Advanced C++

Hidden Features and Expert Techniques

Istvan Gellai

Contents

Part I: Templates

Chapter 1: Template Metaprogramming

Template metaprogramming (TMP) is a powerful and sophisticated feature of C++ that allows developers to perform computations at compile time. This technique leverages the template system to create programs that can generate other programs, leading to highly optimized and efficient code. By understanding and mastering template metaprogramming, C++ developers can write code that is not only more reusable and maintainable but also significantly faster.

In this chapter, we will explore the fundamentals of template metaprogramming, starting with the basic concepts of templates in C++. We will then delve into the more advanced aspects, such as template specializations, SFINAE (Substitution Failure Is Not An Error), and constexpr functions. We'll also discuss the practical applications of TMP, including type traits, compile-time algorithms, and policy-based design.

Template metaprogramming can be daunting due to its complexity and the intricate syntax involved. However, with a clear understanding of the underlying principles and guided examples, you will be able to harness the full potential of TMP to solve complex programming problems with elegance and efficiency.

Key topics covered in this chapter include:

1. **Introduction to Templates**: Understanding the basics of function templates, class templates, and template parameters.
2. **Recursive Templates**: Implementing recursive algorithms at compile time.
3. **Template Specialization**: Differentiating between full and partial specializations to handle specific cases.
4. **SFINAE**: Utilizing SFINAE to create robust and flexible templates.
5. **Type Traits and Compile-Time Computations**: Leveraging standard and custom type traits for meta-programming.
6. **Advanced Template Techniques**: Exploring variadic templates and template aliases.
7. **Practical Applications**: Applying TMP in real-world scenarios, such as optimizing performance and reducing code duplication.

By the end of this chapter, you will have a solid foundation in template metaprogramming and be prepared to apply these techniques to create high-performance C++ applications.

1.1 Recursive Templates

Understanding Recursive Templates Recursive templates are a cornerstone of template metaprogramming in C++. They enable the definition of complex compile-time algorithms by breaking down problems into simpler subproblems, akin to traditional recursion in runtime programming. In essence, a recursive template calls itself with modified parameters until a base case is reached. This technique allows for powerful compile-time computations and optimizations that would be impossible with runtime recursion alone.

Basics of Recursive Templates To understand recursive templates, let's start with a simple example: calculating the factorial of a number at compile time.

```cpp
#include <iostream>

// Base case: factorial of 0 is 1
template<int N>
struct Factorial {
    static constexpr int value = N * Factorial<N - 1>::value;
};

// Specialization for base case
template<>
struct Factorial<0> {
    static constexpr int value = 1;
};

int main() {
    constexpr int result = Factorial<5>::value;
    std::cout << "Factorial of 5 is " << result << std::endl; // Output: 120
    return 0;
}
```

In this example, the `Factorial` template recursively calculates the factorial of a number. The primary template handles the general case, while the specialization for `Factorial<0>` provides the base case, terminating the recursion.

Practical Example: Fibonacci Sequence Another classic example of recursion is the Fibonacci sequence. Let's see how we can compute Fibonacci numbers using recursive templates.

```
#include <iostream>

// General case
template<int N>
struct Fibonacci {
    static constexpr int value = Fibonacci<N - 1>::value + Fibonacci<N - 2>::value;
};

// Specializations for base cases
template<>
struct Fibonacci<0> {
    static constexpr int value = 0;
};

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

int main() {
    constexpr int result = Fibonacci<10>::value;
    std::cout << "Fibonacci of 10 is " << result << std::endl; // Output: 55
    return 0;
}
```

In this example, `Fibonacci<N>` recursively calculates the Fibonacci number by summing the values of the two preceding numbers. The specializations for `Fibonacci<0>` and `Fibonacci<1>` provide the base cases.

Compile-Time Computation and Optimization One of the significant advantages of recursive templates is that computations are performed at compile time, leading to potential optimizations. For instance, consider the power of a number:

```
#include <iostream>

// General case
template<int Base, int Exponent>
struct Power {
    static constexpr int value = Base * Power<Base, Exponent - 1>::value;
};

// Specialization for base case
template<int Base>
struct Power<Base, 0> {
    static constexpr int value = 1;
};

int main() {
    constexpr int result = Power<2, 8>::value;
    std::cout << "2 to the power of 8 is " << result << std::endl; // Output: 256
    return 0;
}
```

The Power template calculates the power of a number recursively. The primary template handles the general case, while the specialization for `Power<Base, 0>` provides the base case.

Optimizing Recursive Templates with Template Specialization Template specialization can be used to optimize recursive templates further. For instance, we can improve the power calculation by reducing the number of multiplications through exponentiation by squaring:

```
#include <iostream>

// General case
template<int Base, int Exponent, bool IsEven = (Exponent % 2 == 0)>
struct Power {
    static constexpr int value = Base * Power<Base, Exponent - 1>::value;
};

// Specialization for even exponents
template<int Base, int Exponent>
struct Power<Base, Exponent, true> {
    static constexpr int value = Power<Base * Base, Exponent / 2>::value;
};

// Specialization for base case
template<int Base>
struct Power<Base, 0, true> {
    static constexpr int value = 1;
};

int main() {
    constexpr int result = Power<2, 8>::value;
    std::cout << "2 to the power of 8 is " << result << std::endl; // Output: 256
    return 0;
}
```

In this optimized version, the `Power` template has an additional template parameter `IsEven` that determines if the exponent is even. The specialization for even exponents reduces the exponent by squaring the base and halving the exponent, significantly improving the efficiency.

Complex Example: Compile-Time Prime Check Let's consider a more complex example: checking if a number is prime at compile time. This involves recursive templates to divide the number by potential factors.

```
#include <iostream>

// Primary template for checking divisibility
template<int N, int Divisor>
struct IsPrimeHelper {
    static constexpr bool value = (N % Divisor != 0) && IsPrimeHelper<N, Divisor -
    ↪  1>::value;
};

// Specialization for base case when divisor is 1
template<int N>
struct IsPrimeHelper<N, 1> {
    static constexpr bool value = true;
};

// Primary template for checking primality
template<int N>
```

```cpp
struct IsPrime {
    static constexpr bool value = IsPrimeHelper<N, N / 2>::value;
};

// Special cases for 0 and 1
template<>
struct IsPrime<0> {
    static constexpr bool value = false;
};

template<>
struct IsPrime<1> {
    static constexpr bool value = false;
};

int main() {
    constexpr bool result = IsPrime<17>::value;
    std::cout << "Is 17 prime? " << (result ? "Yes" : "No") << std::endl; // Output: Yes
    return 0;
}
```

In this example, `IsPrimeHelper` recursively checks if `N` is divisible by any number from `N/2` down to `1`. The `IsPrime` template uses `IsPrimeHelper` to determine the primality of `N`. Specializations handle the base cases for `0` and `1`.

Conclusion Recursive templates are a powerful feature in C++ template metaprogramming, allowing for complex compile-time computations and optimizations. By understanding and utilizing recursive templates, developers can create highly efficient and maintainable code. The examples provided illustrate the fundamental concepts and practical applications of recursive templates, showcasing their potential to solve intricate programming problems elegantly. As you continue exploring template metaprogramming, recursive templates will become an invaluable tool in your C++ toolkit.

1.2. SFINAE (Substitution Failure Is Not An Error)

Understanding SFINAE Substitution Failure Is Not An Error (SFINAE) is a fundamental concept in C++ template metaprogramming that allows for elegant handling of template instantiation failures. When a template is instantiated with certain types, if the resulting instantiation leads to a substitution failure, the compiler does not treat this as an error. Instead, it simply ignores that template specialization and continues to look for other viable candidates. This feature is instrumental in creating highly flexible and adaptable code, as it enables conditional compilation and overloading based on the properties of types.

