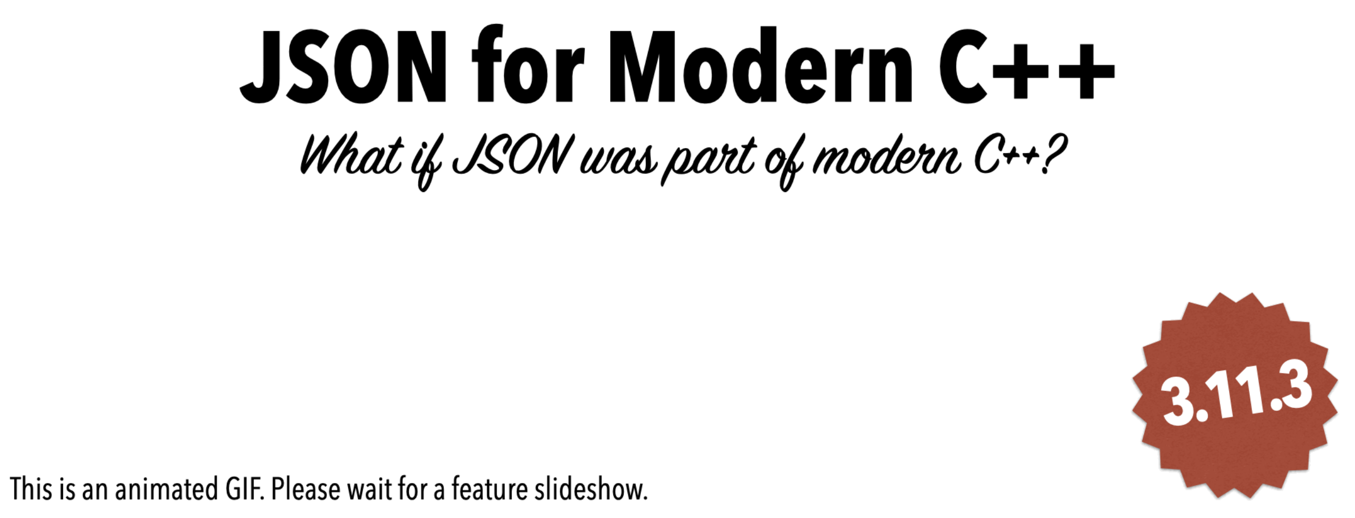

JSON for Modern C++
What if JSON was part of modern C++?
3.11.3
This is an animated GIF. Please wait for a feature slideshow.

```
/*
Note:杨小兵-2025-01-23

1、![JSON for Modern C++](docs/mkdocs/docs/images/json.gif)
  1.1 ![url_name](url)语法用来在markdown中显示资源，例如图片
  1.2 [url_name](url)语法用来在markdown中显示网址资源
  1.3 通过上述组合方式可以直接点击图片跳转到指定的网址
*/
```

build passing | Ubuntu failing | macOS passing | Windows passing | coverage 99%
coverity passed | code quality A | build passing | oss-fuzz fuzzing | try online | docs mkdocs | license MIT
release v3.11.3 | in repositories 39 | downloads 14M | issues 57 open | issue resolution 1 d
openssf best practices passing | openssf scorecard 9.3 | GitHub Sponsors | REUSE compliant | chat 2 online

```
/*
Note:杨小兵-2025-01-23

1、![JSON for Modern C++](docs/mkdocs/docs/images/json.gif)
  1.1 ![url_name](url)语法用来在markdown中显示资源，例如图片
  1.2 [url_name](url)语法用来在markdown中显示网址资源
  1.3 通过上述组合方式可以直接点击图片跳转到指定的网址
  1.4 上述内容采用的都是相同的markdown语法
*/
```

```
/*
Note:杨小兵-2025-01-23

1、[name](chapter)语法用来排版和显示该文档的目录结构和对应跳转的目录。
*/
```

1 Design goals

There are myriads of JSON libraries out there, and each may even have its reason to exist. Our class had these design goals:

```
/*
Note:杨小兵-2025-01-23

1、这部分内容介绍一些关于JSON库的设计目标（design goals）
2、市面上有无数的 [JSON](https://json.org) 库，每个库都有其存在的理由。
  2.1 myriads: 无数
  2.2 从这个消息中可以得到的信息是：市面上存在着各种各样的不同类型的json库，每一个json库
都有其存在的理由。
  2.3 下列的内容想要说明当前的json被设计出来的一些goals。
*/
```

- **Intuitive syntax**. In languages such as Python, JSON feels like a first class data type. We used all the operator magic of modern C++ to achieve the same feeling in your code. Check out the examples below and you'll know what I mean.

  ```
  /*
  Note:杨小兵-2025-01-23

  1、直观的语法
  2、在 Python 等语言中，JSON 感觉就像是一流的数据类型。我们使用现代 C++ 的所有运算
  符魔法在您的代码中实现同样的感觉。查看下面的[示例](#examples)，您就会明白我的意思。
  3、"first-class data type" 指的是 JSON 被当作语言中的一种原生数据类型来使用，意味
  着它可以像基本数据类型（如整数、字符串等）一样被处理和操作，具有相同的灵活性和能力。
  */
  ```

- **Trivial integration**. Our whole code consists of a single header file `json.hpp`. That's it. No library, no subproject, no dependencies, no complex build system. The class is written in vanilla C++11. All in all, everything should require no adjustment of your compiler flags or project settings. The library is also included in all popular package managers.

```
/*
Note:杨小兵-2025-01-23

1、简单集成
2、我们的整个代码由一个头文件 [`json.hpp`]
(https://github.com/nlohmann/json/blob/develop/single_include/nlohmann/json.
hpp) 组成。就是这样。没有库，没有子项目，没有依赖项，没有复杂的构建系统。该类是用原
始 C++11 编写的。总而言之，一切都不需要调整编译器标志或项目设置。该库还包含在所有流
行的 [包管理器](https://json.nlohmann.me/integration/package_managers/) 中。
  2.1 整个代码就是一个头文件构成。
  2.2 使用原始的C++11编写。
  2.3 该json库还可以通过所有流行的包管理器来进行下载安装。
3、这些特点使得该 JSON 库非常适合需要快速、简单集成 JSON 处理功能的 C++ 项目，无论
是个人开发者还是大型团队，都能轻松上手并高效使用。
*/
```

- **Serious testing**. Our code is heavily unit-tested and covers 100% of the code, including all exceptional behavior. Furthermore, we checked with Valgrind and the Clang Sanitizers that there are no memory leaks. Google OSS-Fuzz additionally runs fuzz tests against all parsers 24/7, effectively executing billions of tests so far. To maintain high quality, the project is following the Core Infrastructure Initiative (CII) best practices. See the quality assurance overview documentation.

```
/*
Note:杨小兵-2025-01-23

1、严格测试
2、我们的代码经过了大量的 [单元测试]
(https://github.com/nlohmann/json/tree/develop/tests/src)，覆盖了 [100%]
(https://coveralls.io/r/nlohmann/json) 的代码，包括所有异常行为。此外，我们使用
[Valgrind](https://valgrind.org) 和 [Clang Sanitizers]
(https://clang.llvm.org/docs/index.html) 检查了是否存在内存泄漏。[Google OSS-
Fuzz](https://github.com/google/oss-fuzz/tree/master/projects/json) 还会全天
候对所有解析器运行模糊测试，迄今为止已有效执行了数十亿次测试。为了保持高质量，该项目
遵循 [核心基础设施计划 (CII) 最佳实践]
(https://bestpractices.coreinfrastructure.org/projects/289)。请参阅质量保证概
述文档。
*/
```

Other aspects were not so important to us:

```
/*
Note:杨小兵-2025-01-23
```

```
1、下列的这些方便对此项目来说并不是特别的重要。
*/
```

- **Memory efficiency**. Each JSON object has an overhead of one pointer (the maximal size of a union) and one enumeration element (1 byte). The default generalization uses the following C++ data types: `std::string` for strings, `int64_t`, `uint64_t` or `double` for numbers, `std::map` for objects, `std::vector` for arrays, and `bool` for Booleans. However, you can template the generalized class `basic_json` to your needs.

  ```
  /*
  Note:杨小兵-2025-01-23

  1、内存效率
  2、每个 JSON 对象都有一个指针（联合的最大大小）和一个枚举元素（1 字节）的开销。默认泛化使用以下 C++ 数据类型：`std::string` 用于字符串，`int64_t`、`uint64_t` 或 `double` 用于数字，`std::map` 用于对象，`std::vector` 用于数组，`bool` 用于布尔值。但是，您可以根据需要模板化泛化类 `basic_json`。
  */
  ```

- **Speed**. There are certainly faster JSON libraries out there. However, if your goal is to speed up your development by adding JSON support with a single header, then this library is the way to go. If you know how to use a `std::vector` or `std::map`, you are already set.

  ```
  /*
  Note:杨小兵-2025-01-23

  1、速度
  2、当然有[更快的 JSON 库](https://github.com/miloyip/nativejson-benchmark#parsing-time)。但是，如果您的目标是通过添加单个头文件的 JSON 支持来加快开发速度，那么此库就是您的最佳选择。如果您知道如何使用 `std::vector` 或 `std::map`，那么您已经准备好了。
  */
  ```

See the contribution guidelines for more information.

```
/*
Note:杨小兵-2025-01-23

1、查看[contribution guidelines]
(https://github.com/nlohmann/json/blob/master/.github/CONTRIBUTING.md#please-dont)
获取更多的信息。
*/
```

2 Sponsors

You can sponsor this library at [GitHub Sponsors](https://github.com/sponsors/nlohmann).

```
/*
Note:杨小兵-2025-01-23

1、可以通过[GitHub Sponsors](https://github.com/sponsors/nlohmann)来赞助这个库。
*/
```

2.1 :raising_hand: Priority Sponsor

- Martti Laine
- Paul Harrington

```
/*
Note:杨小兵-2025-01-23

1、上述提到的两个人是"优先赞助商"
*/
```

2.2 :label: Named Sponsors

- Michael Hartmann
- Stefan Hagen
- Steve Sperandeo
- Robert Jefe Lindstädt
- Steve Wagner
- Lion Yang

```
/*
Note:杨小兵-2025-01-23

1、上述提到的多个人是"指定赞助商"
*/
```

2.3 Further support

The development of the library is further supported by JetBrains by providing free access to their IDE tools.

