« ✖ Remove all ✖ ID (Book) = json libraries #nlohmann #rapidjson #boost.json

1 - 10 | 2 | 3 | 4 | 5 | 6 | 7 | › | » | 10

ID	Chapter	Brief	Links	Type	Target	Level	Text	Page	Pic	Connection
Introduction										
10736		**Json libraries compared - nlohmann rapidjson boost.json**	0	Introduction				5		

Comparing **nlohmann rapidjson boost.json**

▶ About this system

10729		**Personal rating**	0	Introduction				10		

Rating

Here is how I rate these libraries. If rating is low, that doesn't mean that it is bad. Libraries have reasons for how they are designed.
Adding a library is a big decision.
Code that will be used in several places but for some reason you notice that the code does not fit, it can be an extensive job to replace.
So therefore it is good to be aware of the pros and cons of what is added.

[Area _ _ _ _ _ _ _]	[nlohmann]	[RapidJSON]	[Boost.JSON]	[simdjson]
Speed	2	5	4	soon
Readable code	3	3	2	
Debuggable code	3	3	1	
Memory	2	4	3	
Documentation	4	4	2	
Features	5	4	3	
Simple to use	4	3	3	
License	MIT	MIT	Boost	

Comments

When to select out of the box (all libraries can be customized)

- Performance
  - Here I would select RapidJSON. RapidJSON are able to handle string pointers and that is a great feature when speed is required, will help to get more hits in cache for common strings. It is also very fast doing other type of operations.
- Simplicity
  - nlohmann. nlohmann is like it a part of stl. nlohmann is based on stl, it hasn't created it's own objects.
- Tiny overhead
  - Here Boost.JSON might be the one to select. It has a very thin layer to work with JSON. Boost.JSON can also be used with another JSON library if requirements need to be met without adding to much to application size.
- Only select one
  - RapidJSON is the best mix of different properties if you are only allowed to use one library.
- Features
  - in this group nlohmann is the most feature rich library.

nlohmann

10706		**Introduction to nlohmann json**	0	Introduction				2		

nlohmann json

Probably the most used C++ json library.
Why? It is known, very flexible and fast enough. With many users it is also safe. It is header only so easy to include in C++ projects.

`sizeof` for each json value is 16 bytes. It relies heavily on stl.

Debug nlohmann code is ok. Documentation in code is extensive, also generated documentation is in the code. Deep knowledge about C++ is required.

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

nlohmann on github (https://github.com/nlohmann/json)
nlohmann manual (https://nlohmann.github.io/json/)

ID	Chapter	Brief	Links	Type	Target	Level	Text	Page	Pic	Connection
10707		**Types type**	0	Documentation				4		

Json types

get type number

```
nlohmann::json jsonNumber = 10;
std::cout << (int)jsonNumber.type() << "\n"; // prints 5 ( nlohmann::value_t::number_unsigned )
```

See: **type**

print type nlohmann name

```
nlohmann::json jsonNumber = 10u;
switch( jsonNumber.type() )
{
   case nlohmann::detail::value_t::null : std::cout << "null" << "\n"; break;
   case nlohmann::detail::value_t::object : std::cout << "object" << "\n"; break;
   case nlohmann::detail::value_t::array : std::cout << "array" << "\n"; break;
   case nlohmann::detail::value_t::string : std::cout << "string" << "\n"; break;
   case nlohmann::detail::value_t::boolean : std::cout << "boolean" << "\n"; break;
   case nlohmann::detail::value_t::number_integer : std::cout << "number_integer" << "\n"; break;
   case nlohmann::detail::value_t::number_unsigned : std::cout << "number_unsigned" << "\n"; break;
   case nlohmann::detail::value_t::number_float : std::cout << "number_float" << "\n"; break;
   case nlohmann::detail::value_t::binary : std::cout << "binary" << "\n"; break;
   case nlohmann::detail::value_t::discarded :  std::cout << "discarded" << "\n"; break;
}
```

See: **type**

get specific value type

```
try
{
   nlohmann::json jsonNumber = 10u;
   std::string stringNumber = jsonNumber.get<std::string>();
}
catch( nlohmann::detail::type_error e )
{
   std::cout << e.what() << "\n";
}
// Output: [json.exception.type_error.302] type must be string, but is number
```

See: **get**

get type name from json object

```
nlohmann::json jsonObject = nlohmann::json::object();
std::cout << jsonObject.type_name() << "\n"; // prints object
```

