LatticeYangMills

Generated by Doxygen 1.8.11

Contents

Chapter 1

Deprecated List

Member nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::operator<< (basic_json &j, std::istream &i)

This stream operator is deprecated and will be removed in a future version of the library. Please use operator>>(std::istream&, basic_json&) instead; that is, replace calls like `j << i;` with `i >> j;`.

Member nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::operator>> (const basic_json &j, std::ostream &o)

This stream operator is deprecated and will be removed in a future version of the library. Please use operator<<(std::ostream&, const basic_json&) instead; that is, replace calls like `j >> o;` with `o << j;`.

Chapter 2

Namespace Index

2.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 5

Namespace Documentation

5.1 nlohmann Namespace Reference

namespace for Niels Lohmann

Namespaces

- detail

  unnamed namespace with internal helper functions

Classes

- struct adl_serializer

  default JSONSerializer template argument
- class basic_json

  a class to store JSON values
- class json_pointer

  JSON Pointer.

Typedefs

- using json = basic_json<>

  default JSON class

Functions

- bool **operator==** (json_pointer const &lhs, json_pointer const &rhs) noexcept
- bool **operator!=** (json_pointer const &lhs, json_pointer const &rhs) noexcept

5.1.1 Detailed Description

namespace for Niels Lohmann

See also

https://github.com/nlohmann

Since

version 1.0.0

5.1.2 Typedef Documentation

5.1.2.1 using nlohmann::json = typedef basic_json<>

default JSON class

This type is the default specialization of the basic_json class which uses the standard template types.

Since

version 1.0.0

5.2 nlohmann::detail Namespace Reference

unnamed namespace with internal helper functions

Classes

- class binary_reader
  deserialization of CBOR and MessagePack values
- class binary_writer
  serialization to CBOR and MessagePack values
- struct conjunction
- struct conjunction< B1 >
- struct conjunction< B1, Bn... >
- class exception
  general exception of the basic_json class
- struct external_constructor
- struct external_constructor< value_t::array >
- struct external_constructor< value_t::boolean >
- struct external_constructor< value_t::number_float >
- struct external_constructor< value_t::number_integer >
- struct external_constructor< value_t::number_unsigned >
- struct external_constructor< value_t::object >
- struct external_constructor< value_t::string >
- struct from_json_fn
- struct has_from_json

- struct has_non_default_from_json
- struct has_to_json
- struct index_sequence
- class input_adapter
- struct input_adapter_protocol

  abstract input adapter interface
- class input_buffer_adapter

  input adapter for buffer input
- class input_stream_adapter
- struct internal_iterator

  an iterator value
- class invalid_iterator

  exception indicating errors with iterators
- struct is_basic_json
- struct is_basic_json< NLOHMANN_BASIC_JSON_TPL >
- struct is_basic_json_nested_type
- struct is_compatible_array_type
- struct is_compatible_integer_type
- struct is_compatible_integer_type_impl
- struct is_compatible_integer_type_impl< true, RealIntegerType, CompatibleNumberIntegerType >
- struct is_compatible_object_type
- struct is_compatible_object_type_impl
- struct is_compatible_object_type_impl< true, RealType, CompatibleObjectType >
- class iter_impl

  a template for a bidirectional iterator for the basic_json class
- class iteration_proxy

  proxy class for the iterator_wrapper functions
- class json_ref
- class json_reverse_iterator

  a template for a reverse iterator class
- class lexer

  lexical analysis
- struct make_index_sequence
- struct make_index_sequence< 0 >
- struct make_index_sequence< 1 >
- struct merge_and_renumber
- struct merge_and_renumber< index_sequence< I1... >, index_sequence< I2... > >
- struct negation
- class other_error

  exception indicating other library errors
- class out_of_range

  exception indicating access out of the defined range
- class output_adapter
- struct output_adapter_protocol

  abstract output adapter interface
- class output_stream_adapter

  output adapter for output streams
- class output_string_adapter

  output adapter for basic_string
- class output_vector_adapter

  output adapter for byte vectors
- class parse_error

exception indicating a parse error

- class parser

  syntax analysis
- class primitive_iterator_t

  an iterator for primitive JSON types
- struct priority_tag
- struct priority_tag< 0 >
- class serializer
- struct static_const
- struct to_json_fn
- class type_error

  exception indicating executing a member function with a wrong type

Typedefs

- template<bool B, typename T = void>
  using **enable_if_t** = typename std::enable_if< B, T >::type
- template<typename T >
  using **uncvref_t** = typename std::remove_cv< typename std::remove_reference< T >::type >::type
- template<typename... Ts>
  using **index_sequence_for** = make_index_sequence< sizeof...(Ts)>
- using input_adapter_t = std::shared_ptr< input_adapter_protocol >

  a type to simplify interfaces
- template<typename CharType >
  using output_adapter_t = std::shared_ptr< output_adapter_protocol< CharType >>

  a type to simplify interfaces

Enumerations

- enum value_t : uint8_t {
  value_t::null, value_t::object, value_t::array, value_t::string,
  value_t::boolean, value_t::number_integer, value_t::number_unsigned, value_t::number_float,
  value_t::discarded }

  the JSON type enumeration

Functions

- bool operator< (const value_t lhs, const value_t rhs) noexcept

  comparison operator for JSON types
- **NLOHMANN_JSON_HAS_HELPER** (mapped_type)
- **NLOHMANN_JSON_HAS_HELPER** (key_type)
- **NLOHMANN_JSON_HAS_HELPER** (value_type)
- **NLOHMANN_JSON_HAS_HELPER** (iterator)
- template<typename BasicJsonType , typename T , enable_if_t< std::is_same< T, typename BasicJsonType::boolean_t >::value, int > = 0>
  void **to_json** (BasicJsonType &j, T b) noexcept
- template<typename BasicJsonType , typename CompatibleString , enable_if_t< std::is_constructible< typename BasicJsonType::string_t, CompatibleString >::value, int > = 0>
  void **to_json** (BasicJsonType &j, const CompatibleString &s)
- template<typename BasicJsonType >
  void **to_json** (BasicJsonType &j, typename BasicJsonType::string_t &&s)

- template<typename BasicJsonType , typename FloatType , enable_if_t< std::is_floating_point< FloatType >::value, int > = 0>
  void **to_json** (BasicJsonType &j, FloatType val) noexcept
- template<typename BasicJsonType , typename CompatibleNumberUnsignedType , enable_if_t< is_compatible_integer_type< typename BasicJsonType::number_unsigned_t, CompatibleNumberUnsignedType >::value, int > = 0>
  void **to_json** (BasicJsonType &j, CompatibleNumberUnsignedType val) noexcept
- template<typename BasicJsonType , typename CompatibleNumberIntegerType , enable_if_t< is_compatible_integer_type< typename BasicJsonType::number_integer_t, CompatibleNumberIntegerType >::value, int > = 0>
  void **to_json** (BasicJsonType &j, CompatibleNumberIntegerType val) noexcept
- template<typename BasicJsonType , typename EnumType , enable_if_t< std::is_enum< EnumType >::value, int > = 0>
  void **to_json** (BasicJsonType &j, EnumType e) noexcept
- template<typename BasicJsonType >
  void **to_json** (BasicJsonType &j, const std::vector< bool > &e)
- template<typename BasicJsonType , typename CompatibleArrayType , enable_if_t< is_compatible_array_type< BasicJsonType, CompatibleArrayType >::value orstd::is_same< typename BasicJsonType::array_t, CompatibleArrayType >::value, int > = 0>
  void **to_json** (BasicJsonType &j, const CompatibleArrayType &arr)
- template<typename BasicJsonType , typename T , enable_if_t< std::is_convertible< T, BasicJsonType >::value, int > = 0>
  void **to_json** (BasicJsonType &j, std::valarray< T > arr)
- template<typename BasicJsonType >
  void **to_json** (BasicJsonType &j, typename BasicJsonType::array_t &&arr)
- template<typename BasicJsonType , typename CompatibleObjectType , enable_if_t< is_compatible_object_type< BasicJsonType, CompatibleObjectType >::value, int > = 0>
  void **to_json** (BasicJsonType &j, const CompatibleObjectType &obj)
- template<typename BasicJsonType >
  void **to_json** (BasicJsonType &j, typename BasicJsonType::object_t &&obj)
- template<typename BasicJsonType , typename T , std::size_t N, enable_if_t< not std::is_constructible< typename BasicJsonType↩
  ::string_t, T(&)[N]>::value, int > = 0>
  void **to_json** (BasicJsonType &j, T(&arr)[N])
- template<typename BasicJsonType , typename... Args>
  void **to_json** (BasicJsonType &j, const std::pair< Args... > &p)
- template<typename BasicJsonType , typename Tuple , std::size_t... Idx>
  void **to_json_tuple_impl** (BasicJsonType &j, const Tuple &t, index_sequence< Idx... >)
- template<typename BasicJsonType , typename... Args>
  void **to_json** (BasicJsonType &j, const std::tuple< Args... > &t)
- template<typename BasicJsonType , typename ArithmeticType , enable_if_t< std::is_arithmetic< ArithmeticType >::value andnot std::is_same< ArithmeticType, typename BasicJsonType::boolean_t >::value, int > = 0>
  void **get_arithmetic_value** (const BasicJsonType &j, ArithmeticType &val)
- template<typename BasicJsonType >
  void **from_json** (const BasicJsonType &j, typename BasicJsonType::boolean_t &b)
- template<typename BasicJsonType >
  void **from_json** (const BasicJsonType &j, typename BasicJsonType::string_t &s)
- template<typename BasicJsonType >
  void **from_json** (const BasicJsonType &j, typename BasicJsonType::number_float_t &val)
- template<typename BasicJsonType >
  void **from_json** (const BasicJsonType &j, typename BasicJsonType::number_unsigned_t &val)
- template<typename BasicJsonType >
  void **from_json** (const BasicJsonType &j, typename BasicJsonType::number_integer_t &val)
- template<typename BasicJsonType , typename EnumType , enable_if_t< std::is_enum< EnumType >::value, int > = 0>
  void **from_json** (const BasicJsonType &j, EnumType &e)
- template<typename BasicJsonType >
  void **from_json** (const BasicJsonType &j, typename BasicJsonType::array_t &arr)
- template<typename BasicJsonType , typename T , typename Allocator , enable_if_t< std::is_convertible< BasicJsonType, T >::value, int > = 0>
  void **from_json** (const BasicJsonType &j, std::forward_list< T, Allocator > &l)
- template<typename BasicJsonType , typename T , enable_if_t< std::is_convertible< BasicJsonType, T >::value, int > = 0>
  void **from_json** (const BasicJsonType &j, std::valarray< T > &l)
- template<typename BasicJsonType , typename CompatibleArrayType >
  void **from_json_array_impl** (const BasicJsonType &j, CompatibleArrayType &arr, priority_tag< 0 >)