JETBRAINS

Thanks everyone!

```
/*
Note:杨小兵-2025-01-23

1、将来的一些支持。
2、JetBrains 通过提供免费访问其 IDE 工具的方式进一步支持了该库的开发。
3、 JetBrains 通过免费提供其强大的集成开发环境（IDE）工具，支持该库的开发工作。这种支持
不仅降低了开发成本，还提升了开发效率和代码质量，促进了库的持续发展和维护。JetBrains 的支
持对于开发团队来说是一种重要的资源，能够帮助他们更高效地完成项目目标。
*/
```

3 Support

❓ If you have a **question**, please check if it is already answered in the **FAQ** or the **Q&A** section. If not, please **ask a new question** there.

```
/*
Note:杨小兵-2025-01-23

1、:question:
2、如果你有一个question，在FAQ或者Q&A部分请查看该问题是否已经被回答了。如果该问题没有被
回答那么可以在ask a new question中提一个新的问题。
*/
```

📚 If you want to **learn more** about how to use the library, check out the rest of the **README**, have a look at **code examples**, or browse through the **help pages**.

```
/*
Note:杨小兵-2025-01-23

1、:books:
2、如果你想要详细了解如何使用该库，请查看其余的 [README](#examples)，查看[代码示例]
(https://github.com/nlohmann/json/tree/develop/docs/mkdocs/docs/examples)，或浏览
[帮助页面](https://json.nlohmann.me)。
*/
```

🚧 If you want to understand the **API** better, check out the **API Reference** or have a look at the quick reference below.

```
/*
Note:杨小兵-2025-01-23

1、:construction:
2、如果你想要更好地理解 API，请查看 [API 参考]
(https://json.nlohmann.me/api/basic_json/) 或查看下面的 [快速参考](#quick-
reference)。
*/
```

🐛 If you found a **bug**, please check the **FAQ** if it is a known issue or the result of a design decision. Please also have a look at the **issue list** before you **create a new issue**. Please provide as much information as possible to help us understand and reproduce your issue.

```
/*
Note:杨小兵-2025-01-23

1、:bug:
2、如果您发现错误，请查看[常见问题](https://json.nlohmann.me/home/faq/)，确定它是已知问题还是设计决策的结果。在[创建新问题](https://github.com/nlohmann/json/issues)之前，还请查看[问题列表](https://github.com/nlohmann/json/issues)。请提供尽可能多的信息，以帮助我们了解和重现您的问题。
*/
```

There is also a **docset** for the documentation browsers Dash, Velocity, and Zeal that contains the full documentation as offline resource.

```
/*
Note:杨小兵-2025-01-23

1、另外，还提供了一个适用于文档浏览器 [Dash](https://kapeli.com/dash)、[Velocity](https://velocity.silverlakesoftware.com) 和 [Zeal](https://zealdocs.org) 的 [docset](https://github.com/Kapeli/Dash-User-Contributions/tree/master/docsets/JSON_for_Modern_C%2B%2B)，其中包含完整的 [文档](https://json.nlohmann.me) 作为离线资源。
*/
```

4 Quick reference

- **Constructors** basic_json, array, binary, object
- **Object inspection**: type, operator value_t, type_name, is_primitive, is_structured, is_null, is_boolean, is_number, is_number_integer, is_number_unsigned, is_number_float, is_object, is_array, is_string, is_binary, is_discarded
- **Value access**; get, get_to, get_ptr, get_ref, operator ValueType, get_binary
- **Element access**: at, operator[], value, front, back
- **Lookup**: find, count, contains
- **Iterators**: begin, cbegin, end, cend, rbegin, rend, crbegin, crend, items
- **Capacity**: empty, size, max_size
- **Modifiers**: clear, push_back, operator+=, emplace_back, emplace, erase, insert, update, swap
- **Lexicographical comparison operators**: operator==, operator!=, operator<, operator>, operator<=, operator>=, operator<=>
- **Serialization / Dumping**: dump

- **Deserialization / Parsing**: parse, accept, sax_parse
- **JSON Pointer functions**: flatten, unflatten
- **JSON Patch functions**: patch, patch_inplace, diff, merge_patch
- **Static functions**: meta, get_allocator
- **Binary formats**: from_bjdata, from_bson, from_cbor, from_msgpack, from_ubjson, to_bjdata, to_bson, to_cbor, to_msgpack, to_ubjson
- **Non-member functions**: operator<<, operator>>, to_string
- **Literals**: operator""_json
- **Helper classes**: std::hash<basic_json>, std::swap<basic_json>

Full API documentation

5 Examples

Here are some examples to give you an idea how to use the class.

Beside the examples below, you may want to:

→ Check the documentation
→ Browse the standalone example files
→ Read the full API Documentation with self-contained examples for every function

5.1 Read JSON from a file

The `json` class provides an API for manipulating a JSON value. To create a `json` object by reading a JSON file:

```cpp
#include <fstream>
#include <nlohmann/json.hpp>
using json = nlohmann::json;

// ...

std::ifstream f("example.json");
json data = json::parse(f);
```

5.2 Creating `json` objects from JSON literals

Assume you want to create hard-code this literal JSON value in a file, as a `json` object:

```json
{
  "pi": 3.141,
  "happy": true
}
```

There are various options:

```
// Using (raw) string literals and json::parse
json ex1 = json::parse(R"(
  {
    "pi": 3.141,
    "happy": true
  }
)");

// Using user-defined (raw) string literals
using namespace nlohmann::literals;
json ex2 = R"(
  {
    "pi": 3.141,
    "happy": true
  }
)"_json;

// Using initializer lists
json ex3 = {
  {"happy", true},
  {"pi", 3.141},
};
```

5.3 JSON as first-class data type

Here are some examples to give you an idea how to use the class.

Assume you want to create the JSON object

```
{
  "pi": 3.141,
  "happy": true,
  "name": "Niels",
  "nothing": null,
  "answer": {
    "everything": 42
  },
  "list": [1, 0, 2],
  "object": {
    "currency": "USD",
    "value": 42.99
  }
}
```

With this library, you could write:

```
// create an empty structure (null)
json j;
```

```
// add a number that is stored as double (note the implicit conversion of j to an
object)
j["pi"] = 3.141;

// add a Boolean that is stored as bool
j["happy"] = true;

// add a string that is stored as std::string
j["name"] = "Niels";

// add another null object by passing nullptr
j["nothing"] = nullptr;

// add an object inside the object
j["answer"]["everything"] = 42;

// add an array that is stored as std::vector (using an initializer list)
j["list"] = { 1, 0, 2 };

// add another object (using an initializer list of pairs)
j["object"] = { {"currency", "USD"}, {"value", 42.99} };

// instead, you could also write (which looks very similar to the JSON above)
json j2 = {
  {"pi", 3.141},
  {"happy", true},
  {"name", "Niels"},
  {"nothing", nullptr},
  {"answer", {
    {"everything", 42}
  }},
  {"list", {1, 0, 2}},
  {"object", {
    {"currency", "USD"},
    {"value", 42.99}
  }}
};
```

Note that in all these cases, you never need to "tell" the compiler which JSON value type you want to use. If you want to be explicit or express some edge cases, the functions `json::array()` and `json::object()` will help:

```
// a way to express the empty array []
json empty_array_explicit = json::array();

// ways to express the empty object {}
json empty_object_implicit = json({});
json empty_object_explicit = json::object();

// a way to express an _array_ of key/value pairs [["currency", "USD"], ["value",
42.99]]
json array_not_object = json::array({ {"currency", "USD"}, {"value", 42.99} });
```

5.4 Serialization / Deserialization

To/from strings

You can create a JSON value (deserialization) by appending `_json` to a string literal:

```
// create object from string literal
json j = "{ \"happy\": true, \"pi\": 3.141 }"_json;

// or even nicer with a raw string literal
auto j2 = R"(
  {
    "happy": true,
    "pi": 3.141
  }
)"_json;
```

Note that without appending the `_json` suffix, the passed string literal is not parsed, but just used as JSON string value. That is, `json j = "{ \"happy\": true, \"pi\": 3.141 }"` would just store the string `"{ "happy": true, "pi": 3.141 }"` rather than parsing the actual object.

The string literal should be brought into scope with `using namespace nlohmann::literals;` (see `json::parse()`).

The above example can also be expressed explicitly using `json::parse()`:

```
// parse explicitly
auto j3 = json::parse(R"({"happy": true, "pi": 3.141})");
```

You can also get a string representation of a JSON value (serialize):

```
// explicit conversion to string
std::string s = j.dump();    // {"happy":true,"pi":3.141}

// serialization with pretty printing
// pass in the amount of spaces to indent
std::cout << j.dump(4) << std::endl;
// {
//     "happy": true,
//     "pi": 3.141
// }
```

Note the difference between serialization and assignment:

```
// store a string in a JSON value
json j_string = "this is a string";

// retrieve the string value
auto cpp_string = j_string.template get<std::string>();
// retrieve the string value (alternative when a variable already exists)
std::string cpp_string2;
j_string.get_to(cpp_string2);

// retrieve the serialized value (explicit JSON serialization)
std::string serialized_string = j_string.dump();

// output of original string
std::cout << cpp_string << " == " << cpp_string2 << " == " << j_string.template
get<std::string>() << '\n';
// output of serialized value
std::cout << j_string << " == " << serialized_string << std::endl;
```

`.dump()` returns the originally stored string value.

Note the library only supports UTF-8. When you store strings with different encodings in the library, calling `dump()` may throw an exception unless `json::error_handler_t::replace` or `json::error_handler_t::ignore` are used as error handlers.

To/from streams (e.g. files, string streams)

You can also use streams to serialize and deserialize:

```
// deserialize from standard input
json j;
std::cin >> j;

// serialize to standard output
std::cout << j;

// the setw manipulator was overloaded to set the indentation for pretty printing
std::cout << std::setw(4) << j << std::endl;
```

These operators work for any subclasses of `std::istream` or `std::ostream`. Here is the same example with files:

```
// read a JSON file
std::ifstream i("file.json");
json j;
i >> j;
```

```
// write prettified JSON to another file
std::ofstream o("pretty.json");
o << std::setw(4) << j << std::endl;
```

Please note that setting the exception bit for `failbit` is inappropriate for this use case. It will result in program termination due to the `noexcept` specifier in use.