See: **type_name**

10709		**Compare the [] operator with at()**	0	Sample				6		

ID	Chapter	Brief	Link	Type	Target	Level	Text	Page	Pic	Connection

[]() compared to at()

`operator[]` Creates items if items isn't found
`at()` throws out_of_range error if item isn't found

Same behaviour as `std::map` in stl.

```
try
{
   nlohmann::json jsonObject1 = nlohmann::json::object();
   nlohmann::json jsonObject2 = nlohmann::json::object();

   auto _item1 = jsonObject1[ "one" ];
   _item1 = jsonObject1[ "two" ];
   std::cout << jsonObject1.size() << "\n";
   auto _item2 = jsonObject2.at( "one" ); // <- throws
   _item2 = jsonObject2.at( "two" );
   std::cout << jsonObject2.size() << "\n";
}
catch( nlohmann::detail::out_of_range e )
{
   std::cout << e.what() << "\n";
}

try
{
   nlohmann::json jsonArray1 = { 0,1,2,3,4,5 };
   nlohmann::json jsonArray2 = { 0,1,2,3,4,5 };

   auto _item1 = jsonArray1[ 1 ];
   _item1 = jsonArray1[ 10 ]; // careful, array will grow
   std::cout << jsonArray1.size() << "\n"; // prints 11
   auto _item2 = jsonArray2.at( 1 );
   _item2 = jsonArray2.at( 10 ); // <- throws
   std::cout << jsonArray2.size() << "\n";
}
catch( nlohmann::detail::out_of_range e )
{
   std::cout << e.what() << "\n";
}
```

10708		**Add your own object to nlohmann**	0	Sample				8		

ID	Chapter	Brief	Link	Type	Target	Level	Text	Page ^	Pic	Connection

Make object compatibile with json

Adding `to_json` and `from_json` for objects simplifies use with nlohmann json object.
Format is `void to_json( nlohmann::json& j, const object_name& object_variable )` and `void from_json( nlohmann::json& j, object_name& object_variable )`.

```
class location
{
public:
   std::string m_stringCity;
   std::string m_stringCountry;
   int m_iSize;
};

// write to json
void to_json( nlohmann::json& j, const location& l )
{
   j = nlohmann::json{ {"city", l.m_stringCity}, {"country", l.m_stringCountry}, {"size", l.m_iSize} };
}

// read from json
void from_json( nlohmann::json& j, location& l )
{
   j.at( "city" ).get_to( l.m_stringCity );
   j.at( "country" ).get_to( l.m_stringCountry );
   j.at( "size" ).get_to( l.m_iSize );
}

void main()
{
   location locationSweden = { "gothenburgh", "sweden", 10 };
   nlohmann::json jsonLocation = locationSweden; // <-- Magic!!
}
```

See **at**

10699		**nlohmann version**	0	Sample				10		

ID	Chapter	Brief	Link	Type	Target	Level	Text	Page ^	Pic	Connection

Get version with meta

return nlohmann version with **meta** . **dump** is used to convert to string.

Get version information from nlohmann json library and dump it

```
   std::string meta()
   {
      nlohmann::json jsonMeta;
      auto _json = jsonMeta.meta();
      std::cout << "JSON\n" << _json.dump() << std::endl;
      std::cout << "JSON with indentation set to 3\n" << _json.dump( 3 ) << std::endl;

      std::string stringDump = nlohmann::to_string( _json );
      if( _json.dump() == stringDump ) std::cout << "COMPARE:  _json.dump() == nlohmann::to_string( _json )" << std::endl;
      else std::cout << "COMPARE: _json.dump() != nlohmann::to_string( _json )" << std::endl;

      return stringDump;
   }
```

Output from meta function

```
JSON
{"compiler":{"c++":"199711","family":"msvc","version":1928},"copyright":"(C) 2013-2020 Niels Lohmann","name":"JSON for Modern C++","p
latform":"win32","url":"https://github.com/nlohmann/json","version":{"major":3,"minor":9,"patch":1,"string":"3.9.1"}}
JSON with indentation set to 3
{
   "compiler": {
      "c++": "199711",
      "family": "msvc",
      "version": 1928
   },
   "copyright": "(C) 2013-2020 Niels Lohmann",
   "name": "JSON for Modern C++",
   "platform": "win32",
   "url": "https://github.com/nlohmann/json",
   "version": {
      "major": 3,
      "minor": 9,
      "patch": 1,
      "string": "3.9.1"
   }
}
COMPARE:  _json.dump() == nlohmann::to_string( _json )
```

