C++26 Reflection for JSON Serialization

A Practical Journey

- Daniel Lemire, *University of Quebec*
- Francisco Geiman Thiesen, *Microsoft*

CppCon 2025

JSON

- Portable, simple
- Douglas Crockford (2001)
- RFC 8259 (December 2017)

JSON

- scalar values
  - strings (controls and quotes must be escaped)
  - numbers (but not `NaN` or `Inf`)
  - `true`, `false`, `null`
- composed values
  - objects (key/value)
  - arrays (list)

```json
{
    "username": "Alice",
    "level": 42,
    "health": 99.5,
    "inventory": ["sword", "shield", "potion"]
}
```

JSON downside?

JSON can be *slow*. E.g., 20 MB/s.

- Much slower than disk or network

Micron shows off world's fastest PCIe 6.0 SSD, hitting 27 GB/s speeds — Astera Labs PCIe 6.0 switch enables impressive sequential reads

News By Sunny Grimm published March 8, 2025

The next-gen of networking and storage is hitting the trade shows

Performance

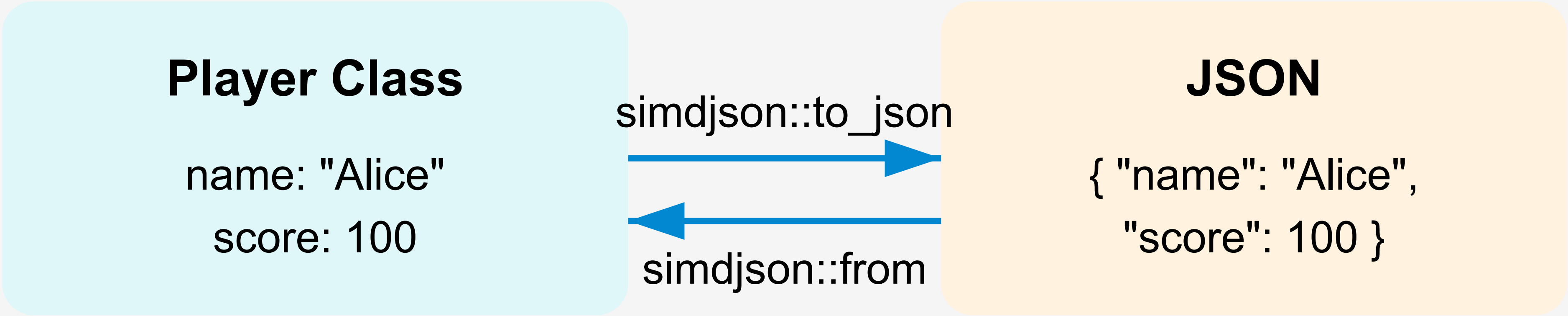
- simdjson was the first library to break the gigabyte per second barrier
  - Parsing Gigabytes of JSON per Second, VLDB Journal 28 (6), 2019
  - On-Demand JSON: A Better Way to Parse Documents? SPE 54 (6), 2024
- JSON for Modern C++ can be $100\times$ slower!

Usage

The simdjson library is found in...