Read from iterator range

You can also parse JSON from an iterator range; that is, from any container accessible by iterators whose `value_type` is an integral type of 1, 2 or 4 bytes, which will be interpreted as UTF-8, UTF-16 and UTF-32 respectively. For instance, a `std::vector<std::uint8_t>`, or a `std::list<std::uint16_t>`:

```
std::vector<std::uint8_t> v = {'t', 'r', 'u', 'e'};
json j = json::parse(v.begin(), v.end());
```

You may leave the iterators for the range [begin, end):

```
std::vector<std::uint8_t> v = {'t', 'r', 'u', 'e'};
json j = json::parse(v);
```

Custom data source

Since the parse function accepts arbitrary iterator ranges, you can provide your own data sources by implementing the `LegacyInputIterator` concept.

```
struct MyContainer {
  void advance();
  const char& get_current();
};

struct MyIterator {
    using difference_type = std::ptrdiff_t;
    using value_type = char;
    using pointer = const char*;
    using reference = const char&;
    using iterator_category = std::input_iterator_tag;

    MyIterator& operator++() {
        target->advance();
        return *this;
    }

    bool operator!=(const MyIterator& rhs) const {
        return rhs.target != target;
    }
    reference operator*() const {
        return target->get_current();
    }

    MyContainer* target = nullptr;
};

MyIterator begin(MyContainer& tgt) {
    return MyIterator{&tgt};
}

MyIterator end(const MyContainer&) {
    return {};
}

void foo() {
    MyContainer c;
    json j = json::parse(c);
}
```

SAX interface

The library uses a SAX-like interface with the following functions:

```
// called when null is parsed
bool null();

// called when a boolean is parsed; value is passed
bool boolean(bool val);

// called when a signed or unsigned integer number is parsed; value is passed
bool number_integer(number_integer_t val);
bool number_unsigned(number_unsigned_t val);

// called when a floating-point number is parsed; value and original string is
passed
bool number_float(number_float_t val, const string_t& s);

// called when a string is parsed; value is passed and can be safely moved away
bool string(string_t& val);
// called when a binary value is parsed; value is passed and can be safely moved
away
bool binary(binary_t& val);

// called when an object or array begins or ends, resp. The number of elements is
passed (or -1 if not known)
bool start_object(std::size_t elements);
bool end_object();
bool start_array(std::size_t elements);
bool end_array();
```

```
// called when an object key is parsed; value is passed and can be safely moved
away
bool key(string_t& val);

// called when a parse error occurs; byte position, the last token, and an
exception is passed
bool parse_error(std::size_t position, const std::string& last_token, const
detail::exception& ex);
```

The return value of each function determines whether parsing should proceed.

To implement your own SAX handler, proceed as follows:

1. Implement the SAX interface in a class. You can use class `nlohmann::json_sax<json>` as base class, but you can also use any class where the functions described above are implemented and public.
2. Create an object of your SAX interface class, e.g. `my_sax`.
3. Call `bool json::sax_parse(input, &my_sax)`; where the first parameter can be any input like a string or an input stream and the second parameter is a pointer to your SAX interface.

Note the `sax_parse` function only returns a `bool` indicating the result of the last executed SAX event. It does not return a `json` value - it is up to you to decide what to do with the SAX events. Furthermore, no exceptions are thrown in case of a parse error - it is up to you what to do with the exception object passed to your `parse_error` implementation. Internally, the SAX interface is used for the DOM parser (class `json_sax_dom_parser`) as well as the acceptor (`json_sax_acceptor`), see file `json_sax.hpp`.

5.5 STL-like access

We designed the JSON class to behave just like an STL container. In fact, it satisfies the **ReversibleContainer** requirement.

```
// create an array using push_back
json j;
j.push_back("foo");
j.push_back(1);
j.push_back(true);

// also use emplace_back
j.emplace_back(1.78);

// iterate the array
for (json::iterator it = j.begin(); it != j.end(); ++it) {
  std::cout << *it << '\n';
}

// range-based for
for (auto& element : j) {
  std::cout << element << '\n';
}

// getter/setter
```

```cpp
const auto tmp = j[0].template get<std::string>();
j[1] = 42;
bool foo = j.at(2);

// comparison
j == R"(["foo", 1, true, 1.78])"_json;  // true

// other stuff
j.size();     // 4 entries
j.empty();    // false
j.type();     // json::value_t::array
j.clear();    // the array is empty again

// convenience type checkers
j.is_null();
j.is_boolean();
j.is_number();
j.is_object();
j.is_array();
j.is_string();

// create an object
json o;
o["foo"] = 23;
o["bar"] = false;
o["baz"] = 3.141;

// also use emplace
o.emplace("weather", "sunny");

// special iterator member functions for objects
for (json::iterator it = o.begin(); it != o.end(); ++it) {
  std::cout << it.key() << " : " << it.value() << "\n";
}

// the same code as range for
for (auto& el : o.items()) {
  std::cout << el.key() << " : " << el.value() << "\n";
}

// even easier with structured bindings (C++17)
for (auto& [key, value] : o.items()) {
  std::cout << key << " : " << value << "\n";
}

// find an entry
if (o.contains("foo")) {
  // there is an entry with key "foo"
}

// or via find and an iterator
if (o.find("foo") != o.end()) {
  // there is an entry with key "foo"
}
```

```
// or simpler using count()
int foo_present = o.count("foo"); // 1
int fob_present = o.count("fob"); // 0

// delete an entry
o.erase("foo");
```

5.6 Conversion from STL containers

Any sequence container (`std::array`, `std::vector`, `std::deque`, `std::forward_list`, `std::list`) whose values can be used to construct JSON values (e.g., integers, floating point numbers, Booleans, string types, or again STL containers described in this section) can be used to create a JSON array. The same holds for similar associative containers (`std::set`, `std::multiset`, `std::unordered_set`, `std::unordered_multiset`), but in these cases the order of the elements of the array depends on how the elements are ordered in the respective STL container.

```
std::vector<int> c_vector {1, 2, 3, 4};
json j_vec(c_vector);
// [1, 2, 3, 4]

std::deque<double> c_deque {1.2, 2.3, 3.4, 5.6};
json j_deque(c_deque);
// [1.2, 2.3, 3.4, 5.6]

std::list<bool> c_list {true, true, false, true};
json j_list(c_list);
// [true, true, false, true]

std::forward_list<int64_t> c_flist {12345678909876, 23456789098765,
34567890987654, 45678909876543};
json j_flist(c_flist);
// [12345678909876, 23456789098765, 34567890987654, 45678909876543]

std::array<unsigned long, 4> c_array {{1, 2, 3, 4}};
json j_array(c_array);
// [1, 2, 3, 4]

std::set<std::string> c_set {"one", "two", "three", "four", "one"};
json j_set(c_set); // only one entry for "one" is used
// ["four", "one", "three", "two"]

std::unordered_set<std::string> c_uset {"one", "two", "three", "four", "one"};
json j_uset(c_uset); // only one entry for "one" is used
// maybe ["two", "three", "four", "one"]

std::multiset<std::string> c_mset {"one", "two", "one", "four"};
json j_mset(c_mset); // both entries for "one" are used
// maybe ["one", "two", "one", "four"]

std::unordered_multiset<std::string> c_umset {"one", "two", "one", "four"};
```

```
json j_umset(c_umset); // both entries for "one" are used
// maybe ["one", "two", "one", "four"]
```

Likewise, any associative key-value containers (`std::map`, `std::multimap`, `std::unordered_map`, `std::unordered_multimap`) whose keys can construct an `std::string` and whose values can be used to construct JSON values (see examples above) can be used to create a JSON object. Note that in case of multimaps only one key is used in the JSON object and the value depends on the internal order of the STL container.

```
std::map<std::string, int> c_map { {"one", 1}, {"two", 2}, {"three", 3} };
json j_map(c_map);
// {"one": 1, "three": 3, "two": 2 }

std::unordered_map<const char*, double> c_umap { {"one", 1.2}, {"two", 2.3},
{"three", 3.4} };
json j_umap(c_umap);
// {"one": 1.2, "two": 2.3, "three": 3.4}

std::multimap<std::string, bool> c_mmap { {"one", true}, {"two", true}, {"three",
false}, {"three", true} };
json j_mmap(c_mmap); // only one entry for key "three" is used
// maybe {"one": true, "two": true, "three": true}

std::unordered_multimap<std::string, bool> c_ummap { {"one", true}, {"two", true},
{"three", false}, {"three", true} };
json j_ummap(c_ummap); // only one entry for key "three" is used
// maybe {"one": true, "two": true, "three": true}
```

5.7 JSON Pointer and JSON Patch

The library supports **JSON Pointer** (RFC 6901) as alternative means to address structured values. On top of this, **JSON Patch** (RFC 6902) allows describing differences between two JSON values - effectively allowing patch and diff operations known from Unix.

```
// a JSON value
json j_original = R"({
  "baz": ["one", "two", "three"],
  "foo": "bar"
})"_json;

// access members with a JSON pointer (RFC 6901)
j_original["/baz/1"_json_pointer];
// "two"

// a JSON patch (RFC 6902)
json j_patch = R"([
  { "op": "replace", "path": "/baz", "value": "boo" },
  { "op": "add", "path": "/hello", "value": ["world"] },
  { "op": "remove", "path": "/foo"}
])"_json;

// apply the patch
json j_result = j_original.patch(j_patch);
// {
//    "baz": "boo",
//    "hello": ["world"]
// }

// calculate a JSON patch from two JSON values
json::diff(j_result, j_original);
// [
//   { "op":" replace", "path": "/baz", "value": ["one", "two", "three"] },
//   { "op": "remove","path": "/hello" },
//   { "op": "add", "path": "/foo", "value": "bar" }
// ]
```

5.8 JSON Merge Patch

The library supports **JSON Merge Patch** (RFC 7386) as a patch format. Instead of using JSON Pointer (see above) to specify values to be manipulated, it describes the changes using a syntax that closely mimics the document being modified.

```
// a JSON value
json j_document = R"({
  "a": "b",
  "c": {
    "d": "e",
    "f": "g"
  }
})"_json;

// a patch
json j_patch = R"({
  "a":"z",
  "c": {
    "f": null
  }
})"_json;

// apply the patch
j_document.merge_patch(j_patch);
// {
//  "a": "z",
//  "c": {
//    "d": "e"
//  }
// }
```

5.9 Implicit conversions

Supported types can be implicitly converted to JSON values.

