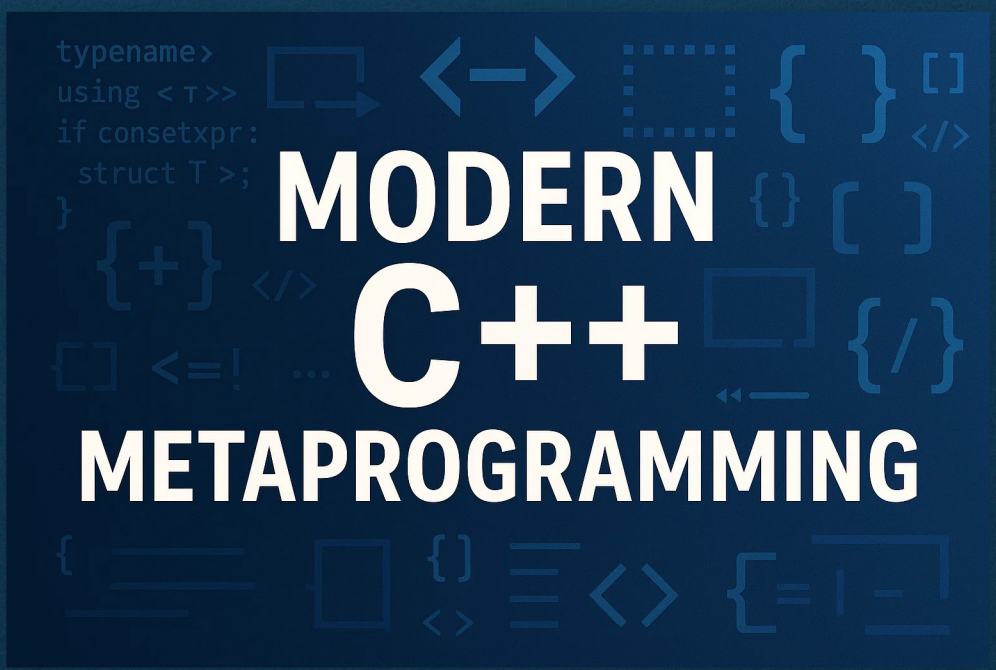
https://simplifyCpp.org

DRAFT

Metaprogramming Techniques in Modern C++



Prepared by Ayman Alheraki

First Edition

Metaprogramming Techniques in Modern C++

Prepared by Ayman Alheraki

simplifycpp.org

August 2025

Contents

Author's Notes

This booklet, **"Metaprogramming Techniques in Modern C++"**, was created to provide a **practical, concise, and modern overview** of metaprogramming in C++, from C++11 to the upcoming C++26 features.

Purpose

- To help C++ developers understand **compile-time computation, type manipulation, and code generation techniques**.
- To provide **examples and best practices** that can be directly applied in real-world projects.
- To introduce **upcoming C++26 proposals**, including reflection, pattern matching, and template loops, so programmers can prepare for the next evolution of C++.

Audience

- Intermediate to advanced C++ programmers who want to **leverage metaprogramming for efficiency, type safety, and maintainability**.
- Developers interested in **modern C++ design patterns and compile-time techniques**.

Approach

- Focused on **concise explanations and practical examples**, avoiding unnecessary complexity.
- Covers **core libraries, advanced libraries, and conceptual C++26 features**.
- Includes **appendices and references** for quick lookup and practical use.

Author's Vision Metaprogramming in C++ is a **powerful tool** for writing **safer, faster, and more maintainable software**. This booklet aims to **bridge the gap between traditional templates and modern C++ features**, giving readers the ability to adopt advanced compile-time techniques confidently.

Introduction

Definition of Metaprogramming

Metaprogramming is a programming technique where code is written to **generate, analyze, or modify other code** at **compile-time** rather than runtime. In C++, this allows developers to perform complex computations, enforce type constraints, and produce optimized code before the program runs.
The main advantages of metaprogramming include:

1. **Performance Optimization**: By computing values and generating code at compile-time, runtime overhead is reduced.

2. **Code Reusability and Reduction**: Common patterns and logic can be abstracted into templates or compile-time functions, reducing manual repetition.

3. **Stronger Type Safety**: Compile-time checks prevent type-related

errors before the program executes.

4. **Custom Code Generation**: Code can adapt automatically based on types or constants, enabling highly flexible and maintainable designs.

Modern C++ (post C++11) has expanded metaprogramming capabilities through:

- `constexpr` **functions and variables** for compile-time computation.
- **Template metaprogramming improvements** with variadic templates and type traits.
- **SFINAE and concepts** for conditional compilation and stricter type constraints.

