Beside the examples below, you may want to check the [documentation](https://nlohmann.github.io/json/) where each function contains a separate code example (e.g., check out [emplace()](https://nlohmann.github.io/json/classnlohmann_1_1basic__json_a5338e282d1d02bed389d852dd670d98d.html#a5338e282d1d02bed389d852dd670d98d)). All [example files](https://github.com/nlohmann/json/tree/develop/doc/examples) can be compiled and executed on their own (e.g., file [emplace.cpp](https://github.com/nlohmann/json/blob/develop/doc/examples/emplace.cpp)).

**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()](https://nlohmann.github.io/json/classnlohmann_1_1basic__json_a9ad7ec0bc1082ed09d10900fbb20a21f.html#a9ad7ec0bc1082ed09d10900fbb20a21f) and [json::object()](https://nlohmann.github.io/json/classnlohmann_1_1basic__json_aaf509a7c029100d292187068f61c99b8.html#aaf509a7c029100d292187068f61c99b8) 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} });

**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 above example can also be expressed explicitly using [json::parse()](https://nlohmann.github.io/json/classnlohmann_1_1basic__json_afd4ef1ac8ad50a5894a9afebca69140a.html#afd4ef1ac8ad50a5894a9afebca69140a):

// parse explicitly

auto j3 = json::parse("{ \"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.get<std::string>();

// retrieve the string value (alternative when an 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.get<std::string>() << '\n';

// output of serialized value

std::cout << j\_string << " == " << serialized\_string << std::endl;

[.dump()](https://nlohmann.github.io/json/classnlohmann_1_1basic__json_a50ec80b02d0f3f51130d4abb5d1cfdc5.html#a50ec80b02d0f3f51130d4abb5d1cfdc5) always returns the serialized value, and [.get<std::string>()](https://nlohmann.github.io/json/classnlohmann_1_1basic__json_aa6602bb24022183ab989439e19345d08.html#aa6602bb24022183ab989439e19345d08) 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()](https://nlohmann.github.io/json/classnlohmann_1_1basic__json_a50ec80b02d0f3f51130d4abb5d1cfdc5.html#a50ec80b02d0f3f51130d4abb5d1cfdc5) 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 content is stored as contiguous byte sequence, for instance a std::vector<std::uint8\_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);

**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 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](https://github.com/nlohmann/json/blob/develop/include/nlohmann/detail/input/json_sax.hpp).

**STL-like access**

We designed the JSON class to behave just like an STL container. In fact, it satisfies the [**ReversibleContainer**](https://en.cppreference.com/w/cpp/named_req/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

const auto tmp = j[0].get<std::string>();

j[1] = 42;

bool foo = j.at(2);

// comparison

j == "[\"foo\", 1, true]"\_json; // true

// other stuff

j.size(); // 3 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.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");

**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}

**JSON Pointer and JSON Patch**

The library supports **JSON Pointer** ([RFC 6901](https://tools.ietf.org/html/rfc6901)) as alternative means to address structured values. On top of this, **JSON Patch** ([RFC 6902](https://tools.ietf.org/html/rfc6902)) allows to describe 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" }

// ]

**JSON Merge Patch**

The library supports **JSON Merge Patch** ([RFC 7386](https://tools.ietf.org/html/rfc7386)) 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"

// }

// }

**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](https://www.github.com/nlohmann/json/issues/958).

// strings

std::string s1 = "Hello, world!";

json js = s1;

auto s2 = js.get<std::string>();

// NOT RECOMMENDED

std::string s3 = js;

std::string s4;

s4 = js;

// Booleans

bool b1 = true;

json jb = b1;

auto b2 = jb.get<bool>();

// NOT RECOMMENDED

bool b3 = jb;

bool b4;

b4 = jb;

// numbers

int i = 42;

json jn = i;

auto f = jn.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"

**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:

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"].get<std::string>(),

j["address"].get<std::string>(),

j["age"].get<int>()

};

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

// 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.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 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 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](https://github.com/nlohmann/json/issues/1108) for errors that may occur otherwise.
* When using get<your\_type>(), your\_type **MUST** be [DefaultConstructible](https://en.cppreference.com/w/cpp/named_req/DefaultConstructible). (There is a way to bypass this requirement described later.)
* In function from\_json, use function [at()](https://nlohmann.github.io/json/classnlohmann_1_1basic__json_a93403e803947b86f4da2d1fb3345cf2c.html#a93403e803947b86f4da2d1fb3345cf2c) 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.

**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](https://en.cppreference.com/w/cpp/language/adl)).

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:

// 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.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](https://en.cppreference.com/w/cpp/named_req/MoveConstructible). You will need to specialize the adl\_serializer as well, but with a special from\_json overload:

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.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](https://github.com/nlohmann/json/blob/develop/test/src/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!

}

};

**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.get<TaskState>() == TS\_RUNNING);

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

json jPi = 3.14;

assert(jPi.get<TaskState>() == TS\_INVALID );

Just as in [Arbitrary Type Conversions](https://github.com/ericyonng/json#arbitrary-types-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 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.

**Binary formats (BSON, CBOR, MessagePack, and UBJSON)**

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](http://bsonspec.org/) (Binary JSON), [CBOR](http://cbor.io/) (Concise Binary Object Representation), [MessagePack](http://msgpack.org/), and [UBJSON](http://ubjson.org/) (Universal Binary JSON Specification) 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);