It is recommended to **NOT USE** implicit conversions **FROM** a JSON value. You can find more details about this recommendation here. You can switch off implicit conversions by defining `JSON_USE_IMPLICIT_CONVERSIONS` to `0` before including the `json.hpp` header. When using CMake, you can also achieve this by setting the option `JSON_ImplicitConversions` to `OFF`.

```
// strings
std::string s1 = "Hello, world!";
json js = s1;
auto s2 = js.template get<std::string>();
// NOT RECOMMENDED
std::string s3 = js;
std::string s4;
s4 = js;

// Booleans
bool b1 = true;
json jb = b1;
auto b2 = jb.template get<bool>();
// NOT RECOMMENDED
bool b3 = jb;
bool b4;
b4 = jb;

// numbers
int i = 42;
json jn = i;
auto f = jn.template get<double>();
// NOT RECOMMENDED
double f2 = jb;
double f3;
f3 = jb;

// etc.
```

Note that `char` types are not automatically converted to JSON strings, but to integer numbers. A conversion to a string must be specified explicitly:

```
char ch = 'A';                       // ASCII value 65
json j_default = ch;                 // stores integer number 65
json j_string = std::string(1, ch);  // stores string "A"
```

5.10 Arbitrary types conversions

Every type can be serialized in JSON, not just STL containers and scalar types. Usually, you would do something along those lines:

```cpp
namespace ns {
    // a simple struct to model a person
    struct person {
        std::string name;
        std::string address;
        int age;
    };
}

ns::person p = {"Ned Flanders", "744 Evergreen Terrace", 60};

// convert to JSON: copy each value into the JSON object
json j;
j["name"] = p.name;
j["address"] = p.address;
j["age"] = p.age;

// ...

// convert from JSON: copy each value from the JSON object
ns::person p {
    j["name"].template get<std::string>(),
    j["address"].template get<std::string>(),
    j["age"].template get<int>()
};
```

It works, but that's quite a lot of boilerplate... Fortunately, there's a better way:

```cpp
// create a person
ns::person p {"Ned Flanders", "744 Evergreen Terrace", 60};

// conversion: person -> json
json j = p;

std::cout << j << std::endl;
// {"address":"744 Evergreen Terrace","age":60,"name":"Ned Flanders"}

// conversion: json -> person
auto p2 = j.template get<ns::person>();

// that's it
assert(p == p2);
```

Basic usage

To make this work with one of your types, you only need to provide two functions:

```
using json = nlohmann::json;

namespace ns {
    void to_json(json& j, const person& p) {
        j = json{{"name", p.name}, {"address", p.address}, {"age", p.age}};
    }

    void from_json(const json& j, person& p) {
        j.at("name").get_to(p.name);
        j.at("address").get_to(p.address);
        j.at("age").get_to(p.age);
    }
} // namespace ns
```

That's all! When calling the `json` constructor with your type, your custom `to_json` method will be automatically called. Likewise, when calling `template get<your_type>()` or `get_to(your_type&)`, the `from_json` method will be called.

Some important things:

- Those methods **MUST** be in your type's namespace (which can be the global namespace), or the library will not be able to locate them (in this example, they are in namespace `ns`, where `person` is defined).
- Those methods **MUST** be available (e.g., proper headers must be included) everywhere you use these conversions. Look at issue 1108 for errors that may occur otherwise.
- When using `template get<your_type>()`, `your_type` **MUST** be DefaultConstructible. (There is a way to bypass this requirement described later.)
- In function `from_json`, use function `at()` to access the object values rather than `operator[]`. In case a key does not exist, `at` throws an exception that you can handle, whereas `operator[]` exhibits undefined behavior.
- You do not need to add serializers or deserializers for STL types like `std::vector`: the library already implements these.

Simplify your life with macros

If you just want to serialize/deserialize some structs, the `to_json`/`from_json` functions can be a lot of boilerplate. There are **several macros** to make your life easier as long as you (1) want to use a JSON object as serialization and (2) want to use the member variable names as object keys in that object.

Which macro to choose depends on whether private member variables need to be accessed, a deserialization is needed, missing values should yield an error or should be replaced by default values, and if derived classes are used. See this overview to choose the right one for your use case.

Example usage of macros

The `to_json`/`from_json` functions for the `person` struct above can be created with `NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE`. In all macros, the first parameter is the name of the class/struct, and all remaining parameters name the members.

```
namespace ns {
    NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE(person, name, address, age)
}
```

Here is another example with private members, where `NLOHMANN_DEFINE_TYPE_INTRUSIVE` is needed:

```
namespace ns {
    class address {
      private:
        std::string street;
        int housenumber;
        int postcode;

      public:
        NLOHMANN_DEFINE_TYPE_INTRUSIVE(address, street, housenumber, postcode)
    };
}
```

How do I convert third-party types?

This requires a bit more advanced technique. But first, let's see how this conversion mechanism works:

The library uses **JSON Serializers** to convert types to json. The default serializer for `nlohmann::json` is `nlohmann::adl_serializer` (ADL means Argument-Dependent Lookup).

It is implemented like this (simplified):

```
template <typename T>
struct adl_serializer {
    static void to_json(json& j, const T& value) {
        // calls the "to_json" method in T's namespace
    }

    static void from_json(const json& j, T& value) {
        // same thing, but with the "from_json" method
    }
};
```

This serializer works fine when you have control over the type's namespace. However, what about `boost::optional` or `std::filesystem::path` (C++17)? Hijacking the `boost` namespace is pretty bad, and it's illegal to add something other than template specializations to `std`...

To solve this, you need to add a specialization of `adl_serializer` to the `nlohmann` namespace, here's an example:

```cpp
// partial specialization (full specialization works too)
namespace nlohmann {
    template <typename T>
    struct adl_serializer<boost::optional<T>> {
        static void to_json(json& j, const boost::optional<T>& opt) {
            if (opt == boost::none) {
                j = nullptr;
            } else {
              j = *opt; // this will call adl_serializer<T>::to_json which will
                        // find the free function to_json in T's namespace!
            }
        }

        static void from_json(const json& j, boost::optional<T>& opt) {
            if (j.is_null()) {
                opt = boost::none;
            } else {
                opt = j.template get<T>(); // same as above, but with
                                           // adl_serializer<T>::from_json
            }
        }
    };
}
```

How can I use `get()` for non-default constructible/non-copyable types?

There is a way, if your type is MoveConstructible. You will need to specialize the `adl_serializer` as well, but with a special `from_json` overload:

```cpp
struct move_only_type {
    move_only_type() = delete;
    move_only_type(int ii): i(ii) {}
    move_only_type(const move_only_type&) = delete;
    move_only_type(move_only_type&&) = default;

    int i;
};

namespace nlohmann {
    template <>
    struct adl_serializer<move_only_type> {
        // note: the return type is no longer 'void', and the method only takes
        // one argument
        static move_only_type from_json(const json& j) {
            return {j.template get<int>()};
        }

        // Here's the catch! You must provide a to_json method! Otherwise, you
        // will not be able to convert move_only_type to json, since you fully
        // specialized adl_serializer on that type
        static void to_json(json& j, move_only_type t) {
            j = t.i;
        }
    };
}
```

Can I write my own serializer? (Advanced use)

Yes. You might want to take a look at `unit-udt.cpp` in the test suite, to see a few examples.

If you write your own serializer, you'll need to do a few things:

- use a different `basic_json` alias than `nlohmann::json` (the last template parameter of `basic_json` is the `JSONSerializer`)
- use your `basic_json` alias (or a template parameter) in all your `to_json`/`from_json` methods
- use `nlohmann::to_json` and `nlohmann::from_json` when you need ADL

Here is an example, without simplifications, that only accepts types with a size <= 32, and uses ADL.

```
// You should use void as a second template argument
// if you don't need compile-time checks on T
template<typename T, typename SFINAE = typename std::enable_if<sizeof(T) <=
32>::type>
struct less_than_32_serializer {
    template <typename BasicJsonType>
    static void to_json(BasicJsonType& j, T value) {
        // we want to use ADL, and call the correct to_json overload
        using nlohmann::to_json; // this method is called by adl_serializer,
                                 // this is where the magic happens
        to_json(j, value);
    }

    template <typename BasicJsonType>
    static void from_json(const BasicJsonType& j, T& value) {
        // same thing here
        using nlohmann::from_json;
        from_json(j, value);
    }
};
```

Be **very** careful when reimplementing your serializer, you can stack overflow if you don't pay attention:

```
template <typename T, void>
struct bad_serializer
{
    template <typename BasicJsonType>
    static void to_json(BasicJsonType& j, const T& value) {
      // this calls BasicJsonType::json_serializer<T>::to_json(j, value)
      // if BasicJsonType::json_serializer == bad_serializer ... oops!
```

```
        j = value;
    }

    template <typename BasicJsonType>
    static void to_json(const BasicJsonType& j, T& value) {
      // this calls BasicJsonType::json_serializer<T>::from_json(j, value)
      // if BasicJsonType::json_serializer == bad_serializer ... oops!
      value = j.template get<T>(); // oops!
    }
};
```

5.11 Specializing enum conversion

By default, enum values are serialized to JSON as integers. In some cases this could result in undesired behavior. If an enum is modified or re-ordered after data has been serialized to JSON, the later de-serialized JSON data may be undefined or a different enum value than was originally intended.

It is possible to more precisely specify how a given enum is mapped to and from JSON as shown below:

```
// example enum type declaration
enum TaskState {
    TS_STOPPED,
    TS_RUNNING,
    TS_COMPLETED,
    TS_INVALID=-1,
};

// map TaskState values to JSON as strings
NLOHMANN_JSON_SERIALIZE_ENUM( TaskState, {
    {TS_INVALID, nullptr},
    {TS_STOPPED, "stopped"},
    {TS_RUNNING, "running"},
    {TS_COMPLETED, "completed"},
})
```

The `NLOHMANN_JSON_SERIALIZE_ENUM()` macro declares a set of `to_json()` / `from_json()` functions for type `TaskState` while avoiding repetition and boilerplate serialization code.

Usage:

```
// enum to JSON as string
json j = TS_STOPPED;
assert(j == "stopped");

// json string to enum
json j3 = "running";
assert(j3.template get<TaskState>() == TS_RUNNING);

// undefined json value to enum (where the first map entry above is the default)
```

```
json jPi = 3.14;
assert(jPi.template get<TaskState>() == TS_INVALID);
```

Just as in Arbitrary Type Conversions above,

- `NLOHMANN_JSON_SERIALIZE_ENUM()` MUST be declared in your enum type's namespace (which can be the global namespace), or the library will not be able to locate it, and it will default to integer serialization.
- It MUST be available (e.g., proper headers must be included) everywhere you use the conversions.

Other Important points:

- When using `template get<ENUM_TYPE>()`, undefined JSON values will default to the first pair specified in your map. Select this default pair carefully.
- If an enum or JSON value is specified more than once in your map, the first matching occurrence from the top of the map will be returned when converting to or from JSON.