Basics of SFINAE At its core, SFINAE allows developers to write templates that can gracefully handle different types without causing compilation errors. This is particularly useful for creating generic libraries and functions that can operate on a wide range of types.

Consider the following simple example where we use SFINAE to determine if a type is integral:

```cpp
#include <iostream>
#include <type_traits>

// Primary template handles non-integral types
template<typename T, typename = void>
struct IsIntegral : std::false_type {};

// Specialization for integral types
template<typename T>
struct IsIntegral<T, std::enable_if_t<std::is_integral_v<T>>> : std::true_type {};

int main() {
    std::cout << "Is int integral? " << IsIntegral<int>::value << std::endl; // Output: 1
    ↪  (true)
    std::cout << "Is float integral? " << IsIntegral<float>::value << std::endl; // Output:
    ↪  0 (false)
    return 0;
}
```

In this example, the primary template for `IsIntegral` inherits from `std::false_type`, indicating that the type is not integral. The specialization uses `std::enable_if_t` to check if the type is integral, and if so, it inherits from `std::true_type`.

Using SFINAE for Function Overloading SFINAE is extremely useful for function overloading based on type traits. For instance, we can create overloaded functions that are enabled only if certain conditions are met.

```
#include <iostream>
#include <type_traits>

// Function enabled only for integral types
template<typename T>
std::enable_if_t<std::is_integral_v<T>, void> process(T value) {
    std::cout << value << " is an integral type." << std::endl;
}

// Function enabled only for floating-point types
template<typename T>
std::enable_if_t<std::is_floating_point_v<T>, void> process(T value) {
    std::cout << value << " is a floating-point type." << std::endl;
}

int main() {
    process(42);       // Output: 42 is an integral type.
    process(3.14);     // Output: 3.14 is a floating-point type.
    // process("text"); // Error: no matching function to call 'process'
    return 0;
}
```

In this example, we have two `process` functions: one for integral types and one for floating-point types. Each function is enabled using `std::enable_if_t` in conjunction with `std::is_integral_v` and `std::is_floating_point_v`, respectively.

Advanced SFINAE Techniques SFINAE can also be used in more complex scenarios to enable or disable functions based on multiple conditions. This is particularly useful when working with custom type traits or when combining multiple standard type traits.

Consider the following example where we enable a function only if the type is both integral and signed:

```
#include <iostream>
#include <type_traits>

// Function enabled only for signed integral types
template<typename T>
std::enable_if_t<std::is_integral_v<T> && std::is_signed_v<T>, void> process(T value) {
    std::cout << value << " is a signed integral type." << std::endl;
}

int main() {
    process(-42);       // Output: -42 is a signed integral type.
    // process(42u);    // Error: no matching function to call 'process'
    // process(3.14);   // Error: no matching function to call 'process'
    return 0;
}
```

In this example, the `process` function is enabled only if the type `T` is both integral and signed. The combination of `std::is_integral_v` and `std::is_signed_v` ensures that only signed integral types are allowed.

SFINAE and Member Functions SFINAE can also be applied to member functions, enabling or disabling them based on the properties of the class or its members. This is particularly useful in template classes that need to provide different functionality based on their template parameters.

Consider a template class that provides different `print` functions depending on whether the type has a `print` member function:

```cpp
#include <iostream>
#include <type_traits>

// Helper trait to detect the presence of a 'print' member function
template<typename T>
class HasPrint {
private:
    template<typename U>
    static auto test(int) -> decltype(std::declval<U>().print(), std::true_type{});

    template<typename>
    static std::false_type test(...);

public:
    static constexpr bool value = decltype(test<T>(0))::value;
};

// Class template
template<typename T>
class Printer {
public:
    // Enabled if T has a 'print' member function
    template<typename U = T>
    std::enable_if_t<HasPrint<U>::value, void> print() {
        static_cast<U*>(this)->print();
    }

    // Enabled if T does not have a 'print' member function
    template<typename U = T>
    std::enable_if_t<!HasPrint<U>::value, void> print() {
        std::cout << "No print member function." << std::endl;
    }
};

class WithPrint {
public:
    void print() {
        std::cout << "WithPrint::print()" << std::endl;
    }
};

class WithoutPrint {};

int main() {
    Printer<WithPrint> p1;
    p1.print(); // Output: WithPrint::print()

    Printer<WithoutPrint> p2;
    p2.print(); // Output: No print member function.
    return 0;
}
```

In this example, the `Printer` class template provides different implementations of the `print` member function depending on whether the type `T` has a `print` member function. The `HasPrint` trait detects the presence of a `print` member function using SFINAE.

Combining SFINAE with Concepts (C++20) With the introduction of concepts in C++20, SFINAE can be further streamlined and made more readable. Concepts allow for more expressive and concise constraints on template parameters.

Here's an example that combines SFINAE with concepts:

```
#include <iostream>
#include <concepts>

template<typename T>
concept Integral = std::is_integral_v<T>;

template<typename T>
concept FloatingPoint = std::is_floating_point_v<T>;

// Function enabled only for integral types
void process(Integral auto value) {
    std::cout << value << " is an integral type." << std::endl;
}

// Function enabled only for floating-point types
void process(FloatingPoint auto value) {
    std::cout << value << " is a floating-point type." << std::endl;
}

int main() {
    process(42);        // Output: 42 is an integral type.
    process(3.14);      // Output: 3.14 is a floating-point type.
    // process("text"); // Error: no matching function to call 'process'
    return 0;
}
```

In this example, the `process` function is overloaded using concepts to constrain the types to integral and floating-point types. This approach is more intuitive and easier to read compared to traditional SFINAE techniques.

Conclusion SFINAE is a powerful feature in C++ template metaprogramming, enabling developers to write highly flexible and robust code. By allowing template instantiations to fail without causing compilation errors, SFINAE facilitates conditional compilation and overloading based on type traits. From basic examples like determining if a type is integral to advanced use cases involving custom type traits and concepts, SFINAE proves to be an invaluable tool for C++ developers. As you continue to explore template metaprogramming, mastering SFINAE will significantly enhance your ability to write versatile and efficient C++ code.

1.3. Variadic Templates

Understanding Variadic Templates Variadic templates are a powerful feature introduced in C++11 that allow functions and classes to accept an arbitrary number of template parameters. This capability is extremely useful for creating generic and flexible code, as it eliminates the need to specify a fixed number of arguments. Variadic templates enable the development of functions and classes that can handle varying numbers of arguments in a type-safe manner, leading to more reusable and maintainable code.

Basics of Variadic Templates At the heart of variadic templates are parameter packs, which can hold zero or more template arguments. These packs can be expanded using recursive template instantiation or with specific language constructs provided by C++11 and later versions.

Let's start with a simple example that demonstrates the use of variadic templates to create a function that prints all its arguments:

```cpp
#include <iostream>

// Base case: no arguments to print
void print() {
    std::cout << "No more arguments." << std::endl;
}

// Recursive case: print the first argument and recurse
template<typename T, typename... Args>
void print(T first, Args... args) {
    std::cout << first << std::endl;
    print(args...); // Recursively call print with the remaining arguments
}

int main() {
    print(1, 2.5, "Hello", 'A');
    // Output:
    // 1
    // 2.5
    // Hello
    // A
    // No more arguments.
    return 0;
}
```

In this example, the `print` function uses variadic templates to accept any number of arguments. The base case handles the scenario where no arguments are left to print, while the recursive case processes the first argument and recursively calls itself with the remaining arguments.