- template<typename BasicJsonType , typename CompatibleArrayType >
  auto **from_json_array_impl** (const BasicJsonType &j, CompatibleArrayType &arr, priority_tag< 1 >) -> decltype(arr.reserve(std::declval< typename CompatibleArrayType::size_type >()), void())
- template<typename BasicJsonType , typename T , std::size_t N>
  void **from_json_array_impl** (const BasicJsonType &j, std::array< T, N > &arr, priority_tag< 2 >)
- template<typename BasicJsonType , typename CompatibleArrayType , enable_if_t< is_compatible_array_type< BasicJsonType, CompatibleArrayType >::value andstd::is_convertible< BasicJsonType, typename CompatibleArrayType::value_type >::value andnot std::is_same< typename BasicJsonType::array_t, CompatibleArrayType >::value, int > = 0>
  void **from_json** (const BasicJsonType &j, CompatibleArrayType &arr)
- template<typename BasicJsonType , typename CompatibleObjectType , enable_if_t< is_compatible_object_type< BasicJsonType, CompatibleObjectType >::value, int > = 0>
  void **from_json** (const BasicJsonType &j, CompatibleObjectType &obj)
- template<typename BasicJsonType , typename ArithmeticType , enable_if_t< std::is_arithmetic< ArithmeticType >::value andnot std::is_same< ArithmeticType, typename BasicJsonType::number_unsigned_t >::value andnot std::is_same< ArithmeticType, typename BasicJsonType::number_integer_t >::value andnot std::is_same< ArithmeticType, typename BasicJsonType::number_float_t >::value andnot std::is_same< ArithmeticType, typename BasicJsonType::boolean_t >::value, int > = 0>
  void **from_json** (const BasicJsonType &j, ArithmeticType &val)
- template<typename BasicJsonType , typename A1 , typename A2 >
  void **from_json** (const BasicJsonType &j, std::pair< A1, A2 > &p)
- template<typename BasicJsonType , typename Tuple , std::size_t... Idx>
  void **from_json_tuple_impl** (const BasicJsonType &j, Tuple &t, index_sequence< Idx... >)
- template<typename BasicJsonType , typename... Args>
  void **from_json** (const BasicJsonType &j, std::tuple< Args... > &t)

5.2.1 Detailed Description

unnamed namespace with internal helper functions

This namespace collects some functions that could not be defined inside the basic_json class.

Since

version 2.1.0

5.2.2 Enumeration Type Documentation

5.2.2.1 enum nlohmann::detail::value_t : uint8_t [strong]

the JSON type enumeration

This enumeration collects the different JSON types. It is internally used to distinguish the stored values, and the functions basic_json::is_null(), basic_json::is_object(), basic_json::is_array(), basic_json::is_string(), basic_json::is_boolean(), basic_json::is_number() (with basic_json::is_number_integer(), basic_json::is_number_unsigned(), and basic_json::is_number_float()), basic_json::is_discarded(), basic_json::is_primitive(), and basic_json::is_structured() rely on it.

Note

There are three enumeration entries (number_integer, number_unsigned, and number_float), because the library distinguishes these three types for numbers: basic_json::number_unsigned_t is used for unsigned integers, basic_json::number_integer_t is used for signed integers, and basic_json::number_float_t is used for floating-point numbers or to approximate integers which do not fit in the limits of their respective type.

See also

basic_json::basic_json(const value_t value_type) – create a JSON value with the default value for a given type

Since

version 1.0.0

Enumerator

null null value

object object (unordered set of name/value pairs)

array array (ordered collection of values)

string string value

boolean boolean value

number_integer number value (signed integer)

number_unsigned number value (unsigned integer)

number_float number value (floating-point)

discarded discarded by the the parser callback function

5.2.3 Function Documentation

5.2.3.1 bool nlohmann::detail::operator< (const value_t *lhs*, const value_t *rhs*) [inline], [noexcept]

comparison operator for JSON types

Returns an ordering that is similar to Python:

- order: null < boolean < number < object < array < string
- furthermore, each type is not smaller than itself

Since

version 1.0.0

Chapter 6

Class Documentation

6.1 Action Class Reference

Inheritance diagram for Action:

6.2 addable Class Reference

Inheritance diagram for addable:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/su3.h

6.3 addable_left Class Reference

Inheritance diagram for addable_left:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/su3.h

6.4 nlohmann::adl_serializer< typename, typename > Struct Template Reference

default JSONSerializer template argument

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

Static Public Member Functions

- template<typename BasicJsonType , typename ValueType >
  static void from_json (BasicJsonType &&j, ValueType &val) noexcept(noexcept(::nlohmann::from_json(std↩
  ::forward< BasicJsonType >(j), val)))

  convert a JSON value to any value type
- template<typename BasicJsonType , typename ValueType >
  static void to_json (BasicJsonType &j, ValueType &&val) noexcept(noexcept(::nlohmann::to_json(j, std↩
  ::forward< ValueType >(val))))

  convert any value type to a JSON value

6.4.1 Detailed Description

template<typename, typename>
struct nlohmann::adl_serializer< typename, typename >

default JSONSerializer template argument

This serializer ignores the template arguments and uses ADL (`argument-dependent lookup`) for serialization.

6.4.2 Member Function Documentation

6.4.2.1 template<typename , typename > template<typename BasicJsonType , typename ValueType > static void nlohmann::adl_serializer< typename, typename >::from_json (BasicJsonType && *j*, ValueType & *val*) [inline], [static], [noexcept]

convert a JSON value to any value type

This function is usually called by the `get()` function of the basic_json class (either explicit or via conversion operators).

Parameters

in	*j*	JSON value to read from
in, out	*val*	value to write to

6.4.2.2 template<typename , typename > template<typename BasicJsonType , typename ValueType > static void nlohmann::adl_serializer< typename, typename >::to_json (BasicJsonType & *j*, ValueType && *val*) [inline], [static], [noexcept]

convert any value type to a JSON value

This function is usually called by the constructors of the basic_json class.

Parameters

in, out	*j*	JSON value to write to
in	*val*	value to read from

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.5 andable Class Reference

Inheritance diagram for andable:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.6 andable_left Class Reference

Inheritance diagram for andable_left:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.7 App Class Reference

Inheritance diagram for App:

Collaboration diagram for App:

Public Member Functions

- void **setAction** (Action *action)
- virtual void **createLattice** (std::array< int, 4 > latticeSize)
- void **addObservable** (Observable *observable)
- virtual void **execute** ()=0

Protected Attributes

- Action * **m_act** = nullptr
- GluonField * **m_lat** = nullptr
- std::array< int, 4 > **m_size**
- std::vector< Observable * > **m_obs**

The documentation for this class was generated from the following files:

- /home/giovanni/Desktop/LatticeYangMills/include/Apps/app.h
- /home/giovanni/Desktop/LatticeYangMills/src/Apps/app.cpp

6.8 B1 Class Reference

Inheritance diagram for B1:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.9 nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer > Class Template Reference

a class to store JSON values

`#include <json.hpp>`

Public Types

- using **value_t** = detail::value_t
- using **json_pointer** = ::nlohmann::json_pointer
- template<typename T , typename SFINAE >
  using **json_serializer** = JSONSerializer< T, SFINAE >
- using **initializer_list_t** = std::initializer_list< detail::json_ref< basic_json >>
- using **parse_event_t** = typename parser::parse_event_t
- using parser_callback_t = typename parser::parser_callback_t

  per-element parser callback type

Public Member Functions

- const char * type_name () const noexcept

  return the type as string

Static Public Member Functions

- static allocator_type get_allocator ()

  returns the allocator associated with the container
- static basic_json meta ()

  returns version information on the library

Friends

- template<detail::value_t >
  struct **detail::external_constructor**
- template<typename BasicJsonType >
  class **::nlohmann::detail::iter_impl**
- template<typename BasicJsonType , typename CharType >
  class **::nlohmann::detail::binary_writer**
- template<typename BasicJsonType >
  class **::nlohmann::detail::binary_reader**

exceptions

Classes to implement user-defined exceptions.

- using exception = detail::exception

  general exception of the basic_json class
- using parse_error = detail::parse_error

  exception indicating a parse error
- using invalid_iterator = detail::invalid_iterator

  exception indicating errors with iterators
- using type_error = detail::type_error

  exception indicating executing a member function with a wrong type
- using out_of_range = detail::out_of_range

  exception indicating access out of the defined range
- using other_error = detail::other_error

  exception indicating other library errors

container types

The canonic container types to use basic_json like any other STL container.

- using value_type = basic_json

  the type of elements in a basic_json container
- using reference = value_type &

  the type of an element reference
- using const_reference = const value_type &

  the type of an element const reference
- using difference_type = std::ptrdiff_t

  a type to represent differences between iterators
- using size_type = std::size_t

  a type to represent container sizes
- using allocator_type = AllocatorType< basic_json >

  the allocator type
- using pointer = typename std::allocator_traits< allocator_type >::pointer

  the type of an element pointer
- using const_pointer = typename std::allocator_traits< allocator_type >::const_pointer

  the type of an element const pointer
- using iterator = iter_impl< basic_json >

  an iterator for a basic_json container
- using const_iterator = iter_impl< const basic_json >

  a const iterator for a basic_json container
- using reverse_iterator = json_reverse_iterator< typename basic_json::iterator >

  a reverse iterator for a basic_json container
- using const_reverse_iterator = json_reverse_iterator< typename basic_json::const_iterator >

  a const reverse iterator for a basic_json container

JSON value data types

The data types to store a JSON value. These types are derived from the template arguments passed to class basic_json.

- using object_comparator_t = std::less< StringType >

  a type for an object
- using **object_t** = ObjectType< StringType, basic_json, object_comparator_t, AllocatorType< std::pair< const StringType, basic_json >>>
- using array_t = ArrayType< basic_json, AllocatorType< basic_json >>

  a type for an array
- using string_t = StringType

  a type for a string
- using boolean_t = BooleanType

  a type for a boolean
- using number_integer_t = NumberIntegerType

  a type for a number (integer)
- using number_unsigned_t = NumberUnsignedType

  a type for a number (unsigned)
- using number_float_t = NumberFloatType

  a type for a number (floating-point)

constructors and destructors

Constructors of class basic_json, copy/move constructor, copy assignment, static functions creating objects, and the destructor.