Metaprogramming is not only a tool for advanced optimization; it is now a core part of writing **efficient, safe, and maintainable C++ code**.

Importance in C++

Metaprogramming plays a central role in modern C++ development by enabling more efficient, reliable, and adaptable software design. Its importance can be summarized in three key areas:

1. **Performance Optimization**
   Computations that would otherwise occur at runtime can be resolved at compile-time. This eliminates unnecessary runtime overhead and leads to faster, smaller executables.

2. **Type Checking and Safety**
   Templates, type traits, and concepts allow developers to enforce strict type rules during compilation. This prevents a wide class of programming errors and ensures that only valid code is generated.

3. **Custom Code Generation**
   Repetitive patterns can be expressed once in a generic template, which then adapts automatically to different types or values. This reduces code duplication, simplifies maintenance, and improves scalability.

Together, these benefits make metaprogramming a key technique for writing **high-performance, type-safe, and maintainable modern C++ applications**.

Chapter 1

Metaprogramming in C++11 and Beyond

1.1 Core Libraries

1.1.1 Type Traits

The <type_traits> library, introduced in C++11 and expanded in later standards, provides tools for compile-time type inspection and transformation. Type traits allow templates to adapt their behavior based on type properties, making them a cornerstone of metaprogramming.

Key Examples

- `std::is_integral<T>`: Checks if `T` is an integral type.
- `std::is_same<T, U>`: Checks if two types are identical.
- `std::remove_const<T>`: Produces the type `T` without the `const` qualifier.

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

template <typename T>
void process(T value) {
    if constexpr (std::is_integral_v<T>) {
        std::cout << "Processing integral: " << value << '\n';
    } else if constexpr (std::is_floating_point_v<T>) {
        std::cout << "Processing floating-point: " << value << '\n';
    } else {
        std::cout << "Unsupported type\n";
    }
}

int main() {
    process(42);        // Integral branch
    process(3.14);      // Floating-point branch
    process("text");    // Unsupported branch
}
```

Modern Usage Example

This example uses type traits with `if constexpr` (introduced in C++17) to select code paths at compile-time. No runtime overhead is introduced for unused branches.

Importance Type traits make templates more **flexible, safe, and expressive**. They eliminate the need for manual specialization in many cases and serve as the foundation for advanced techniques such as SFINAE, concepts, and static reflection.

1.1.2 SFINAE

SFINAE (Substitution Failure Is Not An Error) is a fundamental rule in C++ template resolution. It allows a template to be discarded when a substitution fails, rather than producing a compile-time error. This mechanism enables selective enabling or disabling of templates based on type properties.

Key Idea SFINAE makes it possible to write multiple overloads of a function or template and let the compiler choose the valid one depending on type traits or expressions.

Example with `std::enable_if`

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

```cpp
// Enabled only for integral types
template <typename T>
std::enable_if_t<std::is_integral_v<T>, void>
print(T value) {
    std::cout << "Integral: " << value << '\n';
}

// 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 << "Floating-point: " << value << '\n';
}

int main() {
    print(42);      // Matches integral overload
    print(3.14);    // Matches floating-point overload
    // print("test"); // Error: no valid overload
}
```

Here, `std::enable_if_t` is used to restrict which overloads are valid for a given type. The compiler discards invalid templates silently, following the SFINAE rule.

Modern Usage

- C++14 introduced `std::enable_if_t` as a shorthand for `typename std::enable_if<...>::type`.

- C++17 added `if constexpr`, which often replaces SFINAE for simpler cases.
- C++20 concepts provide a more readable and expressive alternative, though SFINAE remains relevant for advanced template metaprogramming.

Importance SFINAE enables **compile-time polymorphism**, ensuring only valid code is instantiated. It underpins many standard library utilities and remains a key technique for building generic and type-safe abstractions.

1.1.3 Type Sequences

Type sequences provide a way to represent a sequence of compile-time constant values, typically used for parameter pack expansion and index-based operations. The standard utility `std::integer_sequence` (introduced in C++14) is the most common example.

Key Idea `std::integer_sequence<T, N...>` holds a pack of compile-time integers of type `T`. It is often combined with `std::index_sequence` to generate index-based expansions for variadic templates.