Variadic Templates in Class Templates Variadic templates can also be used in class templates to create data structures that can handle a variable number of types. For example, let's implement a simple tuple class that can hold a heterogeneous collection of elements:

```cpp
#include <iostream>

// Base case: empty tuple
template<typename... Args>
class Tuple {};

// Recursive case: tuple with at least one element
template<typename Head, typename... Tail>
class Tuple<Head, Tail...> {
public:
    Head head;
    Tuple<Tail...> tail;

    Tuple(Head h, Tail... t) : head(h), tail(t...) {}

    void print() const {
        std::cout << head << std::endl;
        tail.print();
    }
};

// Specialization for empty tuple
```

```
template<>
class Tuple<> {
public:
    void print() const {}
};

int main() {
    Tuple<int, double, const char*, char> t(1, 2.5, "Hello", 'A');
    t.print();
    // Output:
    // 1
    // 2.5
    // Hello
    // A
    return 0;
}
```

In this example, the `Tuple` class template can hold an arbitrary number of elements. The recursive case defines a tuple with at least one element, while the specialization handles the empty tuple. The `print` function recursively prints each element of the tuple.

Variadic Template Functions with Fold Expressions C++17 introduced fold expressions, which simplify the expansion of parameter packs by applying a binary operator to each element. This feature makes it easier to implement operations that involve all elements of a parameter pack.

Here is an example of using a fold expression to implement a variadic function that sums all its arguments:

```
#include <iostream>

// Variadic sum function using fold expression
template<typename... Args>
auto sum(Args... args) {
    return (args + ...); // Fold expression
}

int main() {
    std::cout << "Sum: " << sum(1, 2, 3, 4, 5) << std::endl; // Output: Sum: 15
    std::cout << "Sum: " << sum(1.1, 2.2, 3.3) << std::endl; // Output: Sum: 6.6
    return 0;
}
```

In this example, the `sum` function uses a fold expression to compute the sum of all its arguments. The expression `(args + ...)` applies the `+` operator to each element in the parameter pack.

Advanced Variadic Template Techniques Variadic templates can be used in combination with other C++ features to create powerful and flexible abstractions. One such technique involves perfect forwarding, which allows the preservation of the value categories of arguments (lvalues and rvalues) when passing them to another function.

Here's an example of using variadic templates with perfect forwarding to implement a factory function that constructs objects of various types:

```
#include <iostream>
#include <memory>

// Factory function using variadic templates and perfect forwarding
template<typename T, typename... Args>
std::unique_ptr<T> create(Args&&... args) {
    return std::make_unique<T>(std::forward<Args>(args)...);
}
```

```cpp
class MyClass {
public:
    MyClass(int x, double y) {
        std::cout << "MyClass constructor called with x = " << x << " and y = " << y <<
        ↪  std::endl;
    }
};

int main() {
    auto obj = create<MyClass>(42, 3.14);
    // Output: MyClass constructor called with x = 42 and y = 3.14
    return 0;
}
```

In this example, the `create` function uses variadic templates and perfect forwarding to construct an object of type `T` with the provided arguments. The `std::forward` function ensures that the value categories of the arguments are preserved.

Variadic Templates and Compile-Time Computations Variadic templates can also be used for compile-time computations, such as calculating the size of a parameter pack or performing compile-time checks on the types of the arguments.

Here's an example of using variadic templates to implement a function that checks if all arguments are of the same type:

```cpp
#include <iostream>
#include <type_traits>

// Helper struct to check if all types are the same
template<typename T, typename... Args>
struct are_all_same;

template<typename T>
struct are_all_same<T> : std::true_type {};

template<typename T, typename U, typename... Args>
struct are_all_same<T, U, Args...> : std::false_type {};

template<typename T, typename... Args>
struct are_all_same<T, T, Args...> : are_all_same<T, Args...> {};

// Function that uses the are_all_same trait
template<typename T, typename... Args>
std::enable_if_t<are_all_same<T, Args...>::value, void> check_types(T, Args...) {
    std::cout << "All arguments are of the same type." << std::endl;
}

template<typename T, typename... Args>
std::enable_if_t<!are_all_same<T, Args...>::value, void> check_types(T, Args...) {
    std::cout << "Arguments are of different types." << std::endl;
}

int main() {
    check_types(1, 2, 3);           // Output: All arguments are of the same type.
    check_types(1, 2.0, 3);         // Output: Arguments are of different types.
    check_types("Hello", "World"); // Output: All arguments are of the same type.
    return 0;
}
```

In this example, the `are_all_same` struct uses variadic templates to check if all types in the parameter pack are the same. The `check_types` function uses this trait to print a message indicating whether all arguments are of the same type.

Conclusion Variadic templates are a versatile and powerful feature in C++ that allow for the creation of functions and classes that can accept an arbitrary number of arguments. By leveraging parameter packs, recursive template instantiation, fold expressions, perfect forwarding, and compile-time computations, variadic templates enable the development of highly generic and reusable code. Understanding and mastering variadic templates will greatly enhance your ability to write flexible and efficient C++ programs, making them an essential tool in modern C++ programming.

1.4. Template Specialization

Understanding Template Specialization Template specialization is a powerful feature in C++ that allows developers to define different implementations of a template for specific types or conditions. By using template specialization, you can provide customized behavior for particular types while maintaining the generality and flexibility of templates. There are two main types of template specialization: full specialization and partial specialization.

Full Template Specialization Full template specialization involves providing a completely different implementation of a template for a specific type. This is useful when you need to handle particular cases differently from the general case.

Consider the following example where we define a general `Max` template to find the maximum of two values, and then specialize it for `const char*` to compare C-style strings:

```cpp
#include <iostream>
#include <cstring>

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

// Full specialization for const char* to compare C-style strings
template<>
const char* Max<const char*>(const char* a, const char* b) {
    return (std::strcmp(a, b) > 0) ? a : b;
}

int main() {
    int x = 10, y = 20;
    std::cout << "Max of " << x << " and " << y << " is " << Max(x, y) << std::endl;

    const char* str1 = "Hello";
    const char* str2 = "World";
    std::cout << "Max of \"" << str1 << "\" and \"" << str2 << "\" is \"" << Max(str1, str2)
    ↪  << "\"" << std::endl;

    return 0;
}
```

In this example, the general `Max` template works for any type that supports the `>` operator. The full specialization for `const char*` uses `std::strcmp` to compare C-style strings. This ensures that the correct comparison function is used for different types.

Partial Template Specialization Partial template specialization allows you to specialize a template for a subset of its parameters, providing more flexibility and control over template behavior. This is particularly useful for class templates where you may want to provide different implementations based on some, but not all, of the template parameters.

Consider the following example where we define a general `Storage` template class and then partially specialize it for pointers:

```cpp
#include <iostream>

// General template for storing a value
template<typename T>
class Storage {
public:
    explicit Storage(T value) : value(value) {}

    void print() const {
        std::cout << "Value: " << value << std::endl;
    }

private:
    T value;
};

// Partial specialization for pointers
template<typename T>
class Storage<T*> {
public:
    explicit Storage(T* value) : value(value) {}

    void print() const {
        if (value) {
            std::cout << "Pointer value: " << *value << std::endl;
        } else {
            std::cout << "Null pointer" << std::endl;
        }
    }

private:
    T* value;
};

int main() {
    Storage<int> s1(42);
    s1.print(); // Output: Value: 42

    int x = 100;
    Storage<int*> s2(&x);
    s2.print(); // Output: Pointer value: 100

    Storage<int*> s3(nullptr);
    s3.print(); // Output: Null pointer

    return 0;
}
```

In this example, the general `Storage` template stores a value of type `T` and provides a `print` method to display the value. The partial specialization for pointers stores a pointer to `T` and provides a `print` method that dereferences the pointer if it is not null.

Practical Example: Type Traits Type traits are a common use case for template specialization. They allow you to query and manipulate types at compile time, enabling more flexible and generic code. The standard library provides many type traits, but you can also define your own.

