C++26 Reflection for JSON Serialization

A Practical Journey

Daniel Lemire and Francisco Geiman Thiesen

CppCon 2025

The Problem: Every Developer's JSON Nightmare

Imagine you're building a game server that needs to persist player data.

You start simple:

```cpp
struct Player {
    std::string username;
    int level;
    double health;
    std::vector<std::string> inventory;
};
```

The Traditional Approach: Manual Serialization

Without reflection, you write this tedious code:

```cpp
// Serialization - converting Player to JSON
std::string serialize_player(const Player& p) {
    std::stringstream ss;
    ss << "{";
    ss << "\"username\":\"" << escape_json(p.username) << "\",";
    ss << "\"level\":" << p.level << ",";
    ss << "\"health\":" << p.health << ",";
    ss << "\"inventory\":[";
    for (size_t i = 0; i < p.inventory.size(); ++i) {
        if (i > 0) ss << ",";
        ss << "\"" << escape_json(p.inventory[i]) << "\"";
    }
    ss << "]";
    ss << "}";
    return ss.str();
}
```

Manual Deserialization

```
// Deserialization - converting JSON back to Player
simdjson::error_code deserialize_player(simdjson::ondemand::value& val, Player& p) {
    simdjson::ondemand::object obj;
    SIMDJSON_TRY(val.get_object().get(obj));

    SIMDJSON_TRY(obj["username"].get_string().get(p.username));
    SIMDJSON_TRY(obj["level"].get_int64().get(p.level));
    SIMDJSON_TRY(obj["health"].get_double().get(p.health));

    simdjson::ondemand::array arr;
    SIMDJSON_TRY(obj["inventory"].get_array().get(arr));
    for (auto item : arr) {
        std::string_view sv;
        SIMDJSON_TRY(item.get_string().get(sv));
        p.inventory.emplace_back(sv);
    }

    return simdjson::SUCCESS;
}
```

The Pain Points

This manual approach has several problems:

1. **Repetition**: Every field needs to be handled twice (serialize + deserialize)
2. **Maintenance Nightmare**: Add a new field? Update both functions!
3. **Error-Prone**: Typos in field names, forgotten fields, type mismatches
4. **Boilerplate Explosion**: 30+ lines for a simple 4-field struct

When Your Game Grows...

```cpp
struct Equipment {
    std::string name;
    int damage;
    int durability;
};

struct Achievement {
    std::string title;
    std::string description;
    bool unlocked;
    std::chrono::system_clock::time_point unlock_time;
};

struct Player {
    std::string username;
    int level;
    double health;
    std::vector<std::string> inventory;
    std::map<std::string, Equipment> equipped;     // New!
    std::vector<Achievement> achievements;         // New!
    std::optional<std::string> guild_name;        // New!
};
```

The Solution: C++26 Static Reflection

With C++26 reflection and simdjson, **all that boilerplate disappears**:

```cpp
// Just define your struct - no extra code needed!
struct Player {
    std::string username;
    int level;
    double health;
    std::vector<std::string> inventory;
    std::map<std::string, Equipment> equipped;
    std::vector<Achievement> achievements;
    std::optional<std::string> guild_name;
};
```

Automatic Serialization & Deserialization

```cpp
// Serialization - one line!
void save_player(const Player& p) {
    std::string json = simdjson::to_json(p);  // That's it!
    // Save json to file...
}

// Deserialization - one line!
Player load_player(const std::string& json_str) {
    return simdjson::from<Player>(json_str);  // That's it!
}
```

- No manual field mapping
- No maintenance burden
- Handles nested structures automatically

Comparison with Other Languages

```python
# Python
import json
json_str = json.dumps(player.__dict__)
player = Player(**json.loads(json_str))
```

```rust
// Rust with serde
let json_str = serde_json::to_string(&player)?;
let player: Player = serde_json::from_str(&json_str)?;
```

```cpp
// C++26 with simdjson - just as clean!
std::string json_str = simdjson::to_json(player);
Player player = simdjson::from<Player>(json_str);
```

How Does It Work?

The Key Insight: Compile-Time Code Generation

A common question: **"How can compile-time reflection handle runtime JSON data?"**

The answer: Reflection operates on **types and structure**, not runtime values.

It generates regular C++ code at compile time that handles your runtime data.

What Happens Behind the Scenes

```cpp
// What you write:
Player p = simdjson::from<Player>(runtime_json_string);

// What reflection generates at COMPILE TIME (conceptually):
Player deserialize_Player(const json& j) {
    Player p;
    p.username = j["username"].get<std::string>();
    p.level = j["level"].get<int>();
    p.health = j["health"].get<double>();
    p.inventory = j["inventory"].get<std::vector<std::string>>();
    // ... etc for all members
    return p;
}
```

The Actual Reflection Magic

```
template <typename T>
  requires(std::is_class_v<T>)  // For user-defined types
error_code deserialize(auto& json_value, T& out) {
    simdjson::ondemand::object obj;
    SIMDJSON_TRY(json_value.get_object().get(obj));

    // This [:expand:] happens at COMPILE TIME
    // It literally generates code for each member
    [:expand(std::meta::nonstatic_data_members_of(^^T)):] >> [&]<auto member>() {
        // These are compile-time constants
        constexpr std::string_view field_name = std::meta::identifier_of(member);
        constexpr auto member_type = std::meta::type_of(member);

        // This generates code for each member
        auto err = obj[field_name].get(out.[:member:]);
        if (err && err != simdjson::NO_SUCH_FIELD) {
            return err;
        }
    };

    return simdjson::SUCCESS;
}
```