5.12 Binary formats (BSON, CBOR, MessagePack, UBJSON, and BJData)

Though JSON is a ubiquitous data format, it is not a very compact format suitable for data exchange, for instance over a network. Hence, the library supports BSON (Binary JSON), CBOR (Concise Binary Object Representation), MessagePack, UBJSON (Universal Binary JSON Specification) and BJData (Binary JData) to efficiently encode JSON values to byte vectors and to decode such vectors.

```
// create a JSON value
json j = R"({"compact": true, "schema": 0})"_json;

// serialize to BSON
std::vector<std::uint8_t> v_bson = json::to_bson(j);

// 0x1B, 0x00, 0x00, 0x00, 0x08, 0x63, 0x6F, 0x6D, 0x70, 0x61, 0x63, 0x74, 0x00,
0x01, 0x10, 0x73, 0x63, 0x68, 0x65, 0x6D, 0x61, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

// roundtrip
json j_from_bson = json::from_bson(v_bson);

// serialize to CBOR
std::vector<std::uint8_t> v_cbor = json::to_cbor(j);

// 0xA2, 0x67, 0x63, 0x6F, 0x6D, 0x70, 0x61, 0x63, 0x74, 0xF5, 0x66, 0x73, 0x63,
0x68, 0x65, 0x6D, 0x61, 0x00

// roundtrip
json j_from_cbor = json::from_cbor(v_cbor);

// serialize to MessagePack
std::vector<std::uint8_t> v_msgpack = json::to_msgpack(j);

// 0x82, 0xA7, 0x63, 0x6F, 0x6D, 0x70, 0x61, 0x63, 0x74, 0xC3, 0xA6, 0x73, 0x63,
0x68, 0x65, 0x6D, 0x61, 0x00
```

```
// roundtrip
json j_from_msgpack = json::from_msgpack(v_msgpack);

// serialize to UBJSON
std::vector<std::uint8_t> v_ubjson = json::to_ubjson(j);

// 0x7B, 0x69, 0x07, 0x63, 0x6F, 0x6D, 0x70, 0x61, 0x63, 0x74, 0x54, 0x69, 0x06,
0x73, 0x63, 0x68, 0x65, 0x6D, 0x61, 0x69, 0x00, 0x7D

// roundtrip
json j_from_ubjson = json::from_ubjson(v_ubjson);
```

The library also supports binary types from BSON, CBOR (byte strings), and MessagePack (bin, ext, fixext). They are stored by default as `std::vector<std::uint8_t>` to be processed outside the library.

```
// CBOR byte string with payload 0xCAFE
std::vector<std::uint8_t> v = {0x42, 0xCA, 0xFE};

// read value
json j = json::from_cbor(v);

// the JSON value has type binary
j.is_binary(); // true

// get reference to stored binary value
auto& binary = j.get_binary();

// the binary value has no subtype (CBOR has no binary subtypes)
binary.has_subtype(); // false

// access std::vector<std::uint8_t> member functions
binary.size(); // 2
binary[0]; // 0xCA
binary[1]; // 0xFE

// set subtype to 0x10
binary.set_subtype(0x10);

// serialize to MessagePack
auto cbor = json::to_msgpack(j); // 0xD5 (fixext2), 0x10, 0xCA, 0xFE
```

6 Customers

The library is used in multiple projects, applications, operating systems, etc. The list below is not exhaustive, but the result of an internet search. If you know further customers of the library, please let me know, see contact.

7 Supported compilers

Though it's 2025 already, the support for C++11 is still a bit sparse. Currently, the following compilers are known to work:

- GCC 4.8 - 14.2 (and possibly later)
- Clang 3.4 - 20.0 (and possibly later)
- Apple Clang 9.1 - 16.0 (and possibly later)
- Intel C++ Compiler 17.0.2 (and possibly later)
- Nvidia CUDA Compiler 11.0.221 (and possibly later)
- Microsoft Visual C++ 2015 / Build Tools 14.0.25123.0 (and possibly later)
- Microsoft Visual C++ 2017 / Build Tools 15.5.180.51428 (and possibly later)
- Microsoft Visual C++ 2019 / Build Tools 16.3.1+1def00d3d (and possibly later)
- Microsoft Visual C++ 2022 / Build Tools 19.30.30709.0 (and possibly later)

I would be happy to learn about other compilers/versions.

Please note:

- GCC 4.8 has a bug 57824: multiline raw strings cannot be the arguments to macros. Don't use multiline raw strings directly in macros with this compiler.

- Android defaults to using very old compilers and C++ libraries. To fix this, add the following to your `Application.mk`. This will switch to the LLVM C++ library, the Clang compiler, and enable C++11 and other features disabled by default.

  ```
  APP_STL := c++_shared
  NDK_TOOLCHAIN_VERSION := clang3.6
  APP_CPPFLAGS += -frtti -fexceptions
  ```

  The code compiles successfully with Android NDK, Revision 9 - 11 (and possibly later) and CrystaX's Android NDK version 10.

- For GCC running on MinGW or Android SDK, the error `'to_string' is not a member of 'std'` (or similarly, for `strtod` or `strtof`) may occur. Note this is not an issue with the code, but rather with the compiler itself. On Android, see above to build with a newer environment. For MinGW, please refer to this site and this discussion for information on how to fix this bug. For Android NDK using `APP_STL := gnustl_static`, please refer to this discussion.
- Unsupported versions of GCC and Clang are rejected by `#error` directives. This can be switched off by defining `JSON_SKIP_UNSUPPORTED_COMPILER_CHECK`. Note that you can expect no support in this case.

See the page quality assurance on the compilers used to check the library in the CI.

8 Integration

`json.hpp` is the single required file in `single_include/nlohmann` or released here. You need to add

```
#include <nlohmann/json.hpp>

// for convenience
using json = nlohmann::json;
```

to the files you want to process JSON and set the necessary switches to enable C++11 (e.g., `-std=c++11` for GCC and Clang).

You can further use file `include/nlohmann/json_fwd.hpp` for forward-declarations. The installation of json_fwd.hpp (as part of cmake's install step), can be achieved by setting `-DJSON_MultipleHeaders=ON`.

CMake

You can also use the `nlohmann_json::nlohmann_json` interface target in CMake. This target populates the appropriate usage requirements for `INTERFACE_INCLUDE_DIRECTORIES` to point to the appropriate include directories and `INTERFACE_COMPILE_FEATURES` for the necessary C++11 flags.

External

To use this library from a CMake project, you can locate it directly with `find_package()` and use the namespaced imported target from the generated package configuration:

```
# CMakeLists.txt
find_package(nlohmann_json 3.11.3 REQUIRED)
...
add_library(foo ...)
...
target_link_libraries(foo PRIVATE nlohmann_json::nlohmann_json)
```

The package configuration file, `nlohmann_jsonConfig.cmake`, can be used either from an install tree or directly out of the build tree.

Embedded

To embed the library directly into an existing CMake project, place the entire source tree in a subdirectory and call `add_subdirectory()` in your `CMakeLists.txt` file:

```
# Typically you don't care so much for a third party library's tests to be
# run from your own project's code.
set(JSON_BuildTests OFF CACHE INTERNAL "")

# If you only include this third party in PRIVATE source files, you do not
# need to install it when your main project gets installed.
# set(JSON_Install OFF CACHE INTERNAL "")

# Don't use include(nlohmann_json/CMakeLists.txt) since that carries with it
# unintended consequences that will break the build.  It's generally
# discouraged (although not necessarily well documented as such) to use
# include(...) for pulling in other CMake projects anyways.
add_subdirectory(nlohmann_json)
...
add_library(foo ...)
...
target_link_libraries(foo PRIVATE nlohmann_json::nlohmann_json)
```

Embedded (FetchContent)

Since CMake v3.11, FetchContent can be used to automatically download a release as a dependency at configure time.

Example:

```
include(FetchContent)

FetchContent_Declare(json URL
https://github.com/nlohmann/json/releases/download/v3.11.3/json.tar.xz)
FetchContent_MakeAvailable(json)

target_link_libraries(foo PRIVATE nlohmann_json::nlohmann_json)
```

Note: It is recommended to use the URL approach described above which is supported as of version 3.10.0. See https://json.nlohmann.me/integration/cmake/#fetchcontent for more information.

Supporting Both

To allow your project to support either an externally supplied or an embedded JSON library, you can use a pattern akin to the following:

```
# Top level CMakeLists.txt
project(FOO)
...
option(FOO_USE_EXTERNAL_JSON "Use an external JSON library" OFF)
...
add_subdirectory(thirdparty)
...
add_library(foo ...)
...
# Note that the namespaced target will always be available regardless of the
# import method
target_link_libraries(foo PRIVATE nlohmann_json::nlohmann_json)
```

```
# thirdparty/CMakeLists.txt
...
if(FOO_USE_EXTERNAL_JSON)
  find_package(nlohmann_json 3.11.3 REQUIRED)
else()
  set(JSON_BuildTests OFF CACHE INTERNAL "")
  add_subdirectory(nlohmann_json)
endif()
...
```

`thirdparty/nlohmann_json` is then a complete copy of this source tree.

Package Managers

Use your favorite **package manager** to use the library.

- **Homebrew** `nlohmann-json`
- **Meson** `nlohmann_json`
- **Bazel** `nlohmann_json`
- **Conan** `nlohmann_json`
- **Spack** `nlohmann-json`
- **Hunter** `nlohmann_json`
- **vcpkg** `nlohmann-json`
- **cget** `nlohmann/json`
- **Swift Package Manager** `nlohmann/json`
- **Nuget** `nlohmann.json`
- **Conda** `nlohmann_json`
- **MacPorts** `nlohmann-json`
- **cpm.cmake** `gh:nlohmann/json`
- **xmake** `nlohmann_json`

The library is part of many package managers. See the **documentation** for detailed descriptions and examples.

Pkg-config

If you are using bare Makefiles, you can use `pkg-config` to generate the include flags that point to where the library is installed:

```
pkg-config nlohmann_json --cflags
```

9 License

- The class contains the UTF-8 Decoder from Bjoern Hoehrmann which is licensed under the MIT License (see above). Copyright © 2008-2009 Björn Hoehrmann bjoern@hoehrmann.de
- The class contains a slightly modified version of the Grisu2 algorithm from Florian Loitsch which is licensed under the MIT License (see above). Copyright © 2009 Florian Loitsch
- The class contains a copy of Hedley from Evan Nemerson which is licensed as CC0-1.0.
- The class contains parts of Google Abseil which is licensed under the Apache 2.0 License.

The library is compliant to version 3.3 of the **REUSE specification**:

- Every source file contains an SPDX copyright header.
- The full text of all licenses used in the repository can be found in the `LICENSES` folder.
- File `.reuse/dep5` contains an overview of all files' copyrights and licenses.
- Run `pipx run reuse lint` to verify the project's REUSE compliance and `pipx run reuse spdx` to generate a SPDX SBOM.

10 Contact

If you have questions regarding the library, I would like to invite you to open an issue at GitHub. Please describe your request, problem, or question as detailed as possible, and also mention the version of the library you are using as well as the version of your compiler and operating system. Opening an issue at GitHub allows other users and contributors to this library to collaborate. For instance, I have little experience with MSVC, and most issues in this regard have been solved by a growing community. If you have a look at the closed issues, you will see that we react quite timely in most cases.