Consider the following example where we define a simple type trait to check if a type is a pointer:

```
#include <iostream>
#include <type_traits>

// Primary template: T is not a pointer
template<typename T>
struct is_pointer : std::false_type {};

// Partial specialization: T* is a pointer
template<typename T>
struct is_pointer<T*> : std::true_type {};

int main() {
    std::cout << "int: " << is_pointer<int>::value << std::endl;          // Output: 0
    std::cout << "int*: " << is_pointer<int*>::value << std::endl;        // Output: 1
    std::cout << "double*: " << is_pointer<double*>::value << std::endl; // Output: 1
    return 0;
}
```

In this example, the primary template for `is_pointer` inherits from `std::false_type`, indicating that the type is not a pointer. The partial specialization for `T*` inherits from `std::true_type`, indicating that the type is a pointer.

Advanced Example: Custom Allocator Template specialization can also be used to provide custom implementations for specific types. Consider a custom allocator that behaves differently for fundamental types and user-defined types:

```
#include <iostream>
#include <memory>

// General template for custom allocator
template<typename T>
class CustomAllocator {
public:
    T* allocate(size_t n) {
        std::cout << "Allocating " << n << " objects of type " << typeid(T).name() <<
        ↪ std::endl;
        return static_cast<T*>(::operator new(n * sizeof(T)));
    }

    void deallocate(T* p, size_t n) {
        std::cout << "Deallocating " << n << " objects of type " << typeid(T).name() <<
        ↪ std::endl;
        ::operator delete(p);
    }
};

// Specialization for fundamental types
template<typename T>
class CustomAllocator<T*> {
public:
    T* allocate(size_t n) {
        std::cout << "Allocating " << n << " pointers to type " << typeid(T).name() <<
        ↪ std::endl;
        return static_cast<T*>(::operator new(n * sizeof(T*)));
    }

    void deallocate(T* p, size_t n) {
        std::cout << "Deallocating " << n << " pointers to type " << typeid(T).name() <<
        ↪ std::endl;
        ::operator delete(p);
    }
};

int main() {
    CustomAllocator<int> intAllocator;
    int* intArray = intAllocator.allocate(5);
    intAllocator.deallocate(intArray, 5);

    CustomAllocator<int*> ptrAllocator;
    int** ptrArray = ptrAllocator.allocate(5);
    ptrAllocator.deallocate(ptrArray, 5);

    return 0;
}
```

In this example, the `CustomAllocator` class template provides an allocation and deallocation mechanism for general types. The specialization for pointers adjusts the allocation and deallocation process to handle pointers specifically.

Combining Specialization with SFINAE Template specialization can be combined with SFINAE to create highly flexible templates that adapt based on complex type traits. This can be particularly useful in generic programming and library development.

Consider the following example where we use SFINAE and template specialization to create a function that prints values differently based on whether they are pointers or not:

```
#include <iostream>
#include <type_traits>

// General template for printing values
template<typename T>
void printValue(T value, std::false_type) {
    std::cout << "Value: " << value << std::endl;
}

// Specialization for printing pointers
template<typename T>
void printValue(T value, std::true_type) {
    if (value) {
        std::cout << "Pointer value: " << *value << std::endl;
    } else {
        std::cout << "Null pointer" << std::endl;
    }
}

// Wrapper function that dispatches to the correct print function
template<typename T>
void printValue(T value) {
    printValue(value, std::is_pointer<T>{});
}

int main() {
    int x = 42;
    int* ptr = &x;
    int* nullPtr = nullptr;

    printValue(x);       // Output: Value: 42
    printValue(ptr);     // Output: Pointer value: 42
    printValue(nullPtr); // Output: Null pointer
```

```cpp
    return 0;
}
```

In this example, the `printValue` function uses SFINAE and template specialization to handle different types. The general template handles non-pointer types, while the specialization handles pointers. The wrapper function determines the correct print function to call based on the type trait `std::is_pointer`.

Conclusion Template specialization is a versatile and powerful feature in C++ that allows developers to tailor the behavior of templates for specific types or conditions. By leveraging full specialization, partial specialization, and combining these with other metaprogramming techniques like SFINAE, you can create highly flexible and efficient code. Understanding template specialization will enable you to write more robust and adaptable C++ programs, making it an essential skill for modern C++ developers.

1.5. Compile-Time Data Structures

Understanding Compile-Time Data Structures Compile-time data structures are a powerful concept in C++ that leverage the capabilities of template metaprogramming to create and manipulate data structures entirely during the compilation process. This approach can lead to significant performance improvements because the computations are performed by the compiler, resulting in no runtime overhead. Compile-time data structures are particularly useful in scenarios where the structure and operations on the data can be determined at compile time, providing both efficiency and type safety.

Basics of Compile-Time Data Structures To understand compile-time data structures, we need to delve into some foundational concepts of template metaprogramming, such as recursive templates, template specialization, and constexpr functions. Let's start with a simple example of a compile-time linked list.

Compile-Time Linked List A compile-time linked list can be implemented using recursive templates. Each node in the list is represented by a template that holds a value and a link to the next node.

```cpp
#include <iostream>

// Compile-time representation of an empty list
struct Nil {};

// Node template for the linked list
template<int Head, typename Tail = Nil>
struct List {
    static constexpr int head = Head;
    using tail = Tail;
};

// Function to print the compile-time list
void printList(Nil) {
    std::cout << "Nil" << std::endl;
}

template<int Head, typename Tail>
void printList(List<Head, Tail>) {
    std::cout << Head << " -> ";
    printList(Tail{});
}

int main() {
    using MyList = List<1, List<2, List<3, List<4>>>>;
    printList(MyList{});
    // Output: 1 -> 2 -> 3 -> 4 -> Nil
    return 0;
}
```

In this example, `List` is a template that represents a node in the linked list. The `head` is the value stored in the node, and `tail` is the next node in the list. The `printList` function recursively prints the elements of the list.

Compile-Time Binary Tree Compile-time data structures are not limited to linear structures like lists. They can also represent more complex structures, such as binary trees. Let's create a compile-time binary tree and implement some basic operations.

```cpp
#include <iostream>

// Compile-time representation of an empty tree
struct EmptyTree {};

// Node template for the binary tree
template<int Value, typename Left = EmptyTree, typename Right = EmptyTree>
struct Tree {
    static constexpr int value = Value;
    using left = Left;
    using right = Right;
};

// Function to print the compile-time binary tree (in-order traversal)
void printTree(EmptyTree) {}

template<int Value, typename Left, typename Right>
void printTree(Tree<Value, Left, Right>) {
    printTree(Left{});
    std::cout << Value << " ";
    printTree(Right{});
}

int main() {
    using MyTree = Tree<4, Tree<2, Tree<1>, Tree<3>>, Tree<6, Tree<5>, Tree<7>>>;
    printTree(MyTree{});
    // Output: 1 2 3 4 5 6 7
    return 0;
}
```

In this example, `Tree` is a template that represents a node in the binary tree. The `value` is the value stored in the node, `left` is the left subtree, and `right` is the right subtree. The `printTree` function recursively prints the elements of the tree in an in-order traversal.

Compile-Time Map A compile-time map (or dictionary) can also be implemented using template metaprogramming. A simple implementation might use a list of key-value pairs.

```cpp
#include <iostream>

// Compile-time representation of an empty map
struct EmptyMap {};

// Key-value pair template
template<int Key, int Value>
struct Pair {
    static constexpr int key = Key;
    static constexpr int value = Value;
};

// Map template
template<typename... Pairs>
struct Map {};
```

```
// Function to get a value from the map by key
template<int Key, typename Map>
struct Get;

template<int Key>
struct Get<Key, EmptyMap> {
    static constexpr int value = -1; // Indicate key not found
};

template<int Key, int Value, typename... Rest>
struct Get<Key, Map<Pair<Key, Value>, Rest...>> {
    static constexpr int value = Value;
};

template<int Key, int OtherKey, int OtherValue, typename... Rest>
struct Get<Key, Map<Pair<OtherKey, OtherValue>, Rest...>> {
    static constexpr int value = Get<Key, Map<Rest...>>::value;
};

int main() {
    using MyMap = Map<Pair<1, 100>, Pair<2, 200>, Pair<3, 300>>;
    constexpr int value1 = Get<1, MyMap>::value; // 100
    constexpr int value2 = Get<2, MyMap>::value; // 200
    constexpr int value3 = Get<4, MyMap>::value; // -1 (not found)

    std::cout << "Value for key 1: " << value1 << std::endl;
    std::cout << "Value for key 2: " << value2 << std::endl;
    std::cout << "Value for key 3: " << value3 << std::endl; // Output: -1

    return 0;
}
```

In this example, `Pair` represents a key-value pair, and `Map` is a template that can hold multiple pairs. The `Get` template recursively searches the map for the given key and retrieves the associated value. If the key is not found, it returns `-1`.

