; }
auto result = max(42, 3.14);  // Error: conflicting types
```

Solutions:

```
// Solution 1: Multiple template parameters
template <typename T, typename U>
void print(T a, U b) { std::cout << a << " " << b << "\n"; }

// Solution 2: Explicit type
auto result = max<double>(42, 3.14);  // OK

// Solution 3: Common type
template <typename T, typename U>
auto max(T a, U b) { return (a > b) ? a : b; }  // C++14
```

Pitfall 4: typename vs class Keyword Confusion

```
template <typename T>
class Container {
    // Wrong: compiler thinks T::iterator is a value
    T::iterator it;  // Error!

    // Correct: tell compiler it's a type
    typename T::iterator it;  // OK

    // Also wrong in dependent contexts
    std::vector<T>::size_type count;  // Error!

    // Correct
    typename std::vector<T>::size_type count;  // OK
};
```

Rule: Use `typename` when referring to dependent type names

Pitfall 5: Template Template Parameter Confusion

```
// Wrong: Can't pass std::vector directly
template <typename Container>
class Adapter {
    Container<int> data;  // Error: Container is a type, not template!
};

Adapter<std::vector<int>> a;  // Wrong usage

// Correct: Use template template parameter
template <template <typename> class Container>
class Adapter {
    Container<int> data;  // OK: Container is a template
};

// But std::vector has multiple template parameters!
Adapter<std::vector> a;  // Error: std::vector needs allocator too

// Full correct version
template <template <typename, typename...> class Container>
class Adapter {
    Container<int> data;
};

Adapter<std::vector> a;  // OK
```

Pitfall 6: Two-Phase Lookup

```
void helper() { std::cout << "Non-template helper\n"; }

template <typename T>
void process() {
    helper();  // Which helper? Looked up at definition time!
}

void helper() { std::cout << "Later helper\n"; }  // Too late!

process<int>();  // Calls first helper, not this one
```

Solution: Understand name lookup phases

- Non-dependent names: looked up at template definition
- Dependent names: looked up at template instantiation

```
template <typename T>
void process(T value) {
    value.method();  // Dependent: looked up at instantiation
    helper();         // Non-dependent: looked up at definition
}
```

Pitfall 7: Reference Collapsing Surprise

```cpp
template <typename T>
void func(T&& param);  // Universal/forwarding reference

int x = 42;
func(x);      // T = int&, param = int& && -> int&  (reference collapse)
func(42);     // T = int,  param = int&&

// Can lead to unexpected behavior
template <typename T>
void wrapper(T&& arg) {
    process(arg);  // Always passes lvalue! Lost rvalue-ness
}

// Correct: Use std::forward
template <typename T>
void wrapper(T&& arg) {
    process(std::forward<T>(arg));  // Preserves value category
}
```

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 Vandevoorde & 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?