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!

SIMD

- Stands for Single instruction, multiple data

- Allows us to process 16 (or more) bytes or more with one instruction

- Supported on all modern CPUs (phone, laptop)

Superscalar vs. SIMD execution

processor	year	arithmetic logic units	SIMD units	simdjson
Apple M*	2019	6+	4×128	🥉
Intel Lion Cove	2024	6	4×256	🥈🥈
AMD Zen 5	2024	6	4×512	🥇🥇🥇

simdjson: Parsing design

- First scan identifies the structural characters, start of all strings at about 10 GB/s using SIMD instructions.

- Validates Unicode (UTF-8) at 30 GB/s.

- Rest of parsing relies on index.

- Allows fast skipping.

https://openbenchmarking.org/test/pts/simdjson
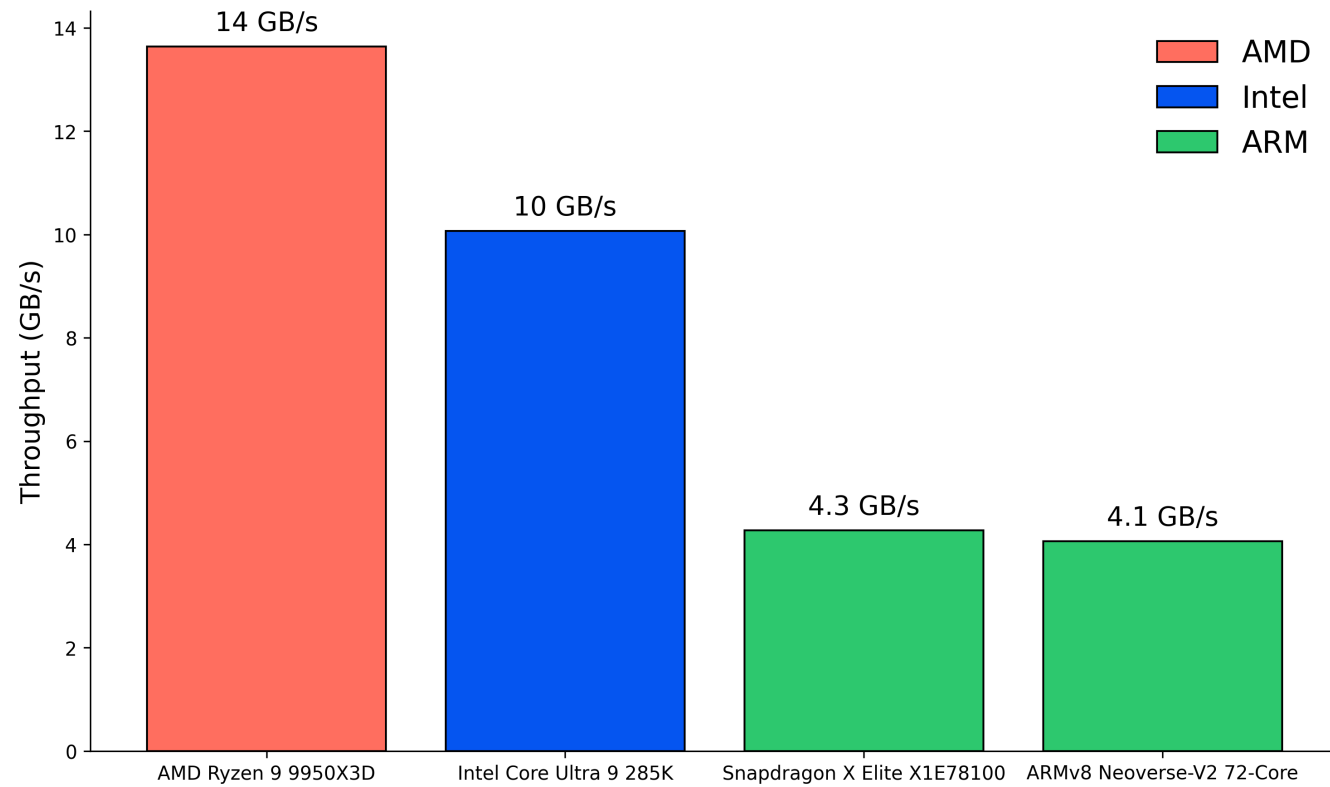
Usage

The simdjson library is found in...

- Node.js

- ClickHouse

- Velox

- Milvus

- QuestDB

- StarRocks

- ...

Automate the serialization/deserialization process.