- static basic_json array (initializer_list_t init={})

  explicitly create an array from an initializer list
- static basic_json object (initializer_list_t init={})

  explicitly create an object from an initializer list
- basic_json (const value_t v)

  create an empty value with a given type
- basic_json (std::nullptr_t=nullptr) noexcept

  create a null object
- template<typename CompatibleType , typename U = detail::uncvref_t<CompatibleType>, detail::enable_if_t< not std::is_base_of< std::istream, U >::value andnot std::is_same< U, basic_json_t >::value andnot detail::is_basic_json_nested_type< basic_json_t, U >::value anddetail::has_to_json< basic_json, U >::value, int > = 0>
  basic_json (CompatibleType &&val) noexcept(noexcept(JSONSerializer< U >::to_json(std::declval< basic↩_json_t & >(), std::forward< CompatibleType >(val))))

  create a JSON value
- basic_json (initializer_list_t init, bool type_deduction=true, value_t manual_type=value_t::array)

  create a container (array or object) from an initializer list
- basic_json (size_type cnt, const basic_json &val)

  construct an array with count copies of given value
- template<class InputIT , typename std::enable_if< std::is_same< InputIT, typename basic_json_t::iterator >::value orstd::is_same< InputIT, typename basic_json_t::const_iterator >::value, int >::type = 0>
  basic_json (InputIT first, InputIT last)

  construct a JSON container given an iterator range
- basic_json (const basic_json &other)

copy constructor

- basic_json (basic_json &&other) noexcept

  move constructor

- reference & operator= (basic_json other) noexcept(std::is_nothrow_move_constructible< value_t >::value andstd::is_nothrow_move_assignable< value_t >::value andstd::is_nothrow_move_constructible< json_↩ value >::value andstd::is_nothrow_move_assignable< json_value >::value)

  copy assignment

- ~basic_json ()

  destructor

object inspection

Functions to inspect the type of a JSON value.

- string_t dump (const int indent=-1, const char indent_char= ' ', const bool ensure_ascii=false) const

  serialization

- constexpr value_t type () const noexcept

  return the type of the JSON value (explicit)

- constexpr bool is_primitive () const noexcept

  return whether type is primitive

- constexpr bool is_structured () const noexcept

  return whether type is structured

- constexpr bool is_null () const noexcept

  return whether value is null

- constexpr bool is_boolean () const noexcept

  return whether value is a boolean

- constexpr bool is_number () const noexcept

  return whether value is a number

- constexpr bool is_number_integer () const noexcept

  return whether value is an integer number

- constexpr bool is_number_unsigned () const noexcept

  return whether value is an unsigned integer number

- constexpr bool is_number_float () const noexcept

  return whether value is a floating-point number

- constexpr bool is_object () const noexcept

  return whether value is an object

- constexpr bool is_array () const noexcept

  return whether value is an array

- constexpr bool is_string () const noexcept

  return whether value is a string

- constexpr bool is_discarded () const noexcept

  return whether value is discarded

- constexpr operator value_t () const noexcept

  return the type of the JSON value (implicit)

value access

Direct access to the stored value of a JSON value.

- template<typename BasicJsonType , detail::enable_if_t< std::is_same< typename std::remove_const< BasicJsonType >::type, basic_json_t >::value, int > = 0>
  basic_json get () const

  get special-case overload
- template<typename ValueTypeCV , typename ValueType = detail::uncvref_t<ValueTypeCV>, detail::enable_if_t< not std::is_same< basic_json_t, ValueType >::value anddetail::has_from_json< basic_json_t, ValueType >::value andnot detail::has_non_default_from↩_json< basic_json_t, ValueType >::value, int > = 0>
  ValueType get () const noexcept(noexcept(JSONSerializer< ValueType >::from_json(std::declval< const basic_json_t & >(), std::declval< ValueType & >())))

  get a value (explicit)
- template<typename ValueTypeCV , typename ValueType = detail::uncvref_t<ValueTypeCV>, detail::enable_if_t< not std::is_same< basic_json_t, ValueType >::value anddetail::has_non_default_from_json< basic_json_t, ValueType >::value, int > = 0>
  ValueType get () const noexcept(noexcept(JSONSerializer< ValueTypeCV >::from_json(std::declval< const basic_json_t & >())))

  get a value (explicit); special case
- template<typename PointerType , typename std::enable_if< std::is_pointer< PointerType >::value, int >::type = 0>
  PointerType get () noexcept

  get a pointer value (explicit)
- template<typename PointerType , typename std::enable_if< std::is_pointer< PointerType >::value, int >::type = 0>
  constexpr const PointerType get () const noexcept

  get a pointer value (explicit)
- template<typename PointerType , typename std::enable_if< std::is_pointer< PointerType >::value, int >::type = 0>
  PointerType get_ptr () noexcept

  get a pointer value (implicit)
- template<typename PointerType , typename std::enable_if< std::is_pointer< PointerType >::value andstd::is_const< typename std↩::remove_pointer< PointerType >::type >::value, int >::type = 0>
  constexpr const PointerType get_ptr () const noexcept

  get a pointer value (implicit)
- template<typename ReferenceType , typename std::enable_if< std::is_reference< ReferenceType >::value, int >::type = 0>
  ReferenceType get_ref ()

  get a reference value (implicit)
- template<typename ReferenceType , typename std::enable_if< std::is_reference< ReferenceType >::value andstd::is_const< typename std::remove_reference< ReferenceType >::type >::value, int >::type = 0>
  ReferenceType get_ref () const

  get a reference value (implicit)
- template<typename ValueType , typename std::enable_if< not std::is_pointer< ValueType >::value andnot std::is_same< ValueType, detail::json_ref< basic_json >>::value andnot std::is_same< ValueType, typename string_t::value_type >::valueand not std::is_↩same< ValueType, std::initializer_list< typename string_t::value_type >>::value, int >::type = 0>
  operator ValueType () const

  get a value (implicit)

element access

Access to the JSON value.

- reference at (size_type idx)

  access specified array element with bounds checking

- const_reference at (size_type idx) const

  access specified array element with bounds checking
- reference at (const typename object_t::key_type &key)

  access specified object element with bounds checking
- const_reference at (const typename object_t::key_type &key) const

  access specified object element with bounds checking
- reference operator[ ] (size_type idx)

  access specified array element
- const_reference operator[ ] (size_type idx) const

  access specified array element
- reference operator[ ] (const typename object_t::key_type &key)

  access specified object element
- const_reference operator[ ] (const typename object_t::key_type &key) const

  read-only access specified object element
- template<typename T >
  reference operator[ ] (T *key)

  access specified object element
- template<typename T >
  const_reference operator[ ] (T *key) const

  read-only access specified object element
- template<class ValueType , typename std::enable_if< std::is_convertible< basic_json_t, ValueType >::value, int >::type = 0>
  ValueType value (const typename object_t::key_type &key, const ValueType &default_value) const

  access specified object element with default value
- string_t value (const typename object_t::key_type &key, const char *default_value) const

  *overload for a default value of type const char**
- template<class ValueType , typename std::enable_if< std::is_convertible< basic_json_t, ValueType >::value, int >::type = 0>
  ValueType value (const json_pointer &ptr, const ValueType &default_value) const

  access specified object element via JSON Pointer with default value
- string_t value (const json_pointer &ptr, const char *default_value) const

  *overload for a default value of type const char**
- reference front ()

  access the first element
- const_reference front () const

  access the first element
- reference back ()

  access the last element
- const_reference back () const

  access the last element
- template<class IteratorType , typename std::enable_if< std::is_same< IteratorType, typename basic_json_t::iterator >::value orstd↩
  ::is_same< IteratorType, typename basic_json_t::const_iterator >::value, int >::type = 0>
  IteratorType erase (IteratorType pos)

  remove element given an iterator
- template<class IteratorType , typename std::enable_if< std::is_same< IteratorType, typename basic_json_t::iterator >::value orstd↩
  ::is_same< IteratorType, typename basic_json_t::const_iterator >::value, int >::type = 0>
  IteratorType erase (IteratorType first, IteratorType last)

  remove elements given an iterator range
- size_type erase (const typename object_t::key_type &key)

  remove element from a JSON object given a key
- void erase (const size_type idx)

  remove element from a JSON array given an index

lookup

- template<typename KeyT >
  iterator find (KeyT &&key)

  find an element in a JSON object
- template<typename KeyT >
  const_iterator find (KeyT &&key) const

  find an element in a JSON object
- template<typename KeyT >
  size_type count (KeyT &&key) const

  returns the number of occurrences of a key in a JSON object

iterators

- static iteration_proxy< iterator > iterator_wrapper (reference cont)

  wrapper to access iterator member functions in range-based for
- static iteration_proxy< const_iterator > iterator_wrapper (const_reference cont)

  wrapper to access iterator member functions in range-based for
- iterator begin () noexcept

  returns an iterator to the first element
- const_iterator begin () const noexcept

  returns a const iterator to the first element
- const_iterator cbegin () const noexcept

  returns a const iterator to the first element
- iterator end () noexcept

  returns an iterator to one past the last element
- const_iterator end () const noexcept

  returns a const iterator to one past the last element
- const_iterator cend () const noexcept

  returns a const iterator to one past the last element
- reverse_iterator rbegin () noexcept

  returns an iterator to the reverse-beginning
- const_reverse_iterator rbegin () const noexcept

  returns a const reverse iterator to the last element
- reverse_iterator rend () noexcept

  returns an iterator to the reverse-end
- const_reverse_iterator rend () const noexcept

  returns a const reverse iterator to one before the first
- const_reverse_iterator crbegin () const noexcept

  returns a const reverse iterator to the last element
- const_reverse_iterator crend () const noexcept

  returns a const reverse iterator to one before the first

capacity

- bool empty () const noexcept

  checks whether the container is empty.
- size_type size () const noexcept

  returns the number of elements
- size_type max_size () const noexcept

  returns the maximum possible number of elements

modifiers

- void clear () noexcept

  clears the contents
- void push_back (basic_json &&val)

  add an object to an array
- reference operator+= (basic_json &&val)

  add an object to an array
- void push_back (const basic_json &val)

  add an object to an array
- reference operator+= (const basic_json &val)

  add an object to an array
- void push_back (const typename object_t::value_type &val)

  add an object to an object
- reference operator+= (const typename object_t::value_type &val)

  add an object to an object
- void push_back (initializer_list_t init)

  add an object to an object
- reference operator+= (initializer_list_t init)

  add an object to an object
- template<class... Args>
  void emplace_back (Args &&...args)

  add an object to an array
- template<class... Args>
  std::pair< iterator, bool > emplace (Args &&...args)

  add an object to an object if key does not exist
- iterator insert (const_iterator pos, const basic_json &val)

  inserts element
- iterator insert (const_iterator pos, basic_json &&val)