Compile-Time Algorithms Compile-time data structures enable the implementation of algorithms that operate entirely at compile time. These algorithms can be used for metaprogramming tasks such as type manipulation and static analysis.

Consider a compile-time algorithm that calculates the factorial of a number using recursive templates:

```
#include <iostream>

// Template to calculate factorial at compile time
template<int N>
struct Factorial {
    static constexpr int value = N * Factorial<N - 1>::value;
};

// Specialization for the base case
template<>
struct Factorial<0> {
    static constexpr int value = 1;
};

int main() {
    constexpr int result = Factorial<5>::value;
    std::cout << "Factorial of 5 is " << result << std::endl; // Output: 120
    return 0;
}
```

In this example, the `Factorial` template recursively calculates the factorial of a number at compile time. The specialization for `Factorial<0>` provides the base case.

Using `constexpr` for Compile-Time Data Structures The `constexpr` keyword in C++11 and later versions allows for the evaluation of functions and objects at compile time. This enables more complex compile-time data structures and algorithms.

Consider a compile-time vector implemented using `constexpr`:

```cpp
#include <iostream>
#include <array>

// Compile-time vector class
template<typename T, std::size_t N>
class Vector {
public:
    constexpr Vector() : data{} {}

    constexpr T& operator[](std::size_t index) {
        return data[index];
    }

    constexpr const T& operator[](std::size_t index) const {
        return data[index];
    }

    constexpr std::size_t size() const {
        return N;
    }

private:
    std::array<T, N> data;
};

int main() {
    constexpr Vector<int, 5> vec;
    static_assert(vec.size() == 5, "Size check failed");

    // Print elements of the compile-time vector
    for (std::size_t i = 0; i < vec.size(); ++i) {
        std::cout << vec[i] << " "; // Default-initialized to zero
    }
    std::cout << std::endl;

    return 0;
}
```

In this example, the `Vector` class template provides a simple implementation of a fixed-size vector that can be evaluated at compile time. The `size` member function and the element access operators are marked as `constexpr` to enable compile-time evaluation.

Compile-Time String Manipulation Compile-time string manipulation can be achieved using template metaprogramming and `constexpr`. This can be useful for tasks such as compile-time hashing or static analysis.

Here's an example of compile-time string concatenation:

```cpp
#include <iostream>

// Compile-time string representation
```

```
template<std::size_t N>
struct CompileTimeString {
    char data[N];

    constexpr CompileTimeString(const char (&str)[N]) {
        for (std::size_t i = 0; i < N; ++i) {
            data[i] = str[i];
        }
    }
};

// Concatenate two compile-time strings
template<std::size_t N, std::size_t M>
constexpr auto concat(const CompileTimeString<N>& a, const CompileTimeString<M>& b) {
    char result[N + M - 1] = {};
    for (std::size_t i = 0; i < N - 1; ++i) {
        result[i] = a.data[i];
    }
    for (std::size_t i = 0; i < M; ++i) {
        result[N - 1 + i] = b.data[i];
    }
    return CompileTimeString<N + M - 1>{result};
}

int main() {
    constexpr CompileTimeString str1{"Hello, "};
    constexpr CompileTimeString str2{"World!"};
    constexpr auto result = concat(str1, str2);

    std::cout << result.data << std::endl; // Output: Hello

, World!

    return 0;
}
```

In this example, `CompileTimeString` represents a string at compile time, and `concat` is a `constexpr` function that concatenates two compile-time strings. The result is evaluated at compile time, and the concatenated string is printed.

Conclusion Compile-time data structures in C++ are a powerful tool for writing efficient and type-safe code. By leveraging template metaprogramming, `constexpr`, and recursive templates, developers can create and manipulate data structures entirely during the compilation process. This approach can lead to significant performance improvements and provides robust compile-time guarantees. Understanding and utilizing compile-time data structures will greatly enhance your ability to write high-performance C++ code and leverage the full potential of the language's metaprogramming capabilities.

1.6. Compile-Time Algorithms

Understanding Compile-Time Algorithms Compile-time algorithms leverage the power of C++ template metaprogramming and `constexpr` functions to perform computations during the compilation process. This approach can lead to highly efficient and optimized code, as the results of these computations are embedded directly into the compiled binary, eliminating runtime overhead. Compile-time algorithms are particularly useful in scenarios where the inputs and the logic are known at compile time, allowing for precomputation and constant folding.

Basics of Compile-Time Algorithms To implement compile-time algorithms, we use templates and `constexpr` functions. Templates allow us to define algorithms that operate on types, while `constexpr` enables functions to be evaluated at compile time. Let's start with a simple example of a compile-time factorial calculation using both templates and `constexpr`.

Compile-Time Factorial Using Templates The factorial of a number can be computed recursively using templates. Here's how you can implement it:

```cpp
#include <iostream>

// Template for compile-time factorial calculation
template<int N>
struct Factorial {
    static constexpr int value = N * Factorial<N - 1>::value;
};

// Specialization for base case
template<>
struct Factorial<0> {
    static constexpr int value = 1;
};

int main() {
    constexpr int result = Factorial<5>::value;
    std::cout << "Factorial of 5 is " << result << std::endl; // Output: 120
    return 0;
}
```

In this example, `Factorial<N>` recursively calculates the factorial of `N`. The specialization for `Factorial<0>` provides the base case, terminating the recursion.

Compile-Time Factorial Using `constexpr` We can achieve the same result using a `constexpr` function, which is more readable and easier to debug:

```cpp
#include <iostream>

// Constexpr function for compile-time factorial calculation
constexpr int factorial(int n) {
    return (n <= 1) ? 1 : (n * factorial(n - 1));
}

int main() {
    constexpr int result = factorial(5);
    std::cout << "Factorial of 5 is " << result << std::endl; // Output: 120
    return 0;
}
```

In this example, the `factorial` function is marked as `constexpr`, allowing it to be evaluated at compile time. The function uses a simple recursive approach to compute the factorial.

Compile-Time Fibonacci Sequence The Fibonacci sequence is another classic example of a recursive algorithm that can be implemented at compile time using both templates and `constexpr`.