```cpp
#include <tuple>
#include <utility>
#include <iostream>

template <typename Tuple, std::size_t... I>
void print_tuple_impl(const Tuple& t, std::index_sequence<I...>) {
    ((std::cout << std::get<I>(t) << " "), ...); // fold expression (C++17)
}

template <typename... Args>
void print_tuple(const std::tuple<Args...>& t) {
    print_tuple_impl(t, std::index_sequence_for<Args...>{});
}

int main() {
    auto data = std::make_tuple(1, 3.14, "C++");
    print_tuple(data); // Output: 1 3.14 C++
}
```

Example: Expanding a Tuple

Here, `std::index_sequence_for<Args...>` generates a sequence of indices `[0, 1, 2]` for the tuple elements, enabling compile-time iteration.

Modern Usage

- `std::integer_sequence` and `std::index_sequence` remove the need for manual recursive template expansion.

- With C++17 fold expressions, type sequences became even more powerful and concise.
- They are widely used in template libraries, serialization frameworks, and tuple utilities.

Importance Type sequences are essential for **compile-time iteration** and **variadic template expansion**, making them a core building block of advanced metaprogramming in modern C++.

1.2 Metaprogramming with `constexpr`

1.2.1 Definition

:

`constexpr` was introduced in C++11 to allow certain functions and variables to be evaluated at compile-time. This makes it possible to compute values, validate logic, and generate constants before the program runs, reducing runtime overhead and enabling safer code.

Key Points

- `constexpr` **Functions**: Must be defined with a single return expression in C++11, later relaxed in C++14 and beyond to support more complex bodies (loops, conditionals).
- **Compile-Time Guarantees**: When arguments are constant expressions, the result is computed during compilation. Otherwise, the function may still run at runtime.
- **Usage**: Often used in numeric computations, array sizes, hashing, and code validation.
- **Evolution**: C++20 extended `constexpr` further by permitting dynamic allocation, exceptions, and more flexible constructs.

Examples

Example 1: Compile-Time Factorial

```cpp
#include <iostream>

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

int main() {
    constexpr unsigned value = factorial(5); // Computed at compile-time
    std::cout << value << '\n'; // Prints 120
}
```

Example 2: `constexpr` with Arrays

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

constexpr int square(int x) { return x * x; }

int main() {
    constexpr std::array<int, square(3)> arr{}; // Array of size 9
    std::cout << arr.size() << '\n';
}
```

Example 3: `constexpr` Validation

```cpp
#include <cassert>

constexpr bool is_power_of_two(unsigned n) {
    return n && !(n & (n - 1));
}

int main() {
    static_assert(is_power_of_two(16), "Must be a power of two");
    assert(is_power_of_two(32)); // Runtime check if needed
}
```

Summary

`constexpr` provides a practical bridge between runtime and compile-time programming. It enables metaprogramming techniques that are both efficient and maintainable, reducing runtime computation and enhancing type and value safety.

1.2.2 Advantages

:

Enhances performance, reduces code size, and provides compile-time computation capabilities.

`constexpr` was introduced in **C++11** and has been significantly extended in later standards. It allows developers to execute computations at compile-time rather than at runtime, producing faster and more optimized programs.

Key Advantages:

1. **Performance Optimization**
   By moving computations from runtime to compile-time, `constexpr` eliminates redundant work during program execution. For example, mathematical constants or lookup tables can be generated once at compile-time.

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

int main() {
    constexpr int val = factorial(5); // Computed at compile-time
    static_assert(val == 120, "Compile-time check");
}
```

2. **Reduced Code Size**
   Since values are computed during compilation, the compiler can directly embed results into the binary, reducing runtime logic and overall code size.

3. **Safer Code with Compile-Time Validation**
   `constexpr` functions integrate with `static_assert`, enabling developers to catch logical errors early in compilation rather than at runtime.

```
constexpr int square(int x) { return x * x; }

static_assert(square(10) == 100, "Validation at compile-time");
```

4. **Improved Template Metaprogramming**
   Previously, template metaprogramming relied on recursive templates, which were often hard to read and maintain. With `constexpr`, many computations can be expressed in a clear functional style, making metaprogramming easier and more accessible.