Only if your request would contain confidential information, please send me an email. For encrypted messages, please use this key.

11 Security

Commits by Niels Lohmann and releases are signed with this PGP Key.

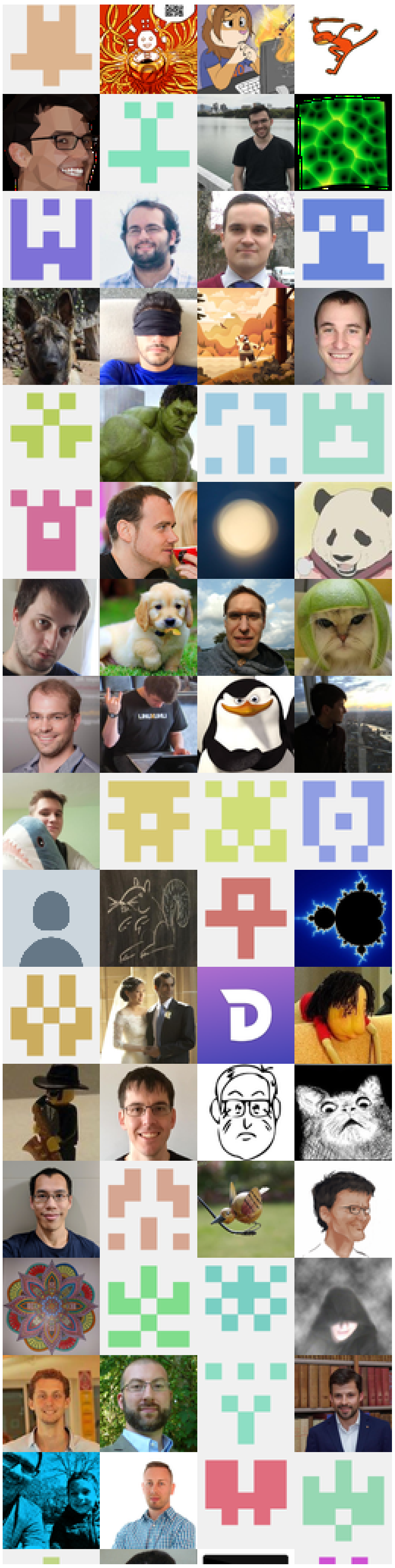
12 Thanks

I deeply appreciate the help of the following people.

1. Teemperor implemented CMake support and lcov integration, realized escape and Unicode handling in the string parser, and fixed the JSON serialization.
2. elliotgoodrich fixed an issue with double deletion in the iterator classes.
3. kirkshoop made the iterators of the class composable to other libraries.
4. wancw fixed a bug that hindered the class to compile with Clang.
5. Tomas Åblad found a bug in the iterator implementation.
6. Joshua C. Randall fixed a bug in the floating-point serialization.
7. Aaron Burghardt implemented code to parse streams incrementally. Furthermore, he greatly improved the parser class by allowing the definition of a filter function to discard undesired elements while parsing.
8. Daniel Kopeček fixed a bug in the compilation with GCC 5.0.
9. Florian Weber fixed a bug in and improved the performance of the comparison operators.
10. Eric Cornelius pointed out a bug in the handling with NaN and infinity values. He also improved the performance of the string escaping.
11. 易思龙 implemented a conversion from anonymous enums.
12. kepkin patiently pushed forward the support for Microsoft Visual studio.
13. gregmarr simplified the implementation of reverse iterators and helped with numerous hints and improvements. In particular, he pushed forward the implementation of user-defined types.
14. Caio Luppi fixed a bug in the Unicode handling.
15. dariomt fixed some typos in the examples.
16. Daniel Frey cleaned up some pointers and implemented exception-safe memory allocation.
17. Colin Hirsch took care of a small namespace issue.
18. Huu Nguyen correct a variable name in the documentation.
19. Silverweed overloaded `parse()` to accept an rvalue reference.
20. dariomt fixed a subtlety in MSVC type support and implemented the `get_ref()` function to get a reference to stored values.
21. ZahlGraf added a workaround that allows compilation using Android NDK.
22. whackashoe replaced a function that was marked as unsafe by Visual Studio.

23. 406345 fixed two small warnings.
24. Glen Fernandes noted a potential portability problem in the `has_mapped_type` function.
25. Corbin Hughes fixed some typos in the contribution guidelines.
26. twelsby fixed the array subscript operator, an issue that failed the MSVC build, and floating-point parsing/dumping. He further added support for unsigned integer numbers and implemented better roundtrip support for parsed numbers.
27. Volker Diels-Grabsch fixed a link in the README file.
28. msm- added support for American Fuzzy Lop.
29. Annihil fixed an example in the README file.
30. Themercee noted a wrong URL in the README file.
31. Lv Zheng fixed a namespace issue with `int64_t` and `uint64_t`.
32. abc100m analyzed the issues with GCC 4.8 and proposed a partial solution.
33. zewt added useful notes to the README file about Android.
34. Róbert Márki added a fix to use move iterators and improved the integration via CMake.
35. Chris Kitching cleaned up the CMake files.
36. Tom Needham fixed a subtle bug with MSVC 2015 which was also proposed by Michael K..
37. Mário Feroldi fixed a small typo.
38. duncanwerner found a really embarrassing performance regression in the 2.0.0 release.
39. Damien fixed one of the last conversion warnings.
40. Thomas Braun fixed a warning in a test case and adjusted MSVC calls in the CI.
41. Théo DELRIEU patiently and constructively oversaw the long way toward iterator-range parsing. He also implemented the magic behind the serialization/deserialization of user-defined types and split the single header file into smaller chunks.
42. Stefan fixed a minor issue in the documentation.
43. Vasil Dimov fixed the documentation regarding conversions from `std::multiset`.
44. ChristophJud overworked the CMake files to ease project inclusion.
45. Vladimir Petrigo made a SFINAE hack more readable and added Visual Studio 17 to the build matrix.
46. Denis Andrejew fixed a grammar issue in the README file.
47. Pierre-Antoine Lacaze found a subtle bug in the `dump()` function.
48. TurpentineDistillery pointed to `std::locale::classic()` to avoid too much locale joggling, found some nice performance improvements in the parser, improved the benchmarking

code, and realized locale-independent number parsing and printing.
49. cgzones had an idea how to fix the Coverity scan.
50. Jared Grubb silenced a nasty documentation warning.
51. Yixin Zhang fixed an integer overflow check.
52. Bosswestfalen merged two iterator classes into a smaller one.
53. Daniel599 helped to get Travis execute the tests with Clang's sanitizers.
54. Jonathan Lee fixed an example in the README file.
55. gnzlbg supported the implementation of user-defined types.
56. Alexej Harm helped to get the user-defined types working with Visual Studio.
57. Jared Grubb supported the implementation of user-defined types.
58. EnricoBilla noted a typo in an example.
59. Martin Hořeňovský found a way for a 2x speedup for the compilation time of the test suite.
60. ukhegg found proposed an improvement for the examples section.
61. rswanson-ihi noted a typo in the README.
62. Mihai Stan fixed a bug in the comparison with `nullptr`s.
63. Tushar Maheshwari added cotire support to speed up the compilation.
64. TedLyngmo noted a typo in the README, removed unnecessary bit arithmetic, and fixed some `-Weffc++` warnings.
65. Krzysztof Woś made exceptions more visible.
66. ftillier fixed a compiler warning.
67. tinloaf made sure all pushed warnings are properly popped.
68. Fytch found a bug in the documentation.
69. Jay Sistar implemented a Meson build description.
70. Henry Lee fixed a warning in ICC and improved the iterator implementation.
71. Vincent Thiery maintains a package for the Conan package manager.
72. Steffen fixed a potential issue with MSVC and `std::min`.
73. Mike Tzou fixed some typos.
74. amrcode noted a misleading documentation about comparison of floats.
75. Oleg Endo reduced the memory consumption by replacing `<iostream>` with `<iosfwd>`.
76. dan-42 cleaned up the CMake files to simplify including/reusing of the library.
77. Nikita Ofitserov allowed for moving values from initializer lists.
78. Greg Hurrell fixed a typo.



79. Dmitry Kukovinets fixed a typo.
80. kbthomp1 fixed an issue related to the Intel OSX compiler.
81. Markus Werle fixed a typo.
82. WebProdPP fixed a subtle error in a precondition check.
83. Alex noted an error in a code sample.
84. Tom de Geus reported some warnings with ICC and helped to fix them.
85. Perry Kundert simplified reading from input streams.
86. Sonu Lohani fixed a small compilation error.
87. Jamie Seward fixed all MSVC warnings.
88. Nate Vargas added a Doxygen tag file.
89. pvleuven helped to fix a warning in ICC.
90. Pavel helped to fix some warnings in MSVC.
91. Jamie Seward avoided unnecessary string copies in `find()` and `count()`.
92. Mitja fixed some typos.
93. Jorrit Wronski updated the Hunter package links.
94. Matthias Möller added a `.natvis` for the MSVC debug view.
95. bogemic fixed some C++17 deprecation warnings.
96. Eren Okka fixed some MSVC warnings.
97. abolz integrated the Grisu2 algorithm for proper floating-point formatting, allowing more roundtrip checks to succeed.
98. Vadim Evard fixed a Markdown issue in the README.
99. zerodefect fixed a compiler warning.
100. Kert allowed to template the string type in the serialization and added the possibility to override the exceptional behavior.
101. mark-99 helped fixing an ICC error.
102. Patrik Huber fixed links in the README file.
103. johnfb found a bug in the implementation of CBOR's indefinite length strings.
104. Paul Fultz II added a note on the cget package manager.
105. Wilson Lin made the integration section of the README more concise.
106. RalfBielig detected and fixed a memory leak in the parser callback.
107. agrianius allowed to dump JSON to an alternative string type.
108. Kevin Tonon overworked the C++11 compiler checks in CMake.
109. Axel Huebl simplified a CMake check and added support for the Spack package manager.
110. Carlos O'Ryan fixed a typo.
111. James Upjohn fixed a version number in the compilers section.
112. Chuck Atkins adjusted the CMake files to the CMake packaging guidelines and provided documentation for the

CMake integration.