Using Templates

```cpp
#include <iostream>

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

```cpp
// Specializations for base cases
template<>
struct Fibonacci<0> {
    static constexpr int value = 0;
};

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

int main() {
    constexpr int result = Fibonacci<10>::value;
    std::cout << "Fibonacci of 10 is " << result << std::endl; // Output: 55
    return 0;
}
```

In this example, `Fibonacci<N>` recursively calculates the `N`th Fibonacci number. The specializations for `Fibonacci<0>` and `Fibonacci<1>` provide the base cases.

Using `constexpr`

```cpp
#include <iostream>

// Constexpr function for compile-time Fibonacci calculation
constexpr int fibonacci(int n) {
    return (n <= 1) ? n : (fibonacci(n - 1) + fibonacci(n - 2));
}

int main() {
    constexpr int result = fibonacci(10);
    std::cout << "Fibonacci of 10 is " << result << std::endl; // Output: 55
    return 0;
}
```

In this example, the `fibonacci` function is marked as `constexpr` and uses recursion to compute the Fibonacci number at compile time.

Compile-Time Prime Check Checking if a number is prime can also be done at compile time. This involves dividing the number by potential factors and ensuring that it is not divisible by any of them.

Using Templates

```cpp
#include <iostream>

// Helper template to check divisibility
template<int N, int D>
struct IsPrimeHelper {
    static constexpr bool value = (N % D != 0) && IsPrimeHelper<N, D - 1>::value;
};

// Specialization for the base case
template<int N>
struct IsPrimeHelper<N, 1> {
    static constexpr bool value = true;
};

// Template for compile-time prime check
template<int N>
```

```cpp
struct IsPrime {
    static constexpr bool value = IsPrimeHelper<N, N / 2>::value;
};

// Special cases for 0 and 1
template<>
struct IsPrime<0> {
    static constexpr bool value = false;
};

template<>
struct IsPrime<1> {
    static constexpr bool value = false;
};

int main() {
    constexpr bool result = IsPrime<17>::value;
    std::cout << "Is 17 prime? " << (result ? "Yes" : "No") << std::endl; // Output: Yes
    return 0;
}
```

In this example, `IsPrimeHelper<N, D>` recursively checks if `N` is divisible by any number from `D` down to `1`. The `IsPrime<N>` template uses `IsPrimeHelper` to determine if `N` is prime, with specializations for `0` and `1`.

Using `constexpr`

```cpp
#include <iostream>

// Constexpr function for compile-time prime check
constexpr bool isPrimeHelper(int n, int d) {
    return (d == 1) ? true : (n % d != 0) && isPrimeHelper(n, d - 1);
}

constexpr bool isPrime(int n) {
    return (n <= 1) ? false : isPrimeHelper(n, n / 2);
}

int main() {
    constexpr bool result = isPrime(17);
    std::cout << "Is 17 prime? " << (result ? "Yes" : "No") << std::endl; // Output: Yes
    return 0;
}
```

In this example, the `isPrimeHelper` and `isPrime` functions are marked as `constexpr`, allowing them to be evaluated at compile time. The functions recursively check if `n` is prime.

Compile-Time Sorting Sorting algorithms can also be implemented at compile time using template metaprogramming. Let's implement a simple compile-time quicksort algorithm.

Using Templates

```cpp
#include <iostream>
#include <array>

// Base case for empty array
template<typename T, std::size_t N>
struct QuickSort {
    static constexpr std::array<T, N> sort(const std::array<T, N>& arr) {
        return arr;
    }
```

```
};

// Partition function for quicksort
template<typename T, std::size_t N>
constexpr std::pair<std::array<T, N>, std::array<T, N>> partition(const std::array<T, N>&
↪  arr, T pivot) {
    std::array<T, N> left = {};
    std::array<T, N> right = {};
    std::size_t leftIndex = 0, rightIndex = 0;

    for (std::size_t i = 0; i < N; ++i) {
        if (arr[i] < pivot) {
            left[leftIndex++] = arr[i];
        } else if (arr[i] > pivot) {
            right[rightIndex++] = arr[i];
        }
    }

    return {left, right};
}

// QuickSort implementation
template<typename T, std::size_t N>
struct QuickSort {
    static constexpr std::array<T, N> sort(const std::array<T, N>& arr) {
        if constexpr (N <= 1) {
            return arr;
        } else {
            T pivot = arr[N / 2];
            auto [left, right] = partition(arr, pivot);
            auto sortedLeft = QuickSort<T, left.size()>::sort(left);
            auto sortedRight = QuickSort<T, right.size()>::sort(right);
            return merge(sortedLeft, pivot, sortedRight);
        }
    }

    static constexpr std::array<T, N + 1> merge(const std::array<T, N>& left, T pivot, const
    ↪  std::array<T, N>& right) {
        std::array<T, N + 1> result = {};
        std::size_t index = 0;
        for (std::size_t i = 0; i < left.size(); ++i) {
            result[index++] = left[i];
        }
        result[index++] = pivot;
        for (std::size_t i = 0; i < right.size(); ++i) {
            result[index++] = right[i];
        }
        return result;
    }
};

int main() {
    constexpr std::array<int, 5> arr = {3, 1, 4, 1, 5};
    constexpr auto sortedArr = QuickSort<int, arr.size()>::sort(arr);

    for (int i : sortedArr) {
        std::cout << i << " ";
    }
    std::cout << std::endl; // Output: 1 1 3 4 5

    return 0;
}
```

In this example, `QuickSort` recursively sorts the array using the quicksort algorithm. The `partition` function divides the array into two subarrays based on a pivot, and the `merge` function combines the sorted subarrays with the pivot.

Advanced Compile-Time Algorithms Advanced compile-time algorithms can involve more complex data structures

and logic. Let's implement a compile-time matrix multiplication algorithm.

```cpp
#include <iostream>
#include <array>

// Compile-time matrix multiplication
template<std::size_t N, std::size_t M, std::size_t P>
constexpr std::array<std::array<int, P>, N> multiply(const std::array<std::array<int, M>,
↪   N>& a, const std::array<std::array<int, P>, M>& b) {
    std::array<std::array<int, P>, N> result = {};

    for (std::size_t i = 0; i < N; ++i) {
        for (std::size_t j = 0; j < P; ++j) {
            int sum = 0;
            for (std::size_t k = 0; k < M; ++k) {
                sum += a[i][k] * b[k][j];
            }
            result[i][j] = sum;
        }
    }

    return result;
}

int main() {
    constexpr std::array<std::array<int, 2>, 2> a = {{{1, 2}, {3, 4}}};
    constexpr std::array<std::array<int, 2>, 2> b = {{{5, 6}, {7, 8}}};
    constexpr auto result = multiply(a, b);

    for (const auto& row : result) {
        for (int val : row) {
            std::cout << val << " ";
        }
        std::cout << std::endl;
    }
    // Output:
    // 19 22
    // 43 50

    return 0;
}
```

In this example, the `multiply` function performs matrix multiplication at compile time. The resulting matrix is computed during compilation, eliminating the need for runtime computation.

Conclusion Compile-time algorithms in C++ offer a powerful way to optimize code by performing computations during the compilation process. By leveraging template metaprogramming and `constexpr` functions, developers can implement a wide range of algorithms, from simple arithmetic operations to complex data structure manipulations, entirely at compile time. Understanding and utilizing compile-time algorithms can lead to significant performance improvements and more robust code, making them an essential tool for modern C++ programming.

1.7. Metafunctions and Metafunction Classes

Understanding Metafunctions Metafunctions are a fundamental concept in C++ template metaprogramming. They are templates that take one or more types as arguments and produce a single type or value as their result. Essentially, metafunctions allow you to perform computations at the type level. These computations can include type transformations, type checks, and more, providing a powerful mechanism for creating flexible and reusable code.

Basics of Metafunctions A typical metafunction is a template that uses `typedef` or `using` to define its result. Let's start with a simple example of a metafunction that removes the const qualifier from a type:

```cpp
#include <iostream>
#include <type_traits>

// Metafunction to remove const qualifier
template<typename T>
struct RemoveConst {
    using type = T;
};

template<typename T>
struct RemoveConst<const T> {
    using type = T;
};

int main() {
    std::cout << std::is_same<int, RemoveConst<const int>::type>::value << std::endl; //
    ↪  Output: 1 (true)
    std::cout << std::is_same<int, RemoveConst<int>::type>::value << std::endl;       //
    ↪  Output: 1 (true)
    return 0;
}
```

In this example, `RemoveConst` is a metafunction that removes the `const` qualifier from a type if it is present. The primary template handles non-const types, while the specialization handles const types.

Metafunction Classes Metafunction classes are a higher-level abstraction that encapsulates metafunctions within a class. This encapsulation allows for more complex and reusable type-level computations. Metafunction classes can be passed as template arguments to other templates, enabling higher-order template metaprogramming.

Basic Metafunction Class Let's create a simple metafunction class that adds a pointer to a given type:

```cpp
#include <iostream>
#include <type_traits>

// Metafunction class to add a pointer to a type
struct AddPointer {
    template<typename T>
    struct apply {
        using type = T*;
    };
};

int main() {
    using T = AddPointer::apply<int>::type;
    std::cout << std::is_same<int*, T>::value << std::endl; // Output: 1 (true)
    return 0;
}
```

In this example, `AddPointer` is a metafunction class with an inner `apply` template that adds a pointer to the given type. The result is accessed using `AddPointer::apply<int>::type`.

Advanced Metafunction Classes Metafunction classes can be used to create more advanced type manipulations and checks. Let's implement a metafunction class that checks if a type is a pointer and, if so, returns the pointed-to type; otherwise, it returns the original type.