```
template<int N>
struct Fib {
    static constexpr int value = Fib<N-1>::value + Fib<N-2>::value;
};

template<> struct Fib<0> { static constexpr int value = 0; };
template<> struct Fib<1> { static constexpr int value = 1; };

// Modern approach with constexpr:
constexpr int fib(int n) {
    return (n <= 1) ? n : fib(n - 1) + fib(n - 2);
}
```

Evolution Beyond C++11

- **C++14**: Relaxed `constexpr` restrictions, allowing local variables and loops.
- **C++17**: Expanded support for more constructs, including `if` and `switch`.
- **C++20**: Enabled `constexpr` in dynamic contexts like containers and algorithms.
- **C++23 and Beyond**: Further integration with compile-time reflection and advanced metaprogramming utilities.

1.2.3 Practical Examples

:

The introduction of `constexpr` in C++11 enabled developers to evaluate functions and expressions at compile time, bringing metaprogramming closer to the language's core. With later standards like C++14, C++17, and beyond, `constexpr` functions became more flexible and expressive, allowing complex logic to be executed during compilation.

Example 1: Factorial Calculation

```
#include <iostream>

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

```
}

int main() {
    constexpr auto val = factorial(10); // evaluated at compile time
    std::cout << val << "\n";           // prints 3628800
}
```

Example 2: Fibonacci Sequence

```
#include <iostream>

constexpr unsigned long long fibonacci(unsigned int n) {
    return (n <= 1) ? n : (fibonacci(n - 1) + fibonacci(n - 2));
}

int main() {
    constexpr auto val = fibonacci(20); // evaluated at compile time
    std::cout << val << "\n";           // prints 6765
}
```

Key Points

- **Compile-Time Guarantees**: Ensures correctness and prevents runtime overhead.
- **Performance**: Expensive computations are resolved before the program runs.
- **Maintainability**: Code is simpler compared to template

metaprogramming solutions used before C++11.

1.3 Advanced Libraries

1.3.1 Boost.MP11

:

Boost.MP11 is a modern C++11 metaprogramming library that simplifies template metaprogramming by replacing older, more complex constructs with intuitive variadic template utilities. Unlike traditional template metaprogramming, which often relies on recursive pattern matching, MP11 leverages `constexpr`, variadic templates, and type pack manipulations for improved readability and performance.

Key Features:

- Works with C++11 and later, but scales well with C++14, C++17, and beyond.
- Provides utilities to manipulate type lists, perform compile-time algorithms, and simplify SFINAE-heavy code.
- Replaces older Boost.MPL and Boost.Hana for many use cases.

Example – Filtering a Type List:

```cpp
#include <boost/mp11.hpp>
#include <type_traits>
#include <iostream>

using namespace boost::mp11;

// Define a type list
using types = mp_list<int, double, char, float>;

// Filter out only integral types
using only_integrals = mp_copy_if<types, std::is_integral>;

// Verify result
static_assert(std::is_same<only_integrals, mp_list<int, char>>::value, "Check
↪  failed");

int main() {
    std::cout << "Filtered type list contains int and char." << std::endl;
}
```

Why It Matters:

Boost.MP11 significantly reduces the boilerplate of template metaprogramming and makes code more expressive. For developers transitioning from C++11 to modern standards, it provides an approachable way to apply metaprogramming without diving into complex recursive templates.

1.3.2 Examples

Boost.MP11 provides high-level utilities for compile-time type list manipulation. Two commonly used tools are `mp_list` for defining type lists and `mp_transform` for applying transformations to each type in the list.

Example 1: Defining and Transforming a Type List

```
#include <boost/mp11.hpp>
#include <type_traits>
#include <iostream>

using namespace boost::mp11;

// Original type list
using types = mp_list<int, double, char>;

// Transform: add const qualifier to all types
using const_types = mp_transform<std::add_const, types>;

// Verify transformation
static_assert(std::is_same<const_types, mp_list<const int, const double, const
↪  char>>::value,
              "Transformation failed");

int main() {
    std::cout << "All types are now const-qualified." << std::endl;
}
```

Example 2: Filtering Types from a List

```
// Filter only floating-point types
using float_types = mp_copy_if<types, std::is_floating_point>;
static_assert(std::is_same<float_types, mp_list<double>>::value, "Filtering failed");
```

Key Points:

- `mp_list` is a lightweight container for compile-time types.
- `mp_transform` applies type modifications consistently across a list.
- `mp_copy_if` allows conditional selection of types based on traits.
- These utilities replace complex recursive templates, making code more readable and maintainable.

Importance:
Using Boost.MP11 simplifies compile-time type manipulation, enabling expressive metaprogramming while reducing boilerplate and improving compile-time performance.

Chapter 2

Future Expectations in C++26

2.1 Reflection

2.1.1 Definition

Reflection in C++26 refers to the ability to **access type metadata**—such as member names, types, and functions—**at compile-time or runtime**. This mechanism enables developers to write code that can inspect, adapt, or generate behavior based on the structure of types, without manually maintaining repetitive boilerplate.

Key Concepts

- **Compile-Time Reflection**: Access type information during compilation for metaprogramming and code generation.
- **Runtime Reflection**: Access type information at runtime for tasks like serialization, logging, or dynamic dispatch.
- **Integration with Templates and `constexpr`**: Enables safe and efficient compile-time operations on types.