The [:expand:] Statement

The `[:expand:]` statement is the key:

- It's like a **compile-time for-loop**
- Generates code for each struct member
- By the time your program runs, all reflection has been "expanded" into normal C++ code

This means:

- **Zero runtime overhead**
- **Full optimization opportunities**
- **Type safety at compile time**

Compile-Time vs Runtime: What Happens When

```cpp
struct Player {
    std::string username;    // ← Compile-time: reflection sees this
    int level;               // ← Compile-time: reflection sees this
    double health;           // ← Compile-time: reflection sees this
};

// COMPILE TIME: Reflection reads Player's structure and generates:
// - Code to read "username" as string
// - Code to read "level" as int
// - Code to read "health" as double

// RUNTIME: The generated code processes actual JSON data
std::string json = R"({"username":"Alice","level":42,"health":100.0})";
Player p = simdjson::from<Player>(json);
// Runtime values flow through compile-time generated code
```

Compile-Time Safety: Catching Errors Before They Run

```cpp
// ❌ COMPILE ERROR: Type mismatch detected
struct BadPlayer {
    int username;  // Oops, should be string!
};
// simdjson::from<BadPlayer>(json) won't compile if JSON has string

// ❌ COMPILE ERROR: Non-serializable type
struct InvalidType {
    std::thread t;  // Threads can't be serialized!
};
// simdjson::to_json(InvalidType{}) fails at compile time

// ✅ COMPILE SUCCESS: All types are serializable
struct GoodType {
    std::vector<int> numbers;
    std::map<std::string, double> scores;
    std::optional<std::string> nickname;
}
```

Zero Overhead: Why It's Fast

Since reflection happens at compile time, there's no runtime penalty:

1. **No runtime type inspection** - everything is known at compile time
2. **No string comparisons for field names** - they become compile-time constants
3. **Optimal code generation** - the compiler sees the full picture
4. **Inline everything** - generated code can be fully optimized

The generated code is often **faster than hand-written code** because:

- It's consistently optimized
- No human errors or inefficiencies
- Leverages simdjson's SIMD parsing throughout

Performance: The Best Part

You might think "automatic = slow", but with simdjson + reflection:

- **Compile-time code generation**: No runtime overhead from reflection
- **SIMD-accelerated parsing**: simdjson uses CPU vector instructions
- **Zero allocation**: String views and in-place parsing
- **Throughput**: ~2-4 GB/s on modern hardware

The generated code is often *faster* than hand-written code!

Real-World Benefits

Before Reflection (Our Game Server example)

- 1000+ lines of serialization code
- Prone to bugs due to serialization mismatching
- Adding new features can imply making tedious changes to boilerplate serialization code

After Reflection

- **0 lines** of serialization code
- **0 serialization bugs** (if it compiles, it works!)
- New features can be added much faster

The Bigger Picture

This pattern extends beyond games:

- **REST APIs**: Automatic request/response serialization
- **Configuration Files**: Type-safe config loading
- **Message Queues**: Serialize/deserialize messages
- **Databases**: Object-relational mapping
- **RPC Systems**: Automatic protocol generation

With C++26 reflection, C++ finally catches up to languages like Rust (serde), Go (encoding/json), and C# (System.Text.Json) in terms of ease of use, but with **better performance** thanks to simdjson's SIMD optimizations.

Try It Yourself

```cpp
struct Meeting {
    std::string title;
    std::chrono::system_clock::time_point start_time;
    std::vector<std::string> attendees;
    std::optional<std::string> location;
    bool is_recurring;
};

// Automatically serializable/deserializable!
std::string json = simdjson::to_json(Meeting{
    .title = "CppCon Planning",
    .start_time = std::chrono::system_clock::now(),
    .attendees = {"Alice", "Bob", "Charlie"},
    .location = "Denver",
    .is_recurring = true
});

Meeting m = simdjson::from<Meeting>(json);
```

Round-Trip Any Data Structure

```
struct TodoItem {
    std::string task;
    bool completed;
    std::optional<std::string> due_date;
};

struct TodoList {
    std::string owner;
    std::vector<TodoItem> items;
    std::map<std::string, int> tags;  // tag -> count
};

// Serialize complex nested structures
TodoList my_todos = { /* ... */ };
std::string json = simdjson::to_json(my_todos);

// Deserialize back - perfect round-trip
TodoList restored = simdjson::from<TodoList>(json);
assert(my_todos == restored);  // Works if you define operator==
```

The Entire API Surface

Just two functions. Infinite possibilities.

```
simdjson::to_json(object)  // → JSON string
simdjson::from<T>(json)    // → T object
```

That's it.

No macros. No code generation. No external tools.

Just simdjson leveraging C++26 reflection.

Supporting Standard Containers

Through concepts and template specialization, we support:

- `std::vector<T>`, `std::array<T, N>`
- `std::map<K, V>`, `std::unordered_map<K, V>`
- `std::optional<T>`
- `std::variant<Types...>`
- And many more...

All work seamlessly with reflection!

Conclusion

C++26 Reflection + simdjson =

- ✅ Zero boilerplate
- ✅ Compile-time safety
- ✅ Blazing fast performance
- ✅ Clean, modern API

Welcome to the future of C++ serialization! 🚀

Questions?

Daniel Lemire and Francisco Geiman Thiesen

GitHub: github.com/simdjson/simdjson

Thank you!