113. Jan Schöppach fixed a typo.
114. martin-mfg fixed a typo.
115. Matthias Möller removed the dependency from `std::stringstream`.
116. agrianius added code to use alternative string implementations.
117. Daniel599 allowed to use more algorithms with the `items()` function.
118. Julius Rakow fixed the Meson include directory and fixed the links to cppreference.com.
119. Sonu Lohani fixed the compilation with MSVC 2015 in debug mode.
120. grembo fixed the test suite and re-enabled several test cases.
121. Hyeon Kim introduced the macro `JSON_INTERNAL_CATCH` to control the exception handling inside the library.
122. thyu fixed a compiler warning.
123. David Guthrie fixed a subtle compilation error with Clang 3.4.2.
124. Dennis Fischer allowed to call `find_package` without installing the library.
125. Hyeon Kim fixed an issue with a double macro definition.
126. Ben Berman made some error messages more understandable.
127. zakalibit fixed a compilation problem with the Intel C++ compiler.
128. mandreyel fixed a compilation problem.
129. Kostiantyn Ponomarenko added version and license information to the Meson build file.
130. Henry Schreiner added support for GCC 4.8.
131. knilch made sure the test suite does not stall when run in the wrong directory.
132. Antonio Borondo fixed an MSVC 2017 warning.
133. Dan Gendreau implemented the `NLOHMANN_JSON_SERIALIZE_ENUM` macro to quickly define an enum/JSON mapping.
134. efp added line and column information to parse errors.
135. julian-becker added BSON support.
136. Pratik Chowdhury added support for structured bindings.
137. David Avedissian added support for Clang 5.0.1 (PS4 version).
138. Jonathan Dumaresq implemented an input adapter to read from `FILE*`.
139. kjpus fixed a link in the documentation.
140. Manvendra Singh fixed a typo in the documentation.
141. ziggurat29 fixed an MSVC warning.
142. Sylvain Corlay added code to avoid an issue with MSVC.

143. mefyl fixed a bug when JSON was parsed from an input stream.
144. Millian Poquet allowed to install the library via Meson.
145. Michael Behrns-Miller found an issue with a missing namespace.
146. Nasztanovics Ferenc fixed a compilation issue with libc 2.12.
147. Andreas Schwab fixed the endian conversion.
148. Mark-Dunning fixed a warning in MSVC.
149. Gareth Sylvester-Bradley added `operator/` for JSON Pointers.
150. John-Mark noted a missing header.
151. Vitaly Zaitsev fixed compilation with GCC 9.0.
152. Laurent Stacul fixed compilation with GCC 9.0.
153. Ivor Wanders helped to reduce the CMake requirement to version 3.1.
154. njlr updated the Buckaroo instructions.
155. Lion fixed a compilation issue with GCC 7 on CentOS.
156. Isaac Nickaein improved the integer serialization performance and implemented the `contains()` function.
157. past-due suppressed an unfixable warning.
158. Elvis Oric improved Meson support.
159. Matěj Plch fixed an example in the README.
160. Mark Beckwith fixed a typo.
161. scinart fixed bug in the serializer.
162. Patrick Boettcher implemented `push_back()` and `pop_back()` for JSON Pointers.
163. Bruno Oliveira added support for Conda.
164. Michele Caini fixed links in the README.
165. Hani documented how to install the library with NuGet.
166. Mark Beckwith fixed a typo.
167. yann-morin-1998 helped to reduce the CMake requirement to version 3.1.
168. Konstantin Podsvirov maintains a package for the MSYS2 software distro.
169. remyabel added GNUInstallDirs to the CMake files.
170. Taylor Howard fixed a unit test.
171. Gabe Ron implemented the `to_string` method.
172. Watal M. Iwasaki fixed a Clang warning.
173. Viktor Kirilov switched the unit tests from Catch to doctest
174. Juncheng E fixed a typo.
175. tete17 fixed a bug in the `contains` function.
176. Xav83 fixed some cppcheck warnings.
177. 0xflotus fixed some typos.
178. Christian Deneke added a const version of `json_pointer::back`.
179. Julien Hamaide made the `items()` function work with custom string types.
180. Evan Nemerson updated fixed a bug in Hedley and updated this library accordingly.

181. Florian Pigorsch fixed a lot of typos.
182. Camille Bégué fixed an issue in the conversion from `std::pair` and `std::tuple` to `json`.
183. Anthony VH fixed a compile error in an enum deserialization.
184. Yuriy Vountesmery noted a subtle bug in a preprocessor check.
185. Chen fixed numerous issues in the library.
186. Antony Kellermann added a CI step for GCC 10.1.
187. Alex fixed an MSVC warning.
188. Rainer proposed an improvement in the floating-point serialization in CBOR.
189. Francois Chabot made performance improvements in the input adapters.
190. Arthur Sonzogni documented how the library can be included via `FetchContent`.
191. Rimas Misevičius fixed an error message.
192. Alexander Myasnikov fixed some examples and a link in the README.
193. Hubert Chathi made CMake's version config file architecture-independent.
194. OmnipotentEntity implemented the binary values for CBOR, MessagePack, BSON, and UBJSON.
195. ArtemSarmini fixed a compilation issue with GCC 10 and fixed a leak.
196. Evgenii Sopov integrated the library to the wsjcpp package manager.
197. Sergey Linev fixed a compiler warning.
198. Miguel Magalhães fixed the year in the copyright.
199. Gareth Sylvester-Bradley fixed a compilation issue with MSVC.
200. Alexander “weej” Jones fixed an example in the README.
201. Antoine Cœur fixed some typos in the documentation.
202. jothepro updated links to the Hunter package.
203. Dave Lee fixed link in the README.
204. Joël Lamotte added instruction for using Build2's package manager.
205. Paul Jurczak fixed an example in the README.
206. Sonu Lohani fixed a warning.
207. Carlos Gomes Martinho updated the Conan package source.
208. Konstantin Podsvirov fixed the MSYS2 package documentation.
209. Tridacnid improved the CMake tests.
210. Michael fixed MSVC warnings.
211. Quentin Barbarat fixed an example in the documentation.
212. XyFreak fixed a compiler warning.
213. TotalCaesar659 fixed links in the README.
214. Tanuj Garg improved the fuzzer coverage for UBSAN input.
215. AODQ fixed a compiler warning.
216. jwittbrodt made `NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE` inline.
217. pfeatherstone improved the upper bound of arguments of the `NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE`/`NLOHMANN_DEFINE_TYPE_INTRUSIVE` macros.
218. Jan Procházka fixed a bug in the CBOR parser for binary and string values.
219. T0b1-iOS fixed a bug in the new hash implementation.
220. Matthew Bauer adjusted the CBOR writer to create tags for binary subtypes.
221. gatopeich implemented an ordered map container for `nlohmann::ordered_json`.
222. Érico Nogueira Rolim added support for pkg-config.
223. KonanM proposed an implementation for the `NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE`/`NLOHMANN_DEFINE_TYPE_INTRUSIVE` macros.
224. Guillaume Racicot implemented `string_view` support and allowed C++20 support.

225. Alex Reinking improved CMake support for `FetchContent`.
226. Hannes Domani provided a GDB pretty printer.
227. Lars Wirzenius reviewed the README file.
228. Jun Jie fixed a compiler path in the CMake scripts.
229. Ronak Buch fixed typos in the documentation.
230. Alexander Karzhenkov fixed a move constructor and the Travis builds.
231. Leonardo Lima added CPM.Cmake support.
232. Joseph Blackman fixed a warning.
233. Yaroslav updated doctest and implemented unit tests.
234. Martin Stump fixed a bug in the CMake files.
235. Jaakko Moisio fixed a bug in the input adapters.
236. bl-ue fixed some Markdown issues in the README file.
237. William A. Wieselquist fixed an example from the README.
238. abbaswasim fixed an example from the README.
239. Remy Jette fixed a warning.
240. Fraser fixed the documentation.
241. Ben Beasley updated doctest.
242. Doron Behar fixed pkg-config.pc.
243. raduteo fixed a warning.
244. David Pfahler added the possibility to compile the library without I/O support.
245. Morten Fyhn Amundsen fixed a typo.
246. jpl-mac allowed to treat the library as a system header in CMake.
247. Jason Dsouza fixed the indentation of the CMake file.
248. offa added a link to Conan Center to the documentation.
249. TotalCaesar659 updated the links in the documentation to use HTTPS.
250. Rafail Giavrimis fixed the Google Benchmark default branch.
251. Louis Dionne fixed a conversion operator.
252. justanotheranonymoususer made the examples in the README more consistent.
253. Finkman suppressed some `-Wfloat-equal` warnings.
254. Ferry Huberts fixed `-Wswitch-enum` warnings.
255. Arseniy Terekhin made the GDB pretty-printer robust against unset variable names.
256. Amir Masoud Abdol updated the Homebrew command as nlohmann/json is now in homebrew-core.
257. Hallot fixed some `-Wextra-semi-stmt warnings`.
258. Giovanni Cerretani fixed `-Wunused` warnings on `JSON_DIAGNOSTICS`.
259. Bogdan Popescu hosts the docset for offline documentation viewers.
260. Carl Smedstad fixed an assertion error when using `JSON_DIAGNOSTICS`.
261. miikka75 provided an important fix to compile C++17 code with Clang 9.
262. Maarten Becker fixed a warning for shadowed variables.
263. Cristi Vîjdea fixed typos in the `operator[]` documentation.
264. Alex Beregszaszi fixed spelling mistakes in comments.
265. Dirk Stolle fixed typos in documentation.
266. Daniel Albuschat corrected the parameter name in the `parse` documentation.
267. Prince Mendiratta fixed a link to the FAQ.
268. Florian Albrechtskirchinger implemented `std::string_view` support for object keys and made dozens of other improvements.
269. Qianqian Fang implemented the Binary JData (BJData) format.