Example (Conceptual, C++26 Proposal Syntax)

```
struct Person {
    std::string name;
    int age;
};

// Obtain compile-time metadata
constexpr auto members = reflect<Person>();

static_assert(members.size() == 2, "Person has exactly two members");

// Iterate over members
for_each_member<Person>([](auto member) {
    std::cout << member.name() << " : " << member.type_name() << '\n';
});
```

Benefits

- Automates repetitive tasks such as serialization, comparison, and validation.
- Reduces human error by ensuring code adapts automatically to changes in type definitions.
- Strengthens metaprogramming by providing precise type-aware operations at compile-time.

Importance
Reflection in C++26 bridges the gap between **templates, metaprogramming, and runtime adaptability**, making modern C++ more expressive, safe, and maintainable.

2.1.2 Advantages

Generate custom code, improve documentation, and simplify maintenance
Reflection in C++26 provides powerful benefits for metaprogramming by exposing type metadata at compile-time and runtime.

Key Advantages

1. **Custom Code Generation**
    - Reflection allows automatic generation of code such as

serialization, comparison operators, or conversion routines based on the structure of types.

- Example (conceptual syntax):

```
struct Person { std::string name; int age; };
constexpr auto members = reflect<Person>();
for_each_member<Person>([](auto member){
    generate_getter(member); // Generates getters automatically
});
```

2. **Improved Documentation**

   - Metadata can be used to extract member names and types, enabling tools to generate consistent API documentation automatically.

3. **Simplified Maintenance**

   - Changes in a type definition propagate automatically to all reflective operations, reducing errors from manual updates and ensuring consistency.

Importance

Reflection reduces repetitive boilerplate, enforces type safety, and enables maintainable and adaptive code. It empowers developers to write more expressive C++ while keeping code concise and robust.

2.1.3 Proposals

P2996 for C++26

The C++26 reflection mechanism is being shaped primarily by **proposal P2996**, which aims to provide standardized compile-time and runtime reflection capabilities. The proposal introduces ways to inspect, query, and manipulate type metadata efficiently.

Key Features of P2996

- **Compile-Time Introspection**: Access type members, functions, and base classes at compile-time for metaprogramming and code generation.
- **Runtime Introspection**: Retrieve type information at runtime for tasks like serialization, logging, or dynamic dispatch.
- **Integration with Templates and `constexpr`**: Ensures type-safe operations and allows reflection results to participate in compile-time computations.

Conceptual Example (C++26 P2996 Style)

```
struct Person { std::string name; int age; };

// Reflect the structure
constexpr auto members = reflect<Person>();
```

```
static_assert(members.size() == 2, "Person has two members");

// Generate code automatically
for_each_member<Person>([](auto member) {
    std::cout << member.name() << " : " << member.type_name() << '\n';
});
```

Impact

- Reduces boilerplate code for serialization, comparison, and validation.
- Enhances maintainability: changes to type structures automatically propagate to reflective operations.
- Makes C++26 metaprogramming more expressive and safer.

2.2 Pattern Matching

2.2.1 Definition

Pattern matching in C++26 is a mechanism to **decompose and inspect data based on its structure**, enabling conditional execution or extraction of values directly from types or objects. It complements reflection and metaprogramming by allowing expressive, type-safe handling of structured data.

Key Concepts

- Matches data against a **pattern** rather than a single value.
- Works with **structs, tuples, and variant types**.
- Often integrated with `constexpr` and reflection for compile-time evaluation.

Example (Conceptual Syntax)

```
struct Point { int x; int y; };

Point p{10, 20};

// Pattern matching pseudo-code (C++26 style)
match(p) {
```

```
    case Point{x, y}:
        std::cout << "x=" << x << ", y=" << y << '\n';
}
```

Benefits

- Reduces boilerplate when decomposing complex structures.
- Enhances readability and maintainability of code by clearly expressing intended operations on structured data.
- Integrates naturally with reflection to automate data-driven operations.

Importance

Pattern matching allows modern C++ to adopt **declarative and expressive coding styles** familiar from functional programming, while remaining type-safe and efficient.

2.2.2 Advantages

Simplifies code, improves clarity, and reduces errors
Pattern matching in C++26 provides a clean and expressive way to decompose data structures and act on their contents.