Player Class

name: "Alice"

score: 100

simdjson::to_json

simdjson::from

JSON

{ "name": "Alice",

"score": 100 }
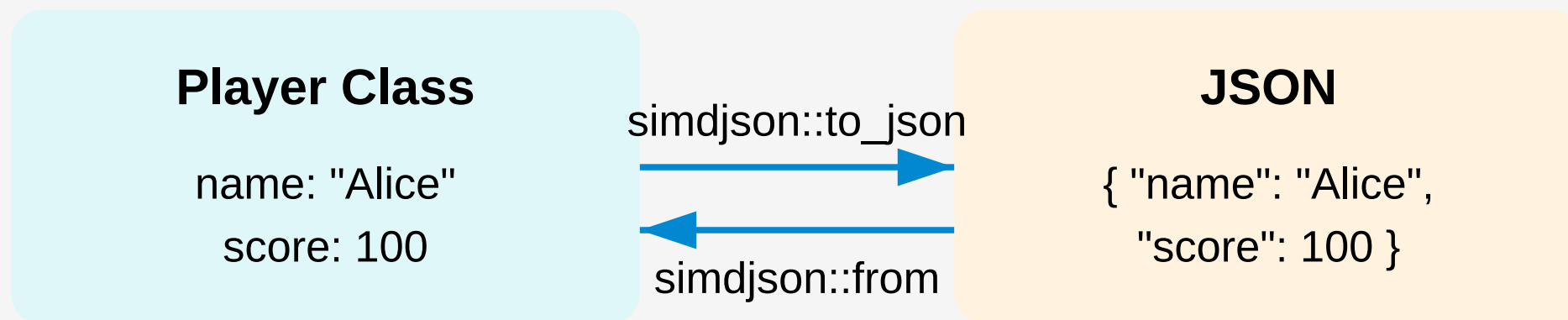
The Problem

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

Player

username: string

level: int

health: double

inventory: string, string, ...

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.

```cpp
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)

```cpp
object obj = val.get_object();
p.username = obj["username"].get_string();
p.level = obj["level"].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**:

```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

```cpp
// 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

```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

Go reflection

- Runtime reflection only

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

28

Java and C#

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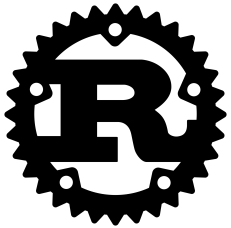
Java

C#

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

```cpp
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;
    auto er = json_value.get_object().get(obj);
    if(er) { return er; }

    // 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!

On-Demand: parse only what you need

```
auto car = doc["Jean-Claude"].get<Car>()
```

- Seeks `""Jean-Claude"` with index, and then parses directly to `Car`.
- No intermediate, no extra parsing

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

```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==
```

47

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.

The Container Challenge

We can say that serializing/parsing the basic types and custom classes/structs is pretty much effortless.

How do we automatically serialize ALL these different containers?

- `std::vector<T>` , `std::list<T>` , `std::deque<T>`
- `std::map<K,V>` , `std::unordered_map<K,V>`
- `std::set<T>` , `std::array<T,N>`
- Custom containers from libraries
- **Future containers not yet invented**

The Naive Approach: Without Concepts

Without concepts, you'd need a separate function for EACH container type:

```
// The OLD way - repetitive and error-prone! 😱
void serialize(string_builder& b, const std::vector<T>& v) { /* ... */ }
void serialize(string_builder& b, const std::list<T>& v) { /* ... */ }
void serialize(string_builder& b, const std::deque<T>& v) { /* ... */ }
void serialize(string_builder& b, const std::set<T>& v) { /* ... */ }
// ... 20+ more overloads for each container type!
```

Problem: New container type? Write more boilerplate!

The Solution: Concepts as Pattern Matching

Concepts let us say: **"If it walks like a duck and quacks like a duck..."**

```cpp
// The NEW way - one function handles ALL array-like containers!
template<typename T>
  requires(has_size_and_subscript<T>)  // "If it has .size() and operator[]"
void serialize(string_builder& b, const T& container) {
    b.append('[');
    for (size_t i = 0; i < container.size(); ++i) {
        serialize(b, container[i]);
    }
    b.append(']');
}
```

✅ Works with `vector` , `array` , `deque` , custom containers...

Concepts + Reflection = Automatic Support

When you write:

```cpp
struct GameData {
    std::vector<int> scores;            // Array-like → [1,2,3]
    std::map<string, Player> players;   // Map-like → {"Alice": {...}}
    MyCustomContainer<Item> items;      // Your container → Just works!
};
```

The magic:

1. **Reflection** discovers your struct's fields

2. **Concepts** match container behavior to serialization strategy

3. **Result**: ALL containers work automatically - standard, custom, or future!