  inserts element
- iterator insert (const_iterator pos, size_type cnt, const basic_json &val)

  inserts elements
- iterator insert (const_iterator pos, const_iterator first, const_iterator last)

  inserts elements
- iterator insert (const_iterator pos, initializer_list_t ilist)

  inserts elements
- void insert (const_iterator first, const_iterator last)

  inserts elements
- void update (const_reference j)

  updates a JSON object from another object, overwriting existing keys
- void update (const_iterator first, const_iterator last)

  updates a JSON object from another object, overwriting existing keys
- void swap (reference other) noexcept(std::is_nothrow_move_constructible< value_t >::value andstd::is_↩ nothrow_move_assignable< value_t >::value andstd::is_nothrow_move_constructible< json_value >::value andstd::is_nothrow_move_assignable< json_value >::value)

  exchanges the values
- void swap (array_t &other)

  exchanges the values
- void swap (object_t &other)

  exchanges the values
- void swap (string_t &other)

  exchanges the values

lexicographical comparison operators

- bool operator== (const_reference lhs, const_reference rhs) noexcept

  comparison: equal
- template<typename ScalarType , typename std::enable_if< std::is_scalar< ScalarType >::value, int >::type = 0>
  bool operator== (const_reference lhs, const ScalarType rhs) noexcept

  comparison: equal
- template<typename ScalarType , typename std::enable_if< std::is_scalar< ScalarType >::value, int >::type = 0>
  bool operator== (const ScalarType lhs, const_reference rhs) noexcept

  comparison: equal
- bool operator!= (const_reference lhs, const_reference rhs) noexcept

  comparison: not equal
- template<typename ScalarType , typename std::enable_if< std::is_scalar< ScalarType >::value, int >::type = 0>
  bool operator!= (const_reference lhs, const ScalarType rhs) noexcept

  comparison: not equal
- template<typename ScalarType , typename std::enable_if< std::is_scalar< ScalarType >::value, int >::type = 0>
  bool operator!= (const ScalarType lhs, const_reference rhs) noexcept

  comparison: not equal
- bool operator< (const_reference lhs, const_reference rhs) noexcept

  comparison: less than
- template<typename ScalarType , typename std::enable_if< std::is_scalar< ScalarType >::value, int >::type = 0>
  bool operator< (const_reference lhs, const ScalarType rhs) noexcept

  comparison: less than
- template<typename ScalarType , typename std::enable_if< std::is_scalar< ScalarType >::value, int >::type = 0>
  bool operator< (const ScalarType lhs, const_reference rhs) noexcept

  comparison: less than
- bool operator<= (const_reference lhs, const_reference rhs) noexcept

  comparison: less than or equal
- template<typename ScalarType , typename std::enable_if< std::is_scalar< ScalarType >::value, int >::type = 0>
  bool operator<= (const_reference lhs, const ScalarType rhs) noexcept

  comparison: less than or equal
- template<typename ScalarType , typename std::enable_if< std::is_scalar< ScalarType >::value, int >::type = 0>
  bool operator<= (const ScalarType lhs, const_reference rhs) noexcept

  comparison: less than or equal
- bool operator> (const_reference lhs, const_reference rhs) noexcept

  comparison: greater than
- template<typename ScalarType , typename std::enable_if< std::is_scalar< ScalarType >::value, int >::type = 0>
  bool operator> (const_reference lhs, const ScalarType rhs) noexcept

  comparison: greater than
- template<typename ScalarType , typename std::enable_if< std::is_scalar< ScalarType >::value, int >::type = 0>
  bool operator> (const ScalarType lhs, const_reference rhs) noexcept

  comparison: greater than
- bool operator>= (const_reference lhs, const_reference rhs) noexcept

  comparison: greater than or equal
- template<typename ScalarType , typename std::enable_if< std::is_scalar< ScalarType >::value, int >::type = 0>
  bool operator>= (const_reference lhs, const ScalarType rhs) noexcept

  comparison: greater than or equal
- template<typename ScalarType , typename std::enable_if< std::is_scalar< ScalarType >::value, int >::type = 0>
  bool operator>= (const ScalarType lhs, const_reference rhs) noexcept

  comparison: greater than or equal

serialization

- std::ostream & operator<< (std::ostream &o, const basic_json &j)

  serialize to stream
- JSON_DEPRECATED friend std::ostream & operator>> (const basic_json &j, std::ostream &o)

  serialize to stream

deserialization

- JSON_DEPRECATED friend std::istream & operator<< (basic_json &j, std::istream &i)

  deserialize from stream
- std::istream & operator>> (std::istream &i, basic_json &j)

  deserialize from stream
- static basic_json parse (detail::input_adapter i, const parser_callback_t cb=nullptr, const bool allow_↩ exceptions=true)

  deserialize from a compatible input
- static basic_json parse (detail::input_adapter &i, const parser_callback_t cb=nullptr, const bool allow_↩ exceptions=true)

  create an empty value with a given type parse(detail::input_adapter, const parser_callback_t)
- static bool **accept** (detail::input_adapter i)
- static bool **accept** (detail::input_adapter &i)
- template<class IteratorType , typename std::enable_if< std::is_base_of< std::random_access_iterator_tag, typename std::iterator_↩ traits< IteratorType >::iterator_category >::value, int >::type = 0>
  static basic_json parse (IteratorType first, IteratorType last, const parser_callback_t cb=nullptr, const bool allow_exceptions=true)

  deserialize from an iterator range with contiguous storage
- template<class IteratorType , typename std::enable_if< std::is_base_of< std::random_access_iterator_tag, typename std::iterator_↩ traits< IteratorType >::iterator_category >::value, int >::type = 0>
  static bool **accept** (IteratorType first, IteratorType last)

binary serialization/deserialization support

- static std::vector< uint8_t > to_cbor (const basic_json &j)

  create a CBOR serialization of a given JSON value
- static void **to_cbor** (const basic_json &j, detail::output_adapter< uint8_t > o)
- static void **to_cbor** (const basic_json &j, detail::output_adapter< char > o)
- static std::vector< uint8_t > to_msgpack (const basic_json &j)

  create a MessagePack serialization of a given JSON value
- static void **to_msgpack** (const basic_json &j, detail::output_adapter< uint8_t > o)
- static void **to_msgpack** (const basic_json &j, detail::output_adapter< char > o)
- static basic_json from_cbor (detail::input_adapter i, const bool strict=true)

  create a JSON value from an input in CBOR format
- template<typename A1 , typename A2 , detail::enable_if_t< std::is_constructible< detail::input_adapter, A1, A2 >::value, int > = 0>
  static basic_json from_cbor (A1 &&a1, A2 &&a2, const bool strict=true)

  create a JSON value from an input in CBOR format
- static basic_json from_msgpack (detail::input_adapter i, const bool strict=true)

  create a JSON value from an input in MessagePack format
- template<typename A1 , typename A2 , detail::enable_if_t< std::is_constructible< detail::input_adapter, A1, A2 >::value, int > = 0>
  static basic_json from_msgpack (A1 &&a1, A2 &&a2, const bool strict=true)

  create a JSON value from an input in MessagePack format

JSON Pointer functions

- reference operator[ ] (const json_pointer &ptr)

  access specified element via JSON Pointer
- const_reference operator[ ] (const json_pointer &ptr) const

  access specified element via JSON Pointer
- reference at (const json_pointer &ptr)

  access specified element via JSON Pointer
- const_reference at (const json_pointer &ptr) const

  access specified element via JSON Pointer
- basic_json flatten () const

  return flattened JSON value
- basic_json unflatten () const

  unflatten a previously flattened JSON value

JSON Patch functions

- static basic_json diff (const basic_json &source, const basic_json &target, const std::string &path="")

  creates a diff as a JSON patch
- basic_json patch (const basic_json &json_patch) const

  applies a JSON patch

6.9.1 Detailed Description

template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberInteger↩ Type = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> class nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >

a class to store JSON values

Template Parameters

ObjectType	type for JSON objects (`std::map` by default; will be used in object_t)
ArrayType	type for JSON arrays (`std::vector` by default; will be used in array_t)
StringType	type for JSON strings and object keys (`std::string` by default; will be used in string_t)
BooleanType	type for JSON booleans (`bool` by default; will be used in boolean_t)
NumberIntegerType	type for JSON integer numbers (`int64_t` by default; will be used in number_integer_t)
NumberUnsignedType	type for JSON unsigned integer numbers (`uint64_t` by default; will be used in number_unsigned_t)
NumberFloatType	type for JSON floating-point numbers (`double` by default; will be used in number_float_t)
AllocatorType	type of the allocator to use (`std::allocator` by default)
JSONSerializer	the serializer to resolve internal calls to `to_json()` and `from_json()` (adl_serializer by default)

The class satisfies the following concept requirements:

- Basic
  - `DefaultConstructible`: JSON values can be default constructed. The result will be a JSON null value.
  - `MoveConstructible`: A JSON value can be constructed from an rvalue argument.
  - `CopyConstructible`: A JSON value can be copy-constructed from an lvalue expression.
  - `MoveAssignable`: A JSON value van be assigned from an rvalue argument.
  - `CopyAssignable`: A JSON value can be copy-assigned from an lvalue expression.
  - `Destructible`: JSON values can be destructed.
- Layout
  - `StandardLayoutType`: JSON values have `standard layout`: All non-static data members are private and standard layout types, the class has no virtual functions or (virtual) base classes.
- Library-wide
  - `EqualityComparable`: JSON values can be compared with `==`, see operator==(const_reference,const_reference).
  - `LessThanComparable`: JSON values can be compared with `<`, see operator<(const_reference,const_reference).
  - `Swappable`: Any JSON lvalue or rvalue of can be swapped with any lvalue or rvalue of other compatible types, using unqualified function call swap().
  - `NullablePointer`: JSON values can be compared against `std::nullptr_t` objects which are used to model the `null` value.
- Container
  - `Container`: JSON values can be used like STL containers and provide iterator access.
  - `ReversibleContainer`; JSON values can be used like STL containers and provide reverse iterator access.

Invariant

The member variables *m_value* and *m_type* have the following relationship:

- If `m_type == value_t::object`, then `m_value.object != nullptr`.
- If `m_type == value_t::array`, then `m_value.array != nullptr`.
- If `m_type == value_t::string`, then `m_value.string != nullptr`. The invariants are checked by member function assert_invariant().

See also

`RFC 7159: The JavaScript Object Notation (JSON) Data Interchange Format`

Since

version 1.0.0

6.9.2 Member Typedef Documentation

6.9.2.1 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> using nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::array_t = ArrayType<basic_json, AllocatorType<basic_json>>

a type for an array

RFC 7159 describes JSON arrays as follows:

> An array is an ordered sequence of zero or more values.