```cpp
#include <iostream>
#include <type_traits>

// Metafunction class to remove pointer if present
struct RemovePointer {
    template<typename T>
    struct apply {
        using type = T;
    };

    template<typename T>
    struct apply<T*> {
        using type = T;
    };
};

int main() {
    using T1 = RemovePointer::apply<int>::type;
    using T2 = RemovePointer::apply<int*>::type;

    std::cout << std::is_same<int, T1>::value << std::endl;   // Output: 1 (true)
    std::cout << std::is_same<int, T2>::value << std::endl;   // Output: 1 (true)
    std::cout << std::is_same<int*, T1>::value << std::endl;  // Output: 0 (false)
    std::cout << std::is_same<int*, T2>::value << std::endl;  // Output: 0 (false)
    return 0;
}
```

In this example, `RemovePointer` is a metafunction class with an inner `apply` template. The primary template handles non-pointer types, while the specialization handles pointer types by returning the pointed-to type.

Combining Metafunctions and Metafunction Classes Metafunctions and metafunction classes can be combined to create more complex and powerful metaprogramming constructs. For example, let's create a metafunction class that applies a sequence of metafunctions to a type.

Chaining Metafunction Classes

```cpp
#include <iostream>
#include <type_traits>

// Metafunction class to add a pointer to a type
struct AddPointer {
    template<typename T>
    struct apply {
        using type = T*;
    };
};

// Metafunction class to remove const qualifier
struct RemoveConst {
    template<typename T>
    struct apply {
        using type = T;
    };

    template<typename T>
    struct apply<const T> {
        using type = T;
    };
};

// Metafunction class to chain multiple metafunctions
template<typename F1, typename F2>
struct Compose {
    template<typename T>
    struct apply {
        using type = typename F1::template apply<typename F2::template
         ↪ apply<T>::type>::type;
    };
};

int main() {
    using T = Compose<AddPointer, RemoveConst>::apply<const int>::type;
    std::cout << std::is_same<int*, T>::value << std::endl; // Output: 1 (true)
    return 0;
}
```

In this example, `Compose` is a metafunction class that chains two metafunction classes, `F1` and `F2`. It applies `F2` to the input type `T` and then applies `F1` to the result. This demonstrates the power of combining metafunction classes to create complex type transformations.

Practical Applications of Metafunctions Metafunctions and metafunction classes are widely used in template metaprogramming to create flexible and reusable code. Some practical applications include type traits, compile-time computations, and type transformations.

Example: Enable If The `std::enable_if` utility is a common use case of metafunctions. It conditionally enables or disables function templates based on a compile-time condition.

```cpp
#include <iostream>
#include <type_traits>

// Function enabled only for integral types
template<typename T>
std::enable_if_t<std::is_integral_v<T>, void> print(T value) {
    std::cout << value << " is an integral type." << std::endl;
}

// Function enabled only for floating-point types
template<typename T>
std::enable_if_t<std::is_floating_point_v<T>, void> print(T value) {
    std::cout << value << " is a floating-point type." << std::endl;
}

int main() {
    print(42);      // Output: 42 is an integral type.
    print(3.14);    // Output: 3.14 is a floating-point type.
    // print("Hello"); // Compilation error: no matching function to call 'print'
    return 0;
}
```

In this example, `std::enable_if_t` is used to conditionally enable the `print` function for integral and floating-point types.

Custom Metafunction: Conditional Type Selection Let's implement a custom metafunction class that selects a type based on a compile-time condition, similar to `std::conditional`.

```cpp
#include <iostream>
#include <type_traits>

// Custom metafunction class to select a type based on a condition
struct Conditional {
    template<bool Condition, typename TrueType, typename FalseType>
    struct apply {
        using type = TrueType;
    };

    template<typename TrueType, typename FalseType>
    struct apply<false, TrueType, FalseType> {
        using type = FalseType;
    };
};

int main() {
    using T1 = Conditional::apply<true, int, double>::type;
    using T2 = Conditional::apply<false, int, double>::type;

    std::cout << std::is_same<int, T1>::value << std::endl;    // Output: 1 (true)
    std::cout << std::is_same<double, T2>::value << std::endl; // Output: 1 (true)
    return 0;
}
```

In this example, `Conditional` is a metafunction class that selects `TrueType` if `Condition` is true and `FalseType` otherwise. This demonstrates how to create custom metafunctions for type selection based on compile-time conditions.

Metafunctions for Compile-Time Computations Metafunctions can also be used for compile-time computations. Let's implement a metafunction class that calculates the greatest common divisor (GCD) of two integers at compile time.

```cpp
#include <iostream>

// Metafunction class to calculate GCD
struct GCD {
    template<int A, int B>
    struct apply {
        static constexpr int value = GCD::apply<B, A % B>::value;
    };

    template<int A>
    struct apply<A, 0> {
        static constexpr int value = A;
    };
};

int main() {
    constexpr int result = GCD::apply<48, 18>::value;
    std::cout << "GCD of 48 and 18 is " << result << std::endl; // Output: 6
    return 0;
}
```

In this example, `GCD` is a metafunction class that calculates the greatest common divisor of two integers using Euclid's algorithm. The result is computed at compile time and accessed using `GCD::apply<48, 18>::value`.

Conclusion Metafunctions and metafunction classes are powerful tools in C++ template metaprogramming that enable type-level computations and transformations. By leveraging these constructs, developers can create highly flexible and reusable

code, perform compile-time computations, and implement complex type manipulations. Understanding and utilizing metafunctions and metafunction classes will significantly enhance your ability to write sophisticated and efficient C++ programs, making them an essential skill for modern C++ development.

1.8. Type Traits and Type Manipulations

Understanding Type Traits Type traits are a crucial part of C++ template metaprogramming, providing a way to query and manipulate types at compile time. They enable developers to write more flexible and generic code by allowing the examination of types and their properties. The standard library provides a comprehensive set of type traits in the `<type_traits>` header, but you can also create custom type traits to suit specific needs.

Basics of Type Traits Type traits are typically implemented as templates that inherit from `std::true_type` or `std::false_type`, depending on whether the type property holds. Let's start with a simple example of a type trait that checks if a type is an integral type.