10700		**Read json from file std::ifstream**	0	Sample				20		

ID	Chapter	Brief	Link	Type	Target	Level	Text	Page	Pic	Connection

Read json from file

Using `std::ifstream` to read file

No boilerplate

```
nlohmann::json jsonRead;
std::ifstream streamJson;
streamJson.open( stringFileName );
streamJson >> jsonRead;
```

Read json information from file. Two methods, the later checks for errors.

```
void load( std::string_view stringFileName )
{
    nlohmann::json jsonRead;
    std::ifstream streamJson( stringFileName );
    streamJson >> jsonRead;

    // std::cout << jsonRead.dump( 3 ) << std::endl;
}

// handle errors
std::pair<bool, std::string> load_safe( std::string_view stringFileName )
{
    try
    {
        nlohmann::json jsonRead;
        std::ifstream streamJson;
        streamJson.exceptions( std::ifstream::badbit | std::ifstream::failbit );
        streamJson.open( stringFileName );
        streamJson >> jsonRead;
        //std::cout << jsonRead.dump( 3 ) << std::endl;
    }
    catch( const std::ifstream::failure e )
    {
        std::ostringstream  stringError;
        stringError << "Failed to load \"" << stringFileName.data() << "\" !";
        return std::pair<bool, std::string>( false, stringError.str() );
    }
    catch( const nlohmann::detail::parse_error e )
    {
        return std::pair<bool, std::string>( false, e.what() );
    }

    return std::pair<bool, std::string>( true, std::string() );
}
```

See: **operator>>**

10701		**Iterate key value pairs in object**	0	Sample				30		

ID	Chapter	Brief	Link	Type	Target	Level	Text	Page	Pic	Connection

Iterate key value pairs in object

Object values in nlohmann can be iterated using `begin` and `end`. If key name isn't known or if you need to get value using index, then iterate values may solve the problem.

Two methods iterating array with objects

```
std::pair<bool, std::string> iterate_items()
{
   constexpr std::string_view stringJson = R"([ {"k1": "v1"}, {"k2": "v2"}, {"k3": "v3"}, {"k4": "v4"} ])";
   nlohmann::json jsonIterate;
   try
   {
      auto _json = jsonIterate.parse( std::begin( stringJson ), std::end( stringJson ) );

      for( auto it : _json ) { std::cout << nlohmann::to_string( it ); }
      std::cout << std::endl;

      int_fast32_t iCount = 0;
      for( auto it = std::begin( _json ); it != std::end( _json ); it++ )
      {
         if( iCount != 0 ) std::cout << ", ";
         auto itBegin = std::begin( *it );
         std::cout << itBegin.key() << " : " << itBegin.value();
         iCount++;
      }
      std::cout << std::endl;
   }
   catch( const nlohmann::detail::parse_error e )
   {
      return std::pair<bool, std::string>( true, e.what() );
   }

   return std::pair<bool, std::string>( true, std::string() );
}
```

See: **key** , **value**

10702		**Position iterator in object or array**	0	Sample				40		

ID	Chapter	Brief	Link	Type	Target	Level	Text	Page ^	Pic	Connection

Position iterator in object or array

Moving iterator in array or object types

```
std::pair<bool, std::string> position()
{
   constexpr std::string_view stringJson =
R"({
"one": 1,
"two": 2,
"tree": { "objects": {"k1": "v1", "k2": "v2"} },
"four": 4,
"five": [0,1,2,3,4,5,6,7,8,9],
"six": "string value"
})";
   nlohmann::json jsonData;
   try
   {
      auto _json = jsonData.parse( stringJson );

      auto first = _json.front();     std::cout << nlohmann::to_string( first ) << "\n";
      auto back = _json.back();       std::cout << nlohmann::to_string( back ) << "\n";
      auto position = _json.begin(); std::advance( position, 2 ); std::cout << nlohmann::to_string( *position ) << "\n";
      std::advance( position, 1 ); std::cout << nlohmann::to_string( *position ) << "\n";
      std::advance( position, 1 ); std::cout << nlohmann::to_string( *position ) << "\n";

      auto five = _json[ "five" ];
      auto it = five.begin(); it++; it = it + 3; std::cout << nlohmann::to_string( *it ) << "\n";

   }
   catch( const nlohmann::detail::parse_error e )
   {
      return std::pair<bool, std::string>( true, e.what() );
   }

   return std::pair<bool, std::string>( true, std::string() );
}
```

See: **front** , **back** , **parse**