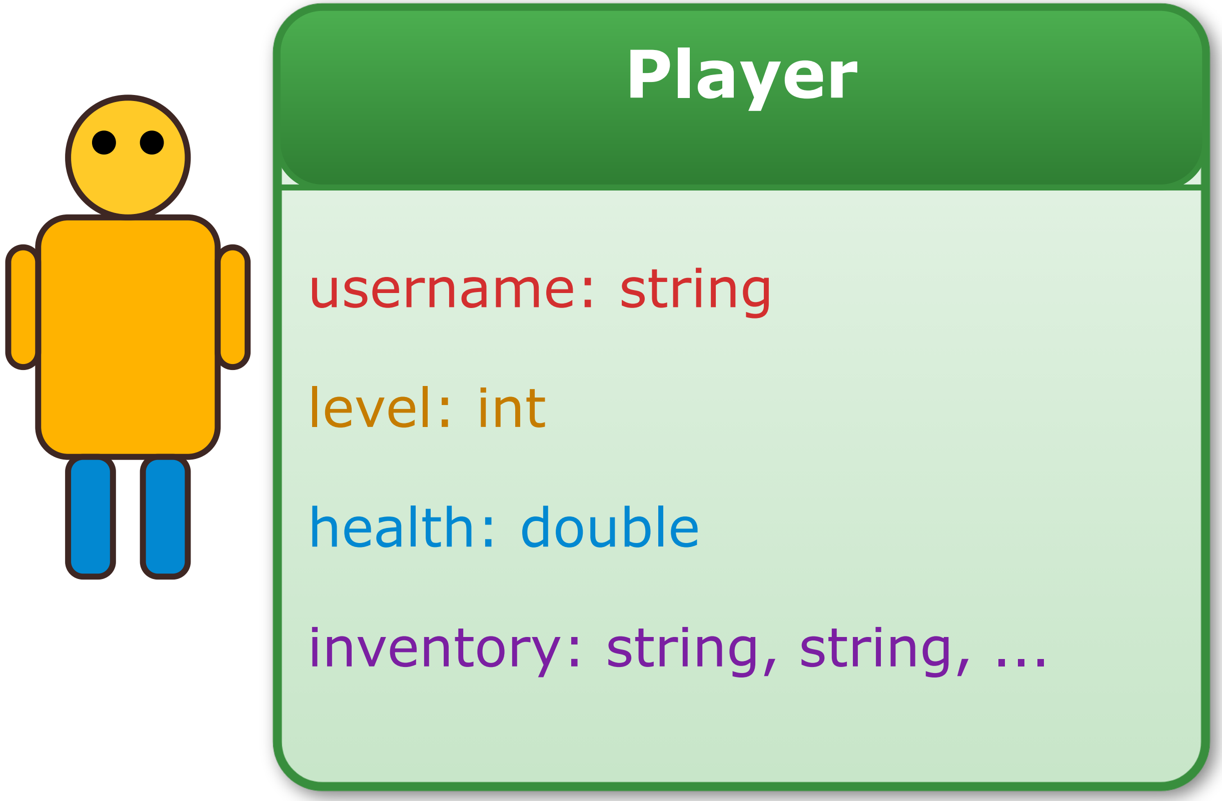
- Node.js
- ClickHouse
- Velox

Automate the serialization/deserialization process.



The Problem

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 may write this tedious code:

```
// Serialization - converting Player to JSON
fmt::format(
        "{{"
        "\"username\":\"{}\","
        "\"level\":{},"
        "\"health\":{},"
        "\"inventory\":{}"
        "}}",
        escape_json(p.username),
        p.level,
        std::isfinite(p.health) ? p.health : -1.0,
        p.inventory| std::views::transform(escape_json)
);
```

With a library (JSON for Modern C++)

Or you might use a library.

```
std::string to_json(Player& p) {
  return nlohmann::json{{"username", p.username},
                        {"level", p.level},
                        {"health", p.health},
                        {"inventory", p.inventory}}
      .dump();
}
```

Manual Deserialization (simdjson)

```
object obj = val.get_object();
p.username = obj["username"].get_string();
p.level = obj["username"].get_int64();
p.health = obj["health"].get_double();
array arr = obj["inventory"].get_array();
for (auto item : arr) {
    p.inventory.emplace_back(item.get_string());
}
```

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
5. **Performance**: You may fall into performance traps

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!
};
```

Suddenly you need to write hundreds of lines of serialization code! 😱

The Solution: C++26 Static Reflection

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

```
// 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

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

Automatic Deserialization

```cpp
// 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
- Performance tuned by the library

Python

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

Python reflection

```python
def inspect_object(obj):
    print(f"Class name: {obj.__class__.__name__}")
    for attr, value in vars(obj).items():
        print(f"  {attr}: {value}")
```

Go

```
jsonData, err := json.MarshalIndent(player, "", "  ")
if err != nil {
        log.Fatalf("Error during serialization: %v", err)
}
var deserializedPlayer Player
err = json.Unmarshal([]byte(jsonStr), &deserializedPlayer)
```

Go reflection

- Runtime reflection only

```go
typ := reflect.TypeOf(obj)
for i := 0; i < typ.NumField(); i++ {
	field := typ.Field(i)
}
```

Java and C#

```
string jsonString = JsonSerializer.Serialize(player, options);
Player deserializedPlayer = JsonSerializer.Deserialize<Player>(jsonInput, options);
```

Java and C# reflection

- Runtime reflection only.

```
Class<?> playerClass = Player.class;
Object playerInstance = playerClass.getDeclaredConstructor().newInstance();
Field nameField = playerClass.getDeclaredField("name");
```

Rust (serde)

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

Rust reflection

- Rust does not have ANY introspection.
- You cannot enumerate the methods of a struct. Either at runtime or at compile-time.
- Rust relies on annotation (serde) followed by re-parsing of the code.

Reflection as accessing the attributes of a struct.

language	runtime reflection	compile-time reflection
C++ 26	👎	✅
Go	✅	👎
Java	✅	👎
C#	✅	👎
Rust	👎	👎

With C++26: simple, maintainable, performant code

```
std::string json_str = simdjson::to_json(player);
Player player = simdjson::from<Player>(json_str);
```

- **AT COMPILE TIME**
- with no extra tooling
- no annotation

How Does It Work?

The Key Insight: Compile-Time Code Generation

"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

```cpp
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

```
// ❌ 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

```
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

```cpp
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!