To store objects in C++, a type is defined by the template parameters explained below.

Template Parameters

ArrayType	container type to store arrays (e.g., `std::vector` or `std::list`)
AllocatorType	allocator to use for arrays (e.g., `std::allocator`)

Default type

With the default values for *ArrayType* (`std::vector`) and *AllocatorType* (`std::allocator`), the default value for *array_t* is:

```
std::vector<
  basic_json, // value_type
  std::allocator<basic_json> // allocator_type
>
```

Limits

RFC 7159 specifies:

> An implementation may set limits on the maximum depth of nesting.

In this class, the array's limit of nesting is not explicitly constrained. However, a maximum depth of nesting may be introduced by the compiler or runtime environment. A theoretical limit can be queried by calling the max_size function of a JSON array.

Storage

Arrays are stored as pointers in a basic_json type. That is, for any access to array values, a pointer of type `array_t*` must be dereferenced.

See also

object_t – type for an object value

Since

version 1.0.0

6.9.2.2 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> using nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::boolean_t = BooleanType

a type for a boolean

RFC 7159 implicitly describes a boolean as a type which differentiates the two literals `true` and `false`.

To store objects in C++, a type is defined by the template parameter *BooleanType* which chooses the type to use.

Default type

With the default values for *BooleanType* (`bool`), the default value for *boolean_t* is:

```
bool
```

Storage

Boolean values are stored directly inside a basic_json type.

Since

version 1.0.0

6.9.2.3 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> using nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::exception = detail::exception

general exception of the basic_json class

This class is an extension of `std::exception` objects with a member *id* for exception ids. It is used as the base class for all exceptions thrown by the basic_json class. This class can hence be used as "wildcard" to catch exceptions.

Subclasses:

- parse_error for exceptions indicating a parse error
- invalid_iterator for exceptions indicating errors with iterators
- type_error for exceptions indicating executing a member function with a wrong type
- out_of_range for exceptions indicating access out of the defined range
- other_error for exceptions indicating other library errors

{The following code shows how arbitrary library exceptions can be caught.,exception}

Since

version 3.0.0

6.9.2.4 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> using nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::invalid_iterator = detail::invalid_iterator

exception indicating errors with iterators

This exception is thrown if iterators passed to a library function do not match the expected semantics.

Exceptions have ids 2xx.

name / id	example message	description
json.exception.invalid_iterator.201	iterators are not compatible	The iterators passed to constructor basic_json(InputIT first, InputIT last) are not compatible, meaning they do not belong to the same container. Therefore, the range (*first*, *last*) is invalid.
json.exception.invalid_iterator.202	iterator does not fit current value	In an erase or insert function, the passed iterator *pos* does not belong to the JSON value for which the function was called. It hence does not define a valid position for the deletion/insertion.
json.exception.invalid_iterator.203	iterators do not fit current value	Either iterator passed to function erase(IteratorType first, IteratorType last) does not belong to the JSON value from which values shall be erased. It hence does not define a valid range to delete values from.
json.exception.invalid_iterator.204	iterators out of range	When an iterator range for a primitive type (number, boolean, or string) is passed to a constructor or an erase function, this range has to be exactly (begin(), end()), because this is the only way the single stored value is expressed. All other ranges are invalid.
json.exception.invalid_iterator.205	iterator out of range	When an iterator for a primitive type (number, boolean, or string) is passed to an erase function, the iterator has to be the begin() iterator, because it is the only way to address the stored value. All other iterators are invalid.
json.exception.invalid_iterator.206	cannot construct with iterators from null	The iterators passed to constructor basic_json(InputIT first, InputIT last) belong to a JSON null value and hence to not define a valid range.
json.exception.invalid_iterator.207	cannot use key() for non-object iterators	The key() member function can only be used on iterators belonging to a JSON object, because other types do not have a concept of a key.

name / id	example message	description
json.exception.invalid_iterator.208	cannot use operator[] for object iterators	The operator[] to specify a concrete offset cannot be used on iterators belonging to a JSON object, because JSON objects are unordered.
json.exception.invalid_iterator.209	cannot use offsets with object iterators	The offset operators (+, -, +=, -=) cannot be used on iterators belonging to a JSON object, because JSON objects are unordered.
json.exception.invalid_iterator.210	iterators do not fit	The iterator range passed to the insert function are not compatible, meaning they do not belong to the same container. Therefore, the range (*first*, *last*) is invalid.
json.exception.invalid_iterator.211	passed iterators may not belong to container	The iterator range passed to the insert function must not be a subrange of the container to insert to.
json.exception.invalid_iterator.212	cannot compare iterators of different containers	When two iterators are compared, they must belong to the same container.
json.exception.invalid_iterator.213	cannot compare order of object iterators	The order of object iterators cannot be compared, because JSON objects are unordered.
json.exception.invalid_iterator.214	cannot get value	Cannot get value for iterator: Either the iterator belongs to a null value or it is an iterator to a primitive type (number, boolean, or string), but the iterator is different to begin().

{The following code shows how an `invalid_iterator` exception can be caught.,invalid_iterator}

See also

exception for the base class of the library exceptions
parse_error for exceptions indicating a parse error
type_error for exceptions indicating executing a member function with a wrong type
out_of_range for exceptions indicating access out of the defined range
other_error for exceptions indicating other library errors

Since

version 3.0.0

6.9.2.5 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> using nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::number_float_t = NumberFloatType

a type for a number (floating-point)

RFC 7159 describes numbers as follows:

> The representation of numbers is similar to that used in most programming languages. A number is represented in base 10 using decimal digits. It contains an integer component that may be prefixed with an optional minus sign, which may be followed by a fraction part and/or an exponent part. Leading zeros are not allowed. (...) Numeric values that cannot be represented in the grammar below (such as Infinity and NaN) are not permitted.

This description includes both integer and floating-point numbers. However, C++ allows more precise storage if it is known whether the number is a signed integer, an unsigned integer or a floating-point number. Therefore, three different types, number_integer_t, number_unsigned_t and number_float_t are used.

To store floating-point numbers in C++, a type is defined by the template parameter *NumberFloatType* which chooses the type to use.

Default type

With the default values for *NumberFloatType* (`double`), the default value for *number_float_t* is:

```
double
```

Default behavior

- The restrictions about leading zeros is not enforced in C++. Instead, leading zeros in floating-point literals will be ignored. Internally, the value will be stored as decimal number. For instance, the C++ floating-point literal `01.2` will be serialized to `1.2`. During deserialization, leading zeros yield an error.
- Not-a-number (NaN) values will be serialized to `null`.

Limits

RFC 7159 states:

> This specification allows implementations to set limits on the range and precision of numbers accepted. Since software that implements IEEE 754-2008 binary64 (double precision) numbers is generally available and widely used, good interoperability can be achieved by implementations that expect no more precision or range than these provide, in the sense that implementations will approximate JSON numbers within the expected precision.

This implementation does exactly follow this approach, as it uses double precision floating-point numbers. Note values smaller than `-1.7976931348623e+308` and values greater than `1.7976931348623e+308` will be stored as NaN internally and be serialized to `null`.

Storage

Floating-point number values are stored directly inside a basic_json type.

See also

number_integer_t – type for number values (integer)
number_unsigned_t – type for number values (unsigned integer)

Since

version 1.0.0

6.9.2.6 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> using nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::number_integer_t = NumberIntegerType

a type for a number (integer)

RFC 7159 describes numbers as follows:

> The representation of numbers is similar to that used in most programming languages. A number is represented in base 10 using decimal digits. It contains an integer component that may be prefixed with an optional minus sign, which may be followed by a fraction part and/or an exponent part. Leading zeros are not allowed. (...) Numeric values that cannot be represented in the grammar below (such as Infinity and NaN) are not permitted.

This description includes both integer and floating-point numbers. However, C++ allows more precise storage if it is known whether the number is a signed integer, an unsigned integer or a floating-point number. Therefore, three different types, number_integer_t, number_unsigned_t and number_float_t are used.

To store integer numbers in C++, a type is defined by the template parameter *NumberIntegerType* which chooses the type to use.

Default type

With the default values for *NumberIntegerType* (`int64_t`), the default value for *number_integer_t* is:

```
int64_t
```

Default behavior

- The restrictions about leading zeros is not enforced in C++. Instead, leading zeros in integer literals lead to an interpretation as octal number. Internally, the value will be stored as decimal number. For instance, the C++ integer literal `010` will be serialized to `8`. During deserialization, leading zeros yield an error.
- Not-a-number (NaN) values will be serialized to `null`.

Limits

RFC 7159 specifies:

> An implementation may set limits on the range and precision of numbers.

When the default type is used, the maximal integer number that can be stored is `9223372036854775807` (INT64_MAX) and the minimal integer number that can be stored is `-9223372036854775808` (INT64_MIN). Integer numbers that are out of range will yield over/underflow when used in a constructor. During deserialization, too large or small integer numbers will be automatically be stored as number_unsigned_t or number_float_t.

RFC 7159 further states:

> Note that when such software is used, numbers that are integers and are in the range $[-2^{53}+1, 2^{53}-1]$ are interoperable in the sense that implementations will agree exactly on their numeric values.

As this range is a subrange of the exactly supported range [INT64_MIN, INT64_MAX], this class's integer type is interoperable.

Storage

Integer number values are stored directly inside a basic_json type.

See also

number_float_t – type for number values (floating-point)
number_unsigned_t – type for number values (unsigned integer)

Since

version 1.0.0

6.9.2.7 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> using nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::number_unsigned_t = NumberUnsignedType

a type for a number (unsigned)

RFC 7159 describes numbers as follows:

> The representation of numbers is similar to that used in most programming languages. A number is represented in base 10 using decimal digits. It contains an integer component that may be prefixed with an optional minus sign, which may be followed by a fraction part and/or an exponent part. Leading zeros are not allowed. (...) Numeric values that cannot be represented in the grammar below (such as Infinity and NaN) are not permitted.

This description includes both integer and floating-point numbers. However, C++ allows more precise storage if it is known whether the number is a signed integer, an unsigned integer or a floating-point number. Therefore, three different types, number_integer_t, number_unsigned_t and number_float_t are used.

To store unsigned integer numbers in C++, a type is defined by the template parameter *NumberUnsignedType* which chooses the type to use.

Default type

With the default values for *NumberUnsignedType* (`uint64_t`), the default value for *number_unsigned_t* is:

```
uint64_t
```

Default behavior

- The restrictions about leading zeros is not enforced in C++. Instead, leading zeros in integer literals lead to an interpretation as octal number. Internally, the value will be stored as decimal number. For instance, the C++ integer literal `010` will be serialized to `8`. During deserialization, leading zeros yield an error.
- Not-a-number (NaN) values will be serialized to `null`.