Write once, works everywhere™

Runtime dispatching

- One function semantically

- Several implementations
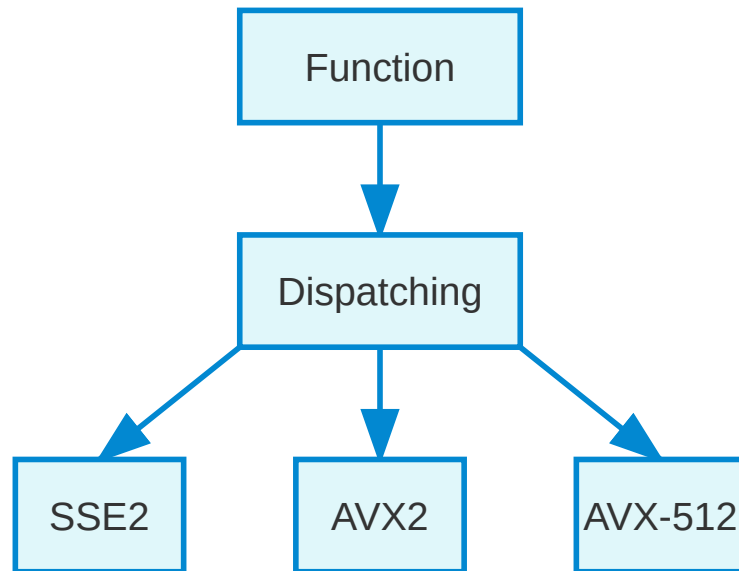
- Select the best one at runtime for performance.

Issue: x64 processors support different instructions

A Zen 5 CPU and a Pentium 4 CPU can be quite different.

```cpp
bool has_sse2() { /* query the CPU */ }
bool has_avx2() { /* query the CPU */ }
bool has_avx512() { /* query the CPU */ }
```

These functions cannot be `consteval`.

Runtime dispatching

Example: Sum function

```cpp
using SumFunc = float (*)(const float *, size_t);
```

Setup a reassignable implementation

```
SumFunc &get_sum_fnc() {
  static SumFunc sum_impl = sum_init;
  return sum_impl;
}
```

We initialize it with some special initialization function.

```
float sum_init(const float *data, size_t n) {
  SumFunc &sum_impl = get_sum_fnc();
  if (has_avx2()) {
    sum_impl = sum_avx2;
  } else if (has_sse2()) {
    sum_impl = sum_sse2;
  } else {
    sum_impl = sum_generic;
  }
  return sum_impl(data, n);
}
```

On first call, `get_sum_fnc()` is modified, and then it will remain constant.

Runtime dispatching and metaprogramming

- Metaprogramming is at compile-time.

- Runtime dispatching is fundamentally at runtime.

Does your string need escaping?

- In JSON, you must escape control characters, quotes.

- Most strings in practice do not need escaping.

```cpp
simple_needs_escaping(std::string_view v) {
  for (unsigned char c : v) {
    if(json_quotable_character[c]) { return true; }
  }
  return false;
}
```

SIMD (Pentium 4 and better)

```
__m128i word = _mm_loadu_si128(data); // load 16 bytes
// check for control characters:
_mm_cmpeq_epi8(_mm_subs_epu8(word, _mm_set1_epi8(31)),
                            _mm_setzero_si128());
```

SIMD (AVX-512)

```
__m512i word = _mm512_loadu_si512(data); // load 64 bytes
// check for control characters:
_mm512_cmple_epu8_mask(word, _mm512_set1_epi8(31));
```

Runtime dispatching is poor with quick functions

- Calling a fast function like `fast_needs_escaping` without inlining prevents useful optimizations.

- Runtime dispatching implies a function call!

Current solution

- No runtime dispatching (*sad face*).

- All x64 processors support Pentium 4-level SIMD. Use that in a short function.

- *Easy* if programmer builds for specific machine (`-march=native`), use fancier tricks.

Compile-time string escaping

- Often the 'keys' are known at compile time.

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

- Keys are: `username` , `level` , `health` , `inventory` .

65

Escape at compile time.

```
[:expand(std::meta::nonstatic_data_members_of(...))] {
  constexpr auto key =
    std::define_static_string(consteval_to_quoted_escaped(
      std::meta::identifier_of(dm)));
  b.append_raw(key);
  b.append(':');
  // ...
};
```

Otherwise tricky to do

- Outside metaprogramming, lots of values are compile-time constants

- But processing it at compile time is not always easy/convenient.

Example: g returns 1

```cpp
constexpr int convert(const char * x) {
    if (std::is_constant_evaluated()) { return 0; }
    return 1;
}

int g() {
    constexpr char key[] = "name";
    auto x = convert(key);
    return x;
}
```

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!