```cpp
#include <iostream>
#include <type_traits>

// Custom type trait to check if a type is integral
template<typename T>
struct is_integral : std::false_type {};

template<>
struct is_integral<int> : std::true_type {};

template<>
struct is_integral<short> : std::true_type {};

template<>
struct is_integral<long> : std::true_type {};

template<>
struct is_integral<long long> : std::true_type {};

// Test the custom type trait
int main() {
    std::cout << std::boolalpha;
    std::cout << "is_integral<int>::value: " << is_integral<int>::value << std::endl; //
    ↪  Output: true
    std::cout << "is_integral<float>::value: " << is_integral<float>::value << std::endl; //
    ↪  Output: false
    return 0;
}
```

In this example, the `is_integral` type trait determines whether a given type is one of the integral types by specializing the template for each integral type and inheriting from `std::true_type`.

Using Standard Type Traits The C++ standard library provides a rich set of type traits that can be used to query and manipulate types. Some common type traits include `std::is_integral`, `std::is_floating_point`, `std::is_pointer`, and `std::is_reference`. Here are some examples of using standard type traits:

```cpp
#include <iostream>
#include <type_traits>
```

```cpp
// Function to demonstrate type traits
template<typename T>
void check_type_traits() {
    std::cout << std::boolalpha;
    std::cout << "std::is_integral<T>::value: " << std::is_integral<T>::value << std::endl;
    std::cout << "std::is_floating_point<T>::value: " << std::is_floating_point<T>::value <<
    ↪  std::endl;
    std::cout << "std::is_pointer<T>::value: " << std::is_pointer<T>::value << std::endl;
    std::cout << "std::is_reference<T>::value: " << std::is_reference<T>::value <<
    ↪  std::endl;
}

int main() {
    std::cout << "Checking int:" << std::endl;
    check_type_traits<int>();

    std::cout << "\nChecking float:" << std::endl;
    check_type_traits<float>();

    std::cout << "\nChecking int*:" << std::endl;
    check_type_traits<int*>();

    std::cout << "\nChecking int&:" << std::endl;
    check_type_traits<int&>();

    return 0;
}
```

In this example, the `check_type_traits` function uses various standard type traits to query properties of different types. The output shows whether each type is integral, floating-point, a pointer, or a reference.

Type Manipulations Type manipulations involve transforming types to obtain new types. This can be done using type traits such as `std::remove_const`, `std::add_pointer`, `std::remove_reference`, and more. These type transformations are essential for creating generic and reusable code.

Removing Const

```cpp
#include <iostream>
#include <type_traits>

// Function to demonstrate removing const
template<typename T>
void remove_const_demo() {
    using NonConstType = typename std::remove_const<T>::type;
    std::cout << "Original type: " << typeid(T).name() << std::endl;
    std::cout << "Non-const type: " << typeid(NonConstType).name() << std::endl;
}

int main() {
    std::cout << "Removing const from const int:" << std::endl;
    remove_const_demo<const int>();

    return 0;
}
```

In this example, the `remove_const_demo` function uses `std::remove_const` to obtain the non-const version of a type. The `typeid` operator is used to display the original and transformed types.

Adding Pointer

```cpp
#include <iostream>
#include <type_traits>

// Function to demonstrate adding pointer
template<typename T>
void add_pointer_demo() {
    using PointerType = typename std::add_pointer<T>::type;
    std::cout << "Original type: " << typeid(T).name() << std::endl;
    std::cout << "Pointer type: " << typeid(PointerType).name() << std::endl;
}

int main() {
    std::cout << "Adding pointer to int:" << std::endl;
    add_pointer_demo<int>();

    return 0;
}
```

In this example, the `add_pointer_demo` function uses `std::add_pointer` to obtain the pointer version of a type.

Removing Reference

```cpp
#include <iostream>
#include <type_traits>

// Function to demonstrate removing reference
template<typename T>
void remove_reference_demo() {
    using NonReferenceType = typename std::remove_reference<T>::type;
    std::cout << "Original type: " << typeid(T).name() << std::endl;
    std::cout << "Non-reference type: " << typeid(NonReferenceType).name() << std::endl;
}

int main() {
    std::cout << "Removing reference from int&:" << std::endl;
    remove_reference_demo<int&>();

    return 0;
}
```

In this example, the `remove_reference_demo` function uses `std::remove_reference` to obtain the non-reference version of a type.

Custom Type Traits and Manipulations While the standard library provides many type traits and manipulations, there are cases where you may need to define custom type traits to suit specific needs. Let's create a custom type trait to check if a type is a smart pointer.

```cpp
#include <iostream>
#include <memory>
#include <type_traits>

// Primary template: T is not a smart pointer
template<typename T>
struct is_smart_pointer : std::false_type {};

// Specializations for std::unique_ptr and std::shared_ptr
template<typename T>
```

```cpp
struct is_smart_pointer<std::unique_ptr<T>> : std::true_type {};

template<typename T>
struct is_smart_pointer<std::shared_ptr<T>> : std::true_type {};

// Function to demonstrate custom type trait
template<typename T>
void check_smart_pointer() {
    std::cout << std::boolalpha;
    std::cout << "is_smart_pointer<T>::value: " << is_smart_pointer<T>::value << std::endl;
}

int main() {
    std::cout << "Checking int:" << std::endl;
    check_smart_pointer<int>();

    std::cout << "\nChecking std::unique_ptr<int>:" << std::endl;
    check_smart_pointer<std::unique_ptr<int>>();

    std::cout << "\nChecking std::shared_ptr<int>:" << std::endl;
    check_smart_pointer<std::shared_ptr<int>>();

    return 0;
}
```

In this example, `is_smart_pointer` is a custom type trait that checks if a type is a smart pointer (`std::unique_ptr` or `std::shared_ptr`). The `check_smart_pointer` function demonstrates the usage of this custom type trait.

Combining Type Traits Type traits can be combined to create more complex type manipulations. Let's create a type trait that adds a pointer to a type if it is not already a pointer.

```cpp
#include <iostream>
#include <type_traits>

// Type trait to add a pointer if not already a pointer
template<typename T>
struct add_pointer_if_not {
    using type = typename std::conditional<std::is_pointer<T>::value, T, typename
     ↪  std::add_pointer<T>::type>::type;
};

// Function to demonstrate the combined type trait
template<typename T>
void add_pointer_if_not_demo() {
    using ResultType = typename add_pointer_if_not<T>::type;
    std::cout << "Original type: " << typeid(T).name() << std::endl;
    std::cout << "Result type: " << typeid(ResultType).name() << std::endl;
}

int main() {
    std::cout << "Adding pointer to int:" << std::endl;
    add_pointer_if_not_demo<int>();

    std::cout << "\nAdding pointer to int*:" << std::endl;
    add_pointer_if_not_demo<int*>();

    return 0;
}
```

In this example, `add_pointer_if_not` is a custom type trait that uses `std::conditional` to add a pointer to a type if it is not already a pointer. The `add_pointer_if_not_demo` function demonstrates the usage of this combined type trait.

Practical Applications of Type Traits Type traits are extensively used in modern C++ programming for various purposes, including type checks, type transformations, and enabling/disabling template specializations.

SFINAE with Type Traits Substitution Failure Is Not An Error (SFINAE) is a technique that relies on type traits to enable or disable function templates based on type properties. Here's an example of using SFINAE with type traits to enable a function only for integral types.

```
cpp
#include <iostream>
#include <type_traits>

// Function enabled only for integral types
template<typename T>
std::enable_if_t<std::is_integral_v<T>, void> process(T value) {
    std::cout << value << " is an integral type." << std::endl;
}

// Function enabled only for floating-point types
template<typename T>
std::enable_if_t<std::is_floating_point_v<T>, void> process(T value) {
    std::cout << value << " is a floating-point type." << std::endl;
}

int main() {
    process(42);       // Output: 42 is an integral type.
    process(3.14);     // Output: 3.14 is a floating-point type.
    // process("text"); // Error: no matching function to call 'process'
    return 0;
}
```

In this example, `std::enable_if_t` and type traits (`std::is_integral_v` and `std::is_floating_point_v`) are used to conditionally enable the `process` function for integral and floating-point types.

Conclusion Type traits and type manipulations are powerful tools in C++ template metaprogramming that enable type-level queries and transformations. By leveraging standard type traits, creating custom type traits, and combining them for complex manipulations, developers can write more flexible and reusable code. Understanding and utilizing type traits and type manipulations will significantly enhance your ability to create sophisticated and efficient C++ programs, making them an essential skill for modern C++ development.