Limits

RFC 7159 specifies:

> An implementation may set limits on the range and precision of numbers.

When the default type is used, the maximal integer number that can be stored is 18446744073709551615 (UINT64_MAX) and the minimal integer number that can be stored is 0. Integer numbers that are out of range will yield over/underflow when used in a constructor. During deserialization, too large or small integer numbers will be automatically be stored as number_integer_t or number_float_t.

RFC 7159 further states:

> Note that when such software is used, numbers that are integers and are in the range $[-2^{53}+1, 2^{53}-1]$ are interoperable in the sense that implementations will agree exactly on their numeric values.

As this range is a subrange (when considered in conjunction with the number_integer_t type) of the exactly supported range [0, UINT64_MAX], this class's integer type is interoperable.

Storage

Integer number values are stored directly inside a basic_json type.

See also

number_float_t – type for number values (floating-point)
number_integer_t – type for number values (integer)

Since

version 2.0.0

6.9.2.8 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> using nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::object_comparator_t = std::less<StringType>

a type for an object

RFC 7159 describes JSON objects as follows:

> An object is an unordered collection of zero or more name/value pairs, where a name is a string and a value is a string, number, boolean, null, object, or array.

To store objects in C++, a type is defined by the template parameters described below.

Template Parameters

ObjectType	the container to store objects (e.g., `std::map` or `std::unordered_map`)
StringType	the type of the keys or names (e.g., `std::string`). The comparison function `std::less<StringType>` is used to order elements inside the container.
AllocatorType	the allocator to use for objects (e.g., `std::allocator`)

Default type

With the default values for *ObjectType* (`std::map`), *StringType* (`std::string`), and *AllocatorType* (`std::allocator`), the default value for *object_t* is:

```
std::map<
  std::string, // key_type
  basic_json, // value_type
  std::less<std::string>, // key_compare
  std::allocator<std::pair<const std::string, basic_json>> // allocator_type
>
```

Behavior

The choice of *object_t* influences the behavior of the JSON class. With the default type, objects have the following behavior:

- When all names are unique, objects will be interoperable in the sense that all software implementations receiving that object will agree on the name-value mappings.
- When the names within an object are not unique, later stored name/value pairs overwrite previously stored name/value pairs, leaving the used names unique. For instance, `{"key": 1}` and `{"key": 2, "key": 1}` will be treated as equal and both stored as `{"key": 1}`.
- Internally, name/value pairs are stored in lexicographical order of the names. Objects will also be serialized (see dump) in this order. For instance, `{"b": 1, "a": 2}` and `{"a": 2, "b": 1}` will be stored and serialized as `{"a": 2, "b": 1}`.
- When comparing objects, the order of the name/value pairs is irrelevant. This makes objects interoperable in the sense that they will not be affected by these differences. For instance, `{"b": 1, "a": 2}` and `{"a": 2, "b": 1}` will be treated as equal.

Limits

RFC 7159 specifies:

> An implementation may set limits on the maximum depth of nesting.

In this class, the object's limit of nesting is not explicitly constrained. However, a maximum depth of nesting may be introduced by the compiler or runtime environment. A theoretical limit can be queried by calling the max_size function of a JSON object.

Storage

Objects are stored as pointers in a basic_json type. That is, for any access to object values, a pointer of type `object_t*` must be dereferenced.

See also

array_t – type for an array value

Since

version 1.0.0

Note

The order name/value pairs are added to the object is *not* preserved by the library. Therefore, iterating an object may return name/value pairs in a different order than they were originally stored. In fact, keys will be traversed in alphabetical order as `std::map` with `std::less` is used by default. Please note this behavior conforms to RFC 7159, because any order implements the specified "unordered" nature of JSON objects.

6.9.2.9 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> using nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::other_error = detail::other_error

exception indicating other library errors

This exception is thrown in case of errors that cannot be classified with the other exception types.

Exceptions have ids 5xx.

name / id	example message	description
json.exception.other_error.501	unsuccessful: {"op":"test","path"↩ :"/baz", "value":"bar"}	A JSON Patch operation 'test' failed. The unsuccessful operation is also printed.
json.exception.other_error.502	invalid object size for conversion	Some conversions to user-defined types impose constraints on the object size (e.g. std::pair)

See also

exception for the base class of the library exceptions
parse_error for exceptions indicating a parse error
invalid_iterator for exceptions indicating errors with iterators
type_error for exceptions indicating executing a member function with a wrong type
out_of_range for exceptions indicating access out of the defined range

{The following code shows how an `other_error` exception can be caught.,other_error}

Since

version 3.0.0

6.9.2.10 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> using nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::out_of_range = detail::out_of_range

exception indicating access out of the defined range

This exception is thrown in case a library function is called on an input parameter that exceeds the expected range, for instance in case of array indices or nonexisting object keys.

Exceptions have ids 4xx.

name / id	example message	description
json.exception.out_of_range.401	array index 3 is out of range	The provided array index *i* is larger than *size-1*.
json.exception.out_of_range.402	array index '-' (3) is out of range	The special array index - in a JSON Pointer never describes a valid element of the array, but the index past the end. That is, it can only be used to add elements at this position, but not to read it.
json.exception.out_of_range.403	key 'foo' not found	The provided key was not found in the JSON object.
json.exception.out_of_range.404	unresolved reference token 'foo'	A reference token in a JSON Pointer could not be resolved.
json.exception.out_of_range.405	JSON pointer has no parent	The JSON Patch operations 'remove' and 'add' can not be applied to the root element of the JSON value.
json.exception.out_of_range.406	number overflow parsing '10E1000'	A parsed number could not be stored as without changing it to NaN or INF.

{The following code shows how an `out_of_range` exception can be caught.,out_of_range}

See also

exception for the base class of the library exceptions
parse_error for exceptions indicating a parse error
invalid_iterator for exceptions indicating errors with iterators
type_error for exceptions indicating executing a member function with a wrong type
other_error for exceptions indicating other library errors

Since

version 3.0.0

6.9.2.11 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> using nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::parse_error = detail::parse_error

exception indicating a parse error

This excpetion is thrown by the library when a parse error occurs. Parse errors can occur during the deserialization of JSON text, CBOR, MessagePack, as well as when using JSON Patch.

Member *byte* holds the byte index of the last read character in the input file.

Exceptions have ids 1xx.

name / id	example message	description
json.exception.parse_error.101	parse error at 2: unexpected end of input; expected string literal	This error indicates a syntax error while deserializing a JSON text. The error message describes that an unexpected token (character) was encountered, and the member *byte* indicates the error position.
json.exception.parse_error.102	parse error at 14: missing or wrong low surrogate	JSON uses the `\uxxxx` format to describe Unicode characters. Code points above above 0xFFFF are split into two `\uxxxx` entries ("surrogate pairs"). This error indicates that the surrogate pair is incomplete or contains an invalid code point.
json.exception.parse_error.103	parse error: code points above 0x10FFFF are invalid	Unicode supports code points up to 0x10FFFF. Code points above 0x10FFFF are invalid.
json.exception.parse_error.104	parse error: JSON patch must be an array of objects	RFC 6902 requires a JSON Patch document to be a JSON document that represents an array of objects.
json.exception.parse_error.105	parse error: operation must have string member 'op'	An operation of a JSON Patch document must contain exactly one "op" member, whose value indicates the operation to perform. Its value must be one of "add", "remove", "replace", "move", "copy", or "test"; other values are errors.
json.exception.parse_error.106	parse error: array index '01' must not begin with '0'	An array index in a JSON Pointer (RFC 6901) may be `0` or any number wihtout a leading `0`.
json.exception.parse_error.107	parse error: JSON pointer must be empty or begin with '/' - was: 'foo'	A JSON Pointer must be a Unicode string containing a sequence of zero or more reference tokens, each prefixed by a `/` character.
json.exception.parse_error.108	parse error: escape character '~' must be followed with '0' or '1'	In a JSON Pointer, only `~0` and `~1` are valid escape sequences.
json.exception.parse_error.109	parse error: array index 'one' is not a number	A JSON Pointer array index must be a number.

name / id	example message	description
json.exception.parse_error.110	parse error at 1: cannot read 2 bytes from vector	When parsing CBOR or MessagePack, the byte vector ends before the complete value has been read.
json.exception.parse_error.112	parse error at 1: error reading CBOR; last byte: 0xf8	Not all types of CBOR or MessagePack are supported. This exception occurs if an unsupported byte was read.
json.exception.parse_error.113	parse error at 2: expected a CBOR string; last byte: 0x98	While parsing a map key, a value that is not a string has been read.

Note

For an input with n bytes, 1 is the index of the first character and n+1 is the index of the terminating null byte or the end of file. This also holds true when reading a byte vector (CBOR or MessagePack).

{The following code shows how a `parse_error` exception can be caught.,parse_error}

See also

exception for the base class of the library exceptions
invalid_iterator for exceptions indicating errors with iterators
type_error for exceptions indicating executing a member function with a wrong type
out_of_range for exceptions indicating access out of the defined range
other_error for exceptions indicating other library errors

Since

version 3.0.0

6.9.2.12 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> using nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::parser_callback_t = typename parser::parser_callback_t

per-element parser callback type

With a parser callback function, the result of parsing a JSON text can be influenced. When passed to parse, it is called on certain events (passed as parse_event_t via parameter *event*) with a set recursion depth *depth* and context JSON value *parsed*. The return value of the callback function is a boolean indicating whether the element that emitted the callback shall be kept or not.

We distinguish six scenarios (determined by the event type) in which the callback function can be called. The following table describes the values of the parameters *depth*, *event*, and *parsed*.

parameter *event*	description	parameter *depth*	parameter *parsed*
parse_event_t::object_start	the parser read { and started to process a JSON object	depth of the parent of the JSON object	a JSON value with type discarded

parameter *event*	description	parameter *depth*	parameter *parsed*
parse_event_t::key	the parser read a key of a value in an object	depth of the currently parsed JSON object	a JSON string containing the key
parse_event_t::object_↩ end	the parser read } and finished processing a JSON object	depth of the parent of the JSON object	the parsed JSON object
parse_event_t::array_↩ start	the parser read [and started to process a JS↩ ON array	depth of the parent of the JSON array	a JSON value with type discarded
parse_event_t::array_↩ end	the parser read] and finished processing a JSON array	depth of the parent of the JSON array	the parsed JSON array
parse_event_t::value	the parser finished reading a JSON value	depth of the value	the parsed JSON value

Discarding a value (i.e., returning `false`) has different effects depending on the context in which function was called:

- Discarded values in structured types are skipped. That is, the parser will behave as if the discarded value was never read.
- In case a value outside a structured type is skipped, it is replaced with `null`. This case happens if the top-level element is skipped.