270. pketelsen added macros `NLOHMANN_DEFINE_TYPE_INTRUSIVE_WITH_DEFAULT` and `NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE_WITH_DEFAULT`.
271. DarkZeros adjusted to code to not clash with Arduino defines.
272. flagarde fixed the output of `meta()` for MSVC.
273. Giovanni Cerretani fixed a check for `std::filesystem`.
274. Dimitris Apostolou fixed a typo.
275. Ferry Huberts fixed a typo.
276. Michael Nosthoff fixed a typo.
277. JungHoon Lee fixed a typo.
278. Faruk D. fixed the CITATION.CFF file.
279. Andrea Cocito added a clarification on macro usage to the documentation.
280. Krzysiek Karbowiak refactored the tests to use `CHECK_THROWS_WITH_AS`.
281. Chaoqi Zhang fixed a typo.
282. ivanovmp fixed a whitespace error.
283. KsaNL fixed a build error when including `<windows.h>`.
284. Andrea Pappacoda moved `.pc` and `.cmake` files to `share` directory.
285. Wolf Vollprecht added the `patch_inplace` function.
286. Jake Zimmerman highlighted common usage patterns in the README file.
287. NN added the Visual Studio output directory to `.gitignore`.
288. Romain Reignier improved the performance the vector output adapter.
289. Mike fixed the `std::iterator_traits`.
290. Richard Hozák added macro `JSON_NO_ENUM` to disable default enum conversions.
291. vakokako fixed tests when compiling with C++20.
292. Alexander “weej” Jones fixed an example in the README.
293. Eli Schwartz added more files to the `include.zip` archive.
294. Kevin Lu fixed a compilation issue when typedefs with certain names were present.
295. Trevor Hickey improved the description of an example.
296. Jef LeCompte updated the year in the README file.
297. Alexandre Hamez fixed a warning.
298. Maninderpal Badhan fixed a typo.
299. kevin-- added a note to an example in the README file.
300. I fixed a typo.
301. Gregorio Litenstein fixed the Clang detection.
302. Andreas Smas added a Doozer badge.
303. WanCW fixed the string conversion with Clang.
304. zhaohuaxishi fixed a Doxygen error.
305. emvivre removed an invalid parameter from CMake.
306. Tobias Hermann fixed a link in the README file.
307. Michael fixed a warning.
308. Ryan Mulder added `ensure_ascii` to the `dump` function.
309. Muri Nicanor fixed the `sed` discovery in the Makefile.
310. David Avedissian implemented SFINAE-friendly `iterator_traits`.
311. AQNOUCH Mohammed fixed a typo in the README.
312. Gareth Sylvester-Bradley added `operator/=` and `operator/` to construct JSON pointers.
313. Michael Macnair added support for afl-fuzz testing.
314. Berkus Decker fixed a typo in the README.

315. Illia Polishchuk improved the CMake testing.
316. Ikko Ashimine fixed a typo.
317. Raphael Grimm added the possibility to define a custom base class.
318. tocic fixed typos in the documentation.
319. Vertexwahn added Bazel build support.
320. Dirk Stolle fixed typos in the documentation.
321. DavidKorczynski added a CIFuzz CI GitHub action.
322. Finkman fixed the debug pretty-printer.
323. Florian Segginger bumped the years in the README.
324. haadfida cleaned up the badges of used services.
325. Arsen Arsenović fixed a build error.
326. theevilone45 fixed a typo in a CMake file.
327. Sergei Trofimovich fixed the custom allocator support.
328. Joyce fixed some security issues in the GitHub workflows.
329. Nicolas Jakob add vcpkg version badge.
330. Tomerkm added tests.
331. No. fixed the use of `get<>` calls.
332. taro fixed a typo in the `CODEOWNERS` file.
333. Ikko Eltociear Ashimine fixed a typo.
334. Felix Yan fixed a typo in the README.
335. HO-COOH fixed a parentheses in the documentation.
336. Ivor Wanders fixed the examples to catch exception by `const&`.
337. miny1233 fixed a parentheses in the documentation.
338. tomalakgeretkal fixed a compilation error.
339. alferov fixed a compilation error.
340. Craig Scott fixed a deprecation warning in CMake.
341. Vyacheslav Zhdanovskiy added macros for serialization-only types.
342. Mathieu Westphal fixed typos.
343. scribam fixed the MinGW workflow.
344. Aleksei Sapitskii added support for Apple's Swift Package Manager.
345. Benjamin Buch fixed the installation path in CMake.
346. Colby Haskell clarified the parse error message in case a file cannot be opened.

Thanks a lot for helping out! Please let me know if I forgot someone.

13 Used third-party tools

The library itself consists of a single header file licensed under the MIT license. However, it is built, tested, documented, and whatnot using a lot of third-party tools and services. Thanks a lot!

- **amalgamate.py - Amalgamate C source and header files** to create a single header file
- **American fuzzy lop** for fuzz testing
- **AppVeyor** for continuous integration on Windows
- **Artistic Style** for automatic source code indentation
- **Clang** for compilation with code sanitizers
- **CMake** for build automation
- **Codacy** for further code analysis

- **Coveralls** to measure code coverage
- **Coverity Scan** for static analysis
- **cppcheck** for static analysis
- **doctest** for the unit tests
- **GitHub Changelog Generator** to generate the ChangeLog
- **Google Benchmark** to implement the benchmarks
- **Hedley** to avoid re-inventing several compiler-agnostic feature macros
- **lcov** to process coverage information and create an HTML view
- **libFuzzer** to implement fuzz testing for OSS-Fuzz
- **Material for MkDocs** for the style of the documentation site
- **MkDocs** for the documentation site
- **OSS-Fuzz** for continuous fuzz testing of the library (project repository)
- **Probot** for automating maintainer tasks such as closing stale issues, requesting missing information, or detecting toxic comments.
- **Valgrind** to check for correct memory management

14 Notes

Character encoding

The library supports **Unicode input** as follows:

- Only **UTF-8** encoded input is supported which is the default encoding for JSON according to RFC 8259.
- `std::u16string` and `std::u32string` can be parsed, assuming UTF-16 and UTF-32 encoding, respectively. These encodings are not supported when reading from files or other input containers.
- Other encodings such as Latin-1 or ISO 8859-1 are **not** supported and will yield parse or serialization errors.
- Unicode noncharacters will not be replaced by the library.
- Invalid surrogates (e.g., incomplete pairs such as `\uDEAD`) will yield parse errors.
- The strings stored in the library are UTF-8 encoded. When using the default string type (`std::string`), note that its length/size functions return the number of stored bytes rather than the number of characters or glyphs.
- When you store strings with different encodings in the library, calling `dump()` may throw an exception unless `json::error_handler_t::replace` or `json::error_handler_t::ignore` are used as error handlers.
- To store wide strings (e.g., `std::wstring`), you need to convert them to a UTF-8 encoded `std::string` before, see an example.

Comments in JSON

This library does not support comments by default. It does so for three reasons:

1. Comments are not part of the JSON specification. You may argue that `//` or `/* */` are allowed in JavaScript, but JSON is not JavaScript.

2. This was not an oversight: Douglas Crockford wrote on this in May 2012:

   > I removed comments from JSON because I saw people were using them to hold parsing directives, a practice which would have destroyed interoperability. I know that the lack of

> comments makes some people sad, but it shouldn't.
>
> Suppose you are using JSON to keep configuration files, which you would like to annotate. Go ahead and insert all the comments you like. Then pipe it through JSMin before handing it to your JSON parser.

3. It is dangerous for interoperability if some libraries would add comment support while others don't. Please check The Harmful Consequences of the Robustness Principle on this.

However, you can pass set parameter `ignore_comments` to true in the `parse` function to ignore `//` or `/* */` comments. Comments will then be treated as whitespace.

Order of object keys

By default, the library does not preserve the **insertion order of object elements**. This is standards-compliant, as the JSON standard defines objects as "an unordered collection of zero or more name/value pairs".

If you do want to preserve the insertion order, you can try the type `nlohmann::ordered_json`. Alternatively, you can use a more sophisticated ordered map like `tsl::ordered_map` (integration) or `nlohmann::fifo_map` (integration).

See the **documentation on object order** for more information.

Memory Release

We checked with Valgrind and the Address Sanitizer (ASAN) that there are no memory leaks.

If you find that a parsing program with this library does not release memory, please consider the following case, and it may be unrelated to this library.

Your program is compiled with glibc. There is a tunable threshold that glibc uses to decide whether to actually return memory to the system or whether to cache it for later reuse. If in your program you make lots of small allocations and those small allocations are not a contiguous block and are presumably below the threshold, then they will not get returned to the OS. Here is a related issue #1924.

Further notes

- The code contains numerous debug **assertions** which can be switched off by defining the preprocessor macro `NDEBUG`, see the documentation of `assert`. In particular, note `operator[]` implements **unchecked access** for const objects: If the given key is not present, the behavior is undefined (think of a dereferenced null pointer) and yields an assertion failure if assertions are switched on. If you are not sure whether an element in an object exists, use checked access with the `at()` function. Furthermore, you can define `JSON_ASSERT(x)` to replace calls to `assert(x)`. See the **documentation on runtime assertions** for more information.
- As the exact number type is not defined in the JSON specification, this library tries to choose the best fitting C++ number type automatically. As a result, the type `double` may be used to store numbers which may yield **floating-point exceptions** in certain rare situations if floating-point exceptions have been unmasked in the calling code. These exceptions are not caused by the library and need to be fixed in the calling code, such as by re-masking the exceptions prior to calling library functions.

- The code can be compiled without C++ **runtime type identification** features; that is, you can use the `-fno-rtti` compiler flag.
- **Exceptions** are used widely within the library. They can, however, be switched off with either using the compiler flag `-fno-exceptions` or by defining the symbol `JSON_NOEXCEPTION`. In this case, exceptions are replaced by `abort()` calls. You can further control this behavior by defining `JSON_THROW_USER` (overriding `throw`), `JSON_TRY_USER` (overriding `try`), and `JSON_CATCH_USER` (overriding `catch`). Note that `JSON_THROW_USER` should leave the current scope (e.g., by throwing or aborting), as continuing after it may yield undefined behavior. Note the explanatory `what()` string of exceptions is not available for MSVC if exceptions are disabled, see #2824. See the **documentation of exceptions** for more information.

15 Execute unit tests

To compile and run the tests, you need to execute

```
mkdir build
cd build
cmake .. -DJSON_BuildTests=On
cmake --build .
ctest --output-on-failure
```

Note that during the `ctest` stage, several JSON test files are downloaded from an external repository. If policies forbid downloading artifacts during testing, you can download the files yourself and pass the directory with the test files via `-DJSON_TestDataDirectory=path` to CMake. Then, no Internet connectivity is required. See issue #2189 for more information.

If the test suite is not found, several test suites will fail like this:

```
===============================================================================
json/tests/src/make_test_data_available.hpp:21:
TEST CASE:  check test suite is downloaded

json/tests/src/make_test_data_available.hpp:23: FATAL ERROR: REQUIRE(
utils::check_testsuite_downloaded() ) is NOT correct!
  values: REQUIRE( false )
  logged: Test data not found in 'json/cmake-build-debug/json_test_data'.
          Please execute target 'download_test_data' before running this test
suite.
          See <https://github.com/nlohmann/json#execute-unit-tests> for more
information.

===============================================================================
```

In case you have downloaded the library rather than checked out the code via Git, test `cmake_fetch_content_configure` will fail. Please execute `ctest -LE git_required` to skip these tests. See issue #2189 for more information.

Some tests change the installed files and hence make the whole process not reproducible. Please execute `ctest -LE not_reproducible` to skip these tests. See issue #2324 for more information. Furthermore, assertions must be switched off to ensure reproducible builds (see discussion 4494).

Note you need to call `cmake -LE "not_reproducible|git_required"` to exclude both labels. See issue #2596 for more information.

As Intel compilers use unsafe floating point optimization by default, the unit tests may fail. Use flag `/fp:precise` then.