Key Advantages

1. **Simplifies Code**

    - Reduces boilerplate code needed for manually extracting values from structs, tuples, or variants.
    - Example (conceptual C++26 syntax):

```
struct Point { int x; int y; };
Point p{5, 10};

match(p) {
    case Point{x, y}:
        std::cout << "x=" << x << ", y=" << y << '\n';
}
```

2. **Improves Clarity**

    - Clearly expresses intent by matching against structure patterns rather than using multiple conditional statements.

3. **Reduces Errors**

    - Eliminates common mistakes from manual decomposition, such as wrong index access in tuples or incorrect type casts.

Importance

Pattern matching, together with reflection, allows C++26 to support **more declarative, readable, and type-safe metaprogramming**, making programs easier to maintain and understand.

2.2.3 Proposals

Pattern-matching support expected in C++26
C++26 is expected to introduce **native pattern-matching support**, enabling developers to destructure and inspect objects, tuples, and variants based on their structure. This feature complements reflection and metaprogramming by allowing code to react to the shape of data rather than just its value.

Key Features (Expected)

- Match structs, tuples, enums, and variant types directly.
- Combine with `constexpr` and reflection for compile-time evaluation.
- Reduce boilerplate for decomposing nested or complex data structures.

Example (Conceptual Syntax)

```cpp
struct Point { int x; int y; };
Point p{10, 20};

// Expected C++26 pattern matching
match(p) {
    case Point{x, y}:
        std::cout << "x=" << x << ", y=" << y << '\n';
}
```

Benefits

- Simplifies data decomposition and reduces repetitive code.
- Improves readability and maintainability.
- Provides type-safe operations that prevent common errors.

Importance

Pattern-matching proposals in C++26 will allow **declarative and expressive handling of structured data**, bringing modern, functional-style metaprogramming features to C++.

2.3 Template Loops

2.3.1 Definition

Template loops in C++26 are a proposed mechanism to **iterate over sets of values or types at compile-time** within templates. This allows repetitive computations or transformations to be expressed cleanly, without relying on recursive template instantiations.

Key Concepts

- Iteration occurs entirely at **compile-time**.
- Can be used to generate code for **arrays, tuples, type lists, or constants**.
- Reduces complexity compared to traditional recursive templates.

Example (Conceptual Syntax)

```
#include <iostream>
#include <tuple>

template<typename... Ts>
constexpr void print_tuple(const std::tuple<Ts...>& t) {
    template_loop<sizeof...(Ts)>([&](auto i) {
        std::cout << std::get<i>(t) << " ";
    });
```

```
    std::cout << "\n";
}

int main() {
    constexpr auto t = std::make_tuple(1, 2, 3, 4);
    print_tuple(t); // Expected output: 1 2 3 4
}
```

Benefits

- Automates repetitive compile-time tasks.
- Improves readability over recursive template solutions.
- Works well with `constexpr`, type sequences, and pattern matching for advanced metaprogramming.

Importance

Template loops represent a **major simplification for compile-time computations** in C++26, making template metaprogramming more intuitive, maintainable, and efficient.

2.3.2 Advantages

Reduces code repetition, improves performance, and simplifies maintenance

Template loops in C++26 enable **compile-time iteration over types or values**, replacing verbose recursive templates and repetitive code.

Key Advantages

1. **Reduces Code Repetition**

    - Replaces manual unrolling of templates or repeated function overloads.

```
template_loop<5>([](auto i){
    std::cout << i << " ";
}); // Output: 0 1 2 3 4
```

2. **Improves Performance**

    - All iterations are executed at compile-time, producing optimized code with no runtime overhead.

3. **Simplifies Maintenance**

    - Updates to logic or value sets automatically propagate, reducing human error in repetitive template code.

Practical Use

- Generating sequences of constants or indices.
- Expanding tuples or parameter packs.
- Integrating with reflection or pattern matching for adaptive compile-time operations.

Importance
Template loops make **advanced metaprogramming in C++26 cleaner, safer, and faster**, bridging the gap between expressive templates and maintainable code.

2.3.3 Proposals

Template loop support expected in C++26
C++26 is expected to introduce **native template loops**, allowing developers to iterate over **types, values, or indices at compile-time** without relying on recursive templates. This feature simplifies common metaprogramming patterns and makes compile-time computation more readable and maintainable.

Key Features (Expected)

- Iteration over type packs or value sequences at compile-time.
- Integration with `constexpr`, reflection, and pattern matching.

- Eliminates boilerplate and complex recursion in template code.

Conceptual Example (C++26 Style)

```
template_loop<5>([](auto i){
    std::cout << i << " "; // Expected output: 0 1 2 3 4
});
```

Benefits

- Reduces repetitive code in template metaprogramming.
- Enables compile-time code generation for sequences, tuples, and arrays.
- Works seamlessly with other C++26 metaprogramming enhancements.