Parameters

`in`	*depth*	the depth of the recursion during parsing
`in`	*event*	an event of type parse_event_t indicating the context in the callback function has been called
`in,out`	*parsed*	the current intermediate parse result; note that writing to this value has no effect for parse_event_t::key events

Returns

Whether the JSON value which called the function during parsing should be kept (`true`) or not (`false`). In the latter case, it is either skipped completely or replaced by an empty discarded object.

See also

parse for examples

Since

version 1.0.0

6.9.2.13 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> using nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::string_t = StringType

a type for a string

RFC 7159 describes JSON strings as follows:

> A string is a sequence of zero or more Unicode characters.

To store objects in C++, a type is defined by the template parameter described below. Unicode values are split by the JSON class into byte-sized characters during deserialization.

Template Parameters

StringType	the container to store strings (e.g., `std::string`). Note this container is used for keys/names in objects, see object_t.

Default type

With the default values for *StringType* (`std::string`), the default value for *string_t* is:

```
std::string
```

Encoding

Strings are stored in UTF-8 encoding. Therefore, functions like `std::string::size()` or `std::string::length()` return the number of bytes in the string rather than the number of characters or glyphs.

String comparison

RFC 7159 states:

> Software implementations are typically required to test names of object members for equality. Implementations that transform the textual representation into sequences of Unicode code units and then perform the comparison numerically, code unit by code unit, are interoperable in the sense that implementations will agree in all cases on equality or inequality of two strings. For example, implementations that compare strings with escaped characters unconverted may incorrectly find that `"a\\\\b"` and `"a\\u005Cb"` are not equal.

This implementation is interoperable as it does compare strings code unit by code unit.

Storage

String values are stored as pointers in a basic_json type. That is, for any access to string values, a pointer of type `string_t*` must be dereferenced.

Since

version 1.0.0

6.9.2.14 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> using nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::type_error = detail::type_error

exception indicating executing a member function with a wrong type

This exception is thrown in case of a type error; that is, a library function is executed on a JSON value whose type does not match the expected semantics.

Exceptions have ids 3xx.

name / id	example message	description
json.exception.type_error.301	cannot create object from initializer list	To create an object from an initializer list, the initializer list must consist only of a list of pairs whose first element is a string. When this constraint is violated, an array is created instead.
json.exception.type_error.302	type must be object, but is array	During implicit or explicit value conversion, the JSON type must be compatible to the target type. For instance, a JSON string can only be converted into string types, but not into numbers or boolean types.
json.exception.type_error.303	incompatible ReferenceType for get_ref, actual type is object	To retrieve a reference to a value stored in a basic_json object with get_ref, the type of the reference must match the value type. For instance, for a JSON array, the *ReferenceType* must be array_t&.
json.exception.type_error.304	cannot use at() with string	The at() member functions can only be executed for certain JSON types.
json.exception.type_error.305	cannot use operator[] with string	The operator[] member functions can only be executed for certain JSON types.
json.exception.type_error.306	cannot use value() with string	The value() member functions can only be executed for certain JSON types.
json.exception.type_error.307	cannot use erase() with string	The erase() member functions can only be executed for certain JSON types.
json.exception.type_error.308	cannot use push_back() with string	The push_back() and operator+= member functions can only be executed for certain JSON types.
json.exception.type_error.309	cannot use insert() with	The insert() member functions can only be executed for certain JSON types.
json.exception.type_error.310	cannot use swap() with number	The swap() member functions can only be executed for certain JSON types.
json.exception.type_error.311	cannot use emplace_back() with string	The emplace_back() member function can only be executed for certain JSON types.
json.exception.type_error.312	cannot use update() with string	The update() member functions can only be executed for certain JSON types.
json.exception.type_error.313	invalid value to unflatten	The unflatten function converts an object whose keys are JSON Pointers back into an arbitrary nested JSON value. The JSON Pointers must not overlap, because then the resulting value would not be well defined.
json.exception.type_error.314	only objects can be unflattened	The unflatten function only works for an object whose keys are JSON Pointers.

name / id	example message	description
json.exception.type_error.315	values in object must be primitive	The unflatten function only works for an object whose keys are JSON Pointers and whose values are primitive.

{The following code shows how a `type_error` exception can be caught.,type_error}

See also

exception for the base class of the library exceptions
parse_error for exceptions indicating a parse error
invalid_iterator for exceptions indicating errors with iterators
out_of_range for exceptions indicating access out of the defined range
other_error for exceptions indicating other library errors

Since

version 3.0.0

6.9.3 Constructor & Destructor Documentation

6.9.3.1 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::basic_json (const value_t *v*) [inline]

create an empty value with a given type

Create an empty JSON value with a given type. The value will be default initialized with an empty value which depends on the type:

Value type	initial value
null	`null`
boolean	`false`
string	`""`
number	`0`
object	`{}`
array	`[]`

Parameters

`in`	*v*	the type of the value to create

Constant.

Strong guarantee: if an exception is thrown, there are no changes to any JSON value.

{The following code shows the constructor for different value_t values,basic_json__value_t}

See also

clear() – restores the postcondition of this constructor

Since

version 1.0.0

6.9.3.2 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::basic_json (std::nullptr_t = nullptr) [inline], [noexcept]

create a null object

Create a `null` JSON value. It either takes a null pointer as parameter (explicitly creating `null`) or no parameter (implicitly creating `null`). The passed null pointer itself is not read – it is only used to choose the right constructor.

Constant.

No-throw guarantee: this constructor never throws exceptions.

{The following code shows the constructor with and without a null pointer parameter.,basic_json__nullptr_t}

Since

version 1.0.0

6.9.3.3 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename CompatibleType , typename U = detail::uncvref_t<CompatibleType>, detail::enable_if_t< not std::is_base_of< std::istream, U >::value andnot std::is_same< U, basic_json_t >::value andnot detail::is_basic_json_nested_type< basic_json_t, U >::value anddetail::has_to_json< basic_json, U >::value, int > = 0> nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::basic_json (CompatibleType && *val*) [inline], [noexcept]

create a JSON value

This is a "catch all" constructor for all compatible JSON types; that is, types for which a `to_json()` method exsits. The constructor forwards the parameter *val* to that method (to `json_serializer<U>::to_json` method with `U = uncvref_t<CompatibleType>`, to be exact).

Template type *CompatibleType* includes, but is not limited to, the following types:

- **arrays**: array_t and all kinds of compatible containers such as `std::vector`, `std::deque`, `std::list`, `std::forward_list`, `std::array`, `std::valarray`, `std::set`, `std::unordered_set`, `std::multiset`, and `std::unordered_multiset` with a `value_type` from which a basic_json value can be constructed.

- **objects**: object_t and all kinds of compatible associative containers such as `std::map`, `std::unordered_map`, `std::multimap`, and `std::unordered_multimap` with a `key_type` compatible to string_t and a `value_type` from which a basic_json value can be constructed.
- **strings**: string_t, string literals, and all compatible string containers can be used.
- **numbers**: number_integer_t, number_unsigned_t, number_float_t, and all convertible number types such as `int`, `size_t`, `int64_t`, `float` or `double` can be used.
- **boolean**: boolean_t / `bool` can be used.

See the examples below.

Template Parameters

CompatibleType	a type such that: • *CompatibleType* is not derived from `std::istream`, • *CompatibleType* is not basic_json (to avoid hijacking copy/move constructors), • *CompatibleType* is not a basic_json nested type (e.g., json_pointer, iterator, etc ...) • json_serializer<U> has a `to_json(basic_json_t&, CompatibleType&&)` method
U	= `uncvref_t<CompatibleType>`

Parameters

`in`	*val*	the value to be forwarded to the respective constructor

Usually linear in the size of the passed *val*, also depending on the implementation of the called `to_json()` method.

Depends on the called constructor. For types directly supported by the library (i.e., all types for which no `to_json()` function was provided), strong guarantee holds: if an exception is thrown, there are no changes to any JSON value.

{The following code shows the constructor with several compatible types.,basic_json__CompatibleType}

Since

version 2.1.0

6.9.3.4 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::basic_json (initializer_list_t *init*, bool *type_deduction* = `true`, value_t *manual_type* = value_t::array) [inline]

create a container (array or object) from an initializer list

Creates a JSON value of type array or object from the passed initializer list *init*. In case *type_deduction* is `true` (default), the type of the JSON value to be created is deducted from the initializer list *init* according to the following rules:

1. If the list is empty, an empty JSON object value `{}` is created.
2. If the list consists of pairs whose first element is a string, a JSON object value is created where the first elements of the pairs are treated as keys and the second elements are as values.
3. In all other cases, an array is created.

The rules aim to create the best fit between a C++ initializer list and JSON values. The rationale is as follows:

1. The empty initializer list is written as `{}` which is exactly an empty JSON object.
2. C++ has no way of describing mapped types other than to list a list of pairs. As JSON requires that keys must be of type string, rule 2 is the weakest constraint one can pose on initializer lists to interpret them as an object.
3. In all other cases, the initializer list could not be interpreted as JSON object type, so interpreting it as JSON array type is safe.

With the rules described above, the following JSON values cannot be expressed by an initializer list:

- the empty array (`[]`): use array(initializer_list_t) with an empty initializer list in this case
- arrays whose elements satisfy rule 2: use array(initializer_list_t) with the same initializer list in this case

Note

When used without parentheses around an empty initializer list, basic_json() is called instead of this function, yielding the JSON null value.

Parameters

`in`	*init*	initializer list with JSON values
`in`	*type_deduction*	internal parameter; when set to `true`, the type of the JSON value is deducted from the initializer list *init*; when set to `false`, the type provided via *manual_type* is forced. This mode is used by the functions array(initializer_list_t) and object(initializer_list_t).
`in`	*manual_type*	internal parameter; when *type_deduction* is set to `false`, the created JSON value will use the provided type (only value_t::array and value_t::object are valid); when *type_deduction* is set to `true`, this parameter has no effect

Exceptions

type_error.301	if *type_deduction* is `false`, *manual_type* is `value_t::object`, but *init* contains an element which is not a pair whose first element is a string. In this case, the constructor could not create an object. If *type_deduction* would have be `true`, an array would have been created. See object(initializer_list_t) for an example.

Linear in the size of the initializer list *init*.

Strong guarantee: if an exception is thrown, there are no changes to any JSON value.