Importance

Template loops in C++26 will make compile-time iteration **more intuitive, concise, and efficient**, bridging the gap between expressive metaprogramming and readable, maintainable code.

Chapter 3

Practical Examples

3.1 Computing Values with `constexpr`

Example: Calculating values like `factorial` or `fibonacci` at **compile-time** using `constexpr`.
`constexpr` functions allow computations to be performed during compilation, reducing runtime overhead and enabling safer, optimized code.

Factorial Example

```
#include <iostream>

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

```
}

int main() {
    constexpr auto val = factorial(10); // Computed at compile-time
    std::cout << val << "\n";           // Prints 3628800
}
```

Fibonacci Example

```
#include <iostream>

constexpr unsigned long long fibonacci(unsigned n) {
    return (n <= 1) ? n : fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
    constexpr auto val = fibonacci(20); // Computed at compile-time
    std::cout << val << "\n";           // Prints 6765
}
```

Key Points

- `constexpr` ensures **compile-time evaluation** when inputs are constant expressions.
- Improves **performance** by removing runtime computation.
- Reduces **errors** by allowing `static_assert` checks on computed values.

Importance

Using `constexpr` for calculations like factorial or fibonacci demonstrates the **power and simplicity of modern C++ metaprogramming**, making code both efficient and maintainable.

3.2 Creating a Utility Library with Boost.MP11

Example: Building a library for **manipulating type lists at compile-time** using Boost.MP11.
Boost.MP11 simplifies compile-time type manipulations, making metaprogramming more readable and maintainable compared to traditional recursive templates.

Defining and Transforming Type Lists

```
#include <boost/mp11.hpp>
#include <type_traits>
#include <iostream>

using namespace boost::mp11;

// Define a type list
using types = mp_list<int, double, char>;

// Transform: add const qualifier to all types
using const_types = mp_transform<std::add_const, types>;

// Verify transformation
static_assert(std::is_same<const_types, mp_list<const int, const double, const
↪  char>>::value,
              "Transformation failed");

int main() {
```

```
    std::cout << "All types are const-qualified.\n";
}
```

Filtering Types from a List

```
// Filter only floating-point types
using float_types = mp_copy_if<types, std::is_floating_point>;
static_assert(std::is_same<float_types, mp_list<double>>::value, "Filtering failed");
```

Key Points

- `mp_list` defines a compile-time collection of types.
- `mp_transform` applies a type trait or transformation to all elements.
- `mp_copy_if` filters types based on compile-time conditions.

Importance

Using Boost.MP11 to create a **utility library** demonstrates how modern C++ metaprogramming can **automate type manipulations**, reduce boilerplate, and improve maintainability, all at compile-time.

3.3 Using Reflection to Generate Custom Code

Example: Accessing type information to generate **custom code based on metadata**.
Reflection in modern C++ (C++26 proposals) allows programs to inspect type structures at compile-time or runtime and **automatically generate code**, reducing boilerplate and ensuring correctness.

Conceptual Example (C++26 Style)

```
struct Person {
    std::string name;
    int age;
};

// Reflect type metadata
constexpr auto members = reflect<Person>();

// Generate code automatically
for_each_member<Person>([](auto member) {
    std::cout << "Generating getter for: " << member.name() << '\n';
    // Conceptually: generate getter function based on member metadata
});
```

Key Points

- Access **member names, types, and functions** at compile-time.

- Automates repetitive tasks like **getters, setters, serialization, and logging**.
- Reduces human error by ensuring generated code adapts to type changes.

Importance

Using reflection to generate custom code illustrates **the power of modern metaprogramming**: programs become **safer, more maintainable, and adaptive** to changes, without additional runtime overhead.

Conclusion

3.4 Importance of Metaprogramming in C++

Metaprogramming in modern C++ is a **powerful tool** that enables developers to:

1. **Improve Performance**

    - Perform computations at compile-time using `constexpr` or template metaprogramming.

```
constexpr int factorial(int n) { return n <= 1 ? 1 : n * factorial(n - 1); }
constexpr auto val = factorial(5); // Computed at compile-time
```

2. **Enhance Type-Checking**

- Enforce compile-time constraints and type safety using concepts, SFINAE, and type traits.

```
template<typename T>
requires std::is_integral_v<T>
T add(T a, T b) { return a + b; }
```

3. **Generate Custom Code**

- Use reflection, pattern matching, and libraries like Boost.MP11 to automatically create code for getters, serialization, or type transformations.

```
struct Person { std::string name; int age; };
for_each_member<Person>([](auto m){ generate_getter(m); });
```

Conclusion

Metaprogramming allows **safer, faster, and more maintainable C++ code**, reducing runtime overhead, eliminating repetitive boilerplate, and enabling advanced compile-time computation. Its evolution from C++11 to C++26 makes modern C++ increasingly expressive, efficient, and suitable for complex software systems.

3.5 Future in C++26

C++26 is expected to **significantly expand metaprogramming capabilities** with features like **reflection, pattern matching, and template loops**. These features will make compile-time computation more expressive, readable, and efficient.

3.6 Key Expected Features

1. **Reflection**

   - Access type metadata at compile-time and runtime.
   - Automate code generation for getters, setters, serialization, and logging.

```
struct Person { std::string name; int age; };
for_each_member<Person>([](auto m){ generate_getter(m); });
```

2. **Pattern Matching**

   - Decompose and inspect data structures based on their shape.
   - Simplifies conditional logic and reduces boilerplate.

```
match(p) { case Point{x, y}: std::cout << x << y; }
```

3. **Template Loops**

   - Iterate over type packs or value sequences at compile-time.
   - Replace recursive templates with clean, maintainable loops.

```
template_loop<5>([](auto i){ std::cout << i << " "; });
```

Impact

- Reduces repetitive code, improves maintainability, and enforces type safety.
- Enables **declarative and adaptive compile-time programming**, making modern C++ more expressive and efficient.

Appendices

The appendices provide **supplementary material** and practical references to support the main content of this booklet on modern C++ metaprogramming.

Appendix A: Quick Reference – Type Traits

- `std::is_integral<T>`: Checks if `T` is an integral type.
- `std::is_same<T, U>`: Checks if `T` and `U` are the same type.
- `std::remove_const<T>`: Removes `const` qualifier from type.

Appendix B: Quick Reference – SFINAE & Concepts

- **SFINAE**: Enables/disables templates based on type properties.

```
template<typename T>
std::enable_if_t<std::is_integral_v<T>, T> add(T a, T b) { return a + b; }
```

- **Concepts (C++20)**: Provides clearer syntax for constraints.

```
template<std::integral T>
T add(T a, T b) { return a + b; }
```

Appendix C: Quick Reference – `constexpr`

- Compile-time computations for functions and variables.
- Example: Factorial

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

Appendix D: Boost.MP11 Essentials

- `mp_list<Ts...>`: Define type lists.
- `mp_transform<F, L>`: Apply transformation to all types in a list.
- `mp_copy_if<L, C>`: Filter types based on compile-time conditions.

Appendix E: C++26 Proposed Features (Conceptual)

- **Reflection**: `reflect<T>()` for compile-time type inspection.
- **Pattern Matching**: `match(obj) { case Type{x, y}: ... }`.
- **Template Loops**: `template_loop<N>([](auto i){ ... });`

Purpose of Appendices

- Provide **quick lookup** for frequently used metaprogramming tools and syntax.
- Serve as **practical examples** for building compile-time utilities and advanced C++ code.
- Support **efficient referencing** while working with modern C++ and upcoming C++26 features.

References

This booklet draws upon **modern C++ metaprogramming resources**, standards, and library documentation to provide accurate and practical guidance.

1. **C++ Standards and Proposals**

    - C++11, C++14, C++17, C++20: Introduced `constexpr`, type traits, template metaprogramming improvements.
    - C++26 (Upcoming): Proposals P2996 (Reflection), pattern matching, template loops.

2. **Core C++ Libraries**

    - <`type_traits`>: For compile-time type inspection (`std::is_integral`, `std::is_same`).
    - <`utility`>: `std::integer_sequence`, `std::index_sequence` for type sequences.

3. **Advanced Libraries**

    - **Boost.MP11**: High-level metaprogramming library for manipulating type lists (`mp_list`, `mp_transform`, `mp_copy_if`).

4. **Online Resources & Documentation**

    - cppreference.com: Comprehensive reference for C++ standard library and features.
    - ISO C++ Committee papers: Technical papers and proposals (e.g., P2996 for reflection).

5. **Books and Articles**

    - "C++ Templates: The Complete Guide" – David Vandevoorde, Nicolai M. Josuttis
    - "Modern C++ Design" – Andrei Alexandrescu
    - Herb Sutter's articles on metaprogramming and Modern C++ techniques

Note: All examples in this booklet are adapted to **modern C++ syntax** and conceptual C++26 proposals to illustrate upcoming metaprogramming capabilities.