{The example below shows how JSON values are created from initializer lists.,basic_json__list_init_t}

See also

array(initializer_list_t) – create a JSON array value from an initializer list
object(initializer_list_t) – create a JSON object value from an initializer list

Since

version 1.0.0

6.9.3.5 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::basic_json (size_type *cnt*, const basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer > & *val*) [inline]

construct an array with count copies of given value

Constructs a JSON array value by creating *cnt* copies of a passed value. In case *cnt* is `0`, an empty array is created.

Parameters

in	*cnt*	the number of JSON copies of *val* to create
in	*val*	the JSON value to copy

Postcondition

`std::distance(begin(),end()) == cnt` holds.

Linear in *cnt*.

Strong guarantee: if an exception is thrown, there are no changes to any JSON value.

{The following code shows examples for the basic_json(size_type\, const basic_json&) constructor.,basic_json__↩ size_type_basic_json}

Since

version 1.0.0

6.9.3.6 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<class InputIT , typename std::enable_if< std::is_same< InputIT, typename basic_json_t::iterator >::value orstd::is_same< InputIT, typename basic_json_t::const_iterator >::value, int >::type = 0> nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::basic_json (InputIT *first*, InputIT *last*) [inline]

construct a JSON container given an iterator range

Constructs the JSON value with the contents of the range `[first, last)`. The semantics depends on the different types a JSON value can have:

- In case of a null type, invalid_iterator.206 is thrown.
- In case of other primitive types (number, boolean, or string), *first* must be `begin()` and *last* must be `end()`. In this case, the value is copied. Otherwise, invalid_iterator.204 is thrown.
- In case of structured types (array, object), the constructor behaves as similar versions for `std::vector` or `std::map`; that is, a JSON array or object is constructed from the values in the range.

Template Parameters

InputIT	an input iterator type (iterator or const_iterator)

Parameters

`in`	*first*	begin of the range to copy from (included)
`in`	*last*	end of the range to copy from (excluded)

Precondition

Iterators *first* and *last* must be initialized. **This precondition is enforced with an assertion (see warning).** If assertions are switched off, a violation of this precondition yields undefined behavior.
Range `[first, last)` is valid. Usually, this precondition cannot be checked efficiently. Only certain edge cases are detected; see the description of the exceptions below. A violation of this precondition yields undefined behavior.

Warning

A precondition is enforced with a runtime assertion that will result in calling `std::abort` if this precondition is not met. Assertions can be disabled by defining `NDEBUG` at compile time. See `http://en.cppreference.com/w/cpp/error/assert` for more information.

Exceptions

invalid_iterator.201	if iterators *first* and *last* are not compatible (i.e., do not belong to the same JSON value). In this case, the range `[first, last)` is undefined.
invalid_iterator.204	if iterators *first* and *last* belong to a primitive type (number, boolean, or string), but *first* does not point to the first element any more. In this case, the range `[first, last)` is undefined. See example code below.
invalid_iterator.206	if iterators *first* and *last* belong to a null value. In this case, the range `[first, last)` is undefined.

Linear in distance between *first* and *last*.

Strong guarantee: if an exception is thrown, there are no changes to any JSON value.

{The example below shows several ways to create JSON values by specifying a subrange with iterators.,basic_json__InputIt_InputIt}

Since

version 1.0.0

6.9.3.7 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::basic_json (const basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer > & *other*) [inline]

copy constructor

Creates a copy of a given JSON value.

Parameters

in	*other*	the JSON value to copy

Postcondition

```
*this == other
```

Linear in the size of *other*.

Strong guarantee: if an exception is thrown, there are no changes to any JSON value.

This function helps `basic_json` satisfying the `Container` requirements:

- The complexity is linear.
- As postcondition, it holds: `other == basic_json(other)`.

{The following code shows an example for the copy constructor.,basic_json__basic_json}

Since

version 1.0.0

6.9.3.8 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::basic_json (basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer > && *other*) [inline], [noexcept]

move constructor

Move constructor. Constructs a JSON value with the contents of the given value *other* using move semantics. It "steals" the resources from *other* and leaves it as JSON null value.

Parameters

in, out	*other*	value to move to this object

Postcondition

`*this` has the same value as *other* before the call.
other is a JSON null value.

Constant.

No-throw guarantee: this constructor never throws exceptions.

This function helps `basic_json` satisfying the `MoveConstructible` requirements.

{The code below shows the move constructor explicitly called via std::move.,basic_json__moveconstructor}

Since

version 1.0.0

6.9.3.9 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::~basic_json ( ) [inline]

destructor

Destroys the JSON value and frees all allocated memory.

Linear.

This function helps `basic_json` satisfying the `Container` requirements:

- The complexity is linear.
- All stored elements are destroyed and all memory is freed.

Since

version 1.0.0

6.9.4 Member Function Documentation

6.9.4.1 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> static basic_json nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::array (initializer_list_t *init* = `{}`) `[inline], [static]`

explicitly create an array from an initializer list

Creates a JSON array value from a given initializer list. That is, given a list of values `a, b, c`, creates the JSON value `[a, b, c]`. If the initializer list is empty, the empty array `[]` is created.

Note

This function is only needed to express two edge cases that cannot be realized with the initializer list constructor (basic_json(initializer_list_t, bool, value_t)). These cases are:

1. creating an array whose elements are all pairs whose first element is a string – in this case, the initializer list constructor would create an object, taking the first elements as keys
2. creating an empty array – passing the empty initializer list to the initializer list constructor yields an empty object

Parameters

`in`	*init*	initializer list with JSON values to create an array from (optional)

Returns

JSON array value

Linear in the size of *init*.

Strong guarantee: if an exception is thrown, there are no changes to any JSON value.

{The following code shows an example for the `array` function.,array}

See also

basic_json(initializer_list_t, bool, value_t) – create a JSON value from an initializer list
object(initializer_list_t) – create a JSON object value from an initializer list

Since

version 1.0.0

6.9.4.2 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> reference nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::at (size_type *idx*) [inline]

access specified array element with bounds checking

Returns a reference to the element at specified location *idx*, with bounds checking.

Parameters

in	*idx*	index of the element to access

Returns

reference to the element at index *idx*

Exceptions

type_error.304	if the JSON value is not an array; in this case, calling `at` with an index makes no sense. See example below.
out_of_range.401	if the index *idx* is out of range of the array; that is, `idx >= size()`. See example below.

Strong guarantee: if an exception is thrown, there are no changes in the JSON value.

Constant.

Since

version 1.0.0

{The example below shows how array elements can be read and written using `at()`. It also demonstrates the different exceptions that can be thrown.,at__size_type}

6.9.4.3 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> const_reference nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::at (size_type *idx*) const [inline]

access specified array element with bounds checking

Returns a const reference to the element at specified location *idx*, with bounds checking.

Parameters

in	*idx*	index of the element to access

Returns

const reference to the element at index *idx*

Exceptions

type_error.304	if the JSON value is not an array; in this case, calling `at` with an index makes no sense. See example below.
out_of_range.401	if the index *idx* is out of range of the array; that is, `idx >= size()`. See example below.

Strong guarantee: if an exception is thrown, there are no changes in the JSON value.

Constant.

Since

version 1.0.0

{The example below shows how array elements can be read using `at()`. It also demonstrates the different exceptions that can be thrown., at__size_type_const}

6.9.4.4 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> reference nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::at (const typename object_t::key_type & *key*) [inline]

access specified object element with bounds checking

Returns a reference to the element at with specified key *key*, with bounds checking.

Parameters

in	*key*	key of the element to access

Returns

reference to the element at key *key*

Exceptions

type_error.304	if the JSON value is not an object; in this case, calling `at` with a key makes no sense. See example below.
out_of_range.403	if the key *key* is is not stored in the object; that is, `find(key) == end()`. See example below.

Strong guarantee: if an exception is thrown, there are no changes in the JSON value.

Logarithmic in the size of the container.

See also

operator[ ](const typename object_t::key_type&) for unchecked access by reference
value() for access by value with a default value

Since

version 1.0.0

{The example below shows how object elements can be read and written using `at()`. It also demonstrates the different exceptions that can be thrown.,at__object_t_key_type}

6.9.4.5 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> const_reference nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::at (const typename object_t::key_type & *key*) const [inline]

access specified object element with bounds checking

Returns a const reference to the element at with specified key *key*, with bounds checking.

Parameters

in	*key*	key of the element to access

Returns

const reference to the element at key *key*

Exceptions

type_error.304	if the JSON value is not an object; in this case, calling `at` with a key makes no sense. See example below.
out_of_range.403	if the key *key* is is not stored in the object; that is, `find(key) == end()`. See example below.

Strong guarantee: if an exception is thrown, there are no changes in the JSON value.

Logarithmic in the size of the container.

See also

operator[ ](const typename object_t::key_type&) for unchecked access by reference
value() for access by value with a default value

Since

version 1.0.0

{The example below shows how object elements can be read using `at()`. It also demonstrates the different exceptions that can be thrown., at__object_t_key_type_const}

6.9.4.6 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> reference nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::at (const json_pointer & *ptr*) [inline]

access specified element via JSON Pointer

Returns a reference to the element at with specified JSON pointer *ptr*, with bounds checking.

Parameters

in	*ptr*	JSON pointer to the desired element

Returns

reference to the element pointed to by *ptr*

Exceptions

parse_error.106	if an array index in the passed JSON pointer *ptr* begins with '0'. See example below.
parse_error.109	if an array index in the passed JSON pointer *ptr* is not a number. See example below.
out_of_range.401	if an array index in the passed JSON pointer *ptr* is out of range. See example below.
out_of_range.402	if the array index '-' is used in the passed JSON pointer *ptr*. As `at` provides checked access (and no elements are implicitly inserted), the index '-' is always invalid. See example below.
out_of_range.404	if the JSON pointer *ptr* can not be resolved. See example below.

Strong guarantee: if an exception is thrown, there are no changes in the JSON value.

Constant.

Since

version 2.0.0

{The behavior is shown in the example.,at_json_pointer}

6.9.4.7 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> const_reference nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::at (const json_pointer & *ptr*) const [inline]

access specified element via JSON Pointer

Returns a const reference to the element at with specified JSON pointer *ptr*, with bounds checking.

Parameters

in	*ptr*	JSON pointer to the desired element

Returns

reference to the element pointed to by *ptr*

Exceptions

parse_error.106	if an array index in the passed JSON pointer *ptr* begins with '0'. See example below.
parse_error.109	if an array index in the passed JSON pointer *ptr* is not a number. See example below.
out_of_range.401	if an array index in the passed JSON pointer *ptr* is out of range. See example below.
out_of_range.402	if the array index '-' is used in the passed JSON pointer *ptr*. As `at` provides checked access (and no elements are implicitly inserted), the index '-' is always invalid. See example below.
out_of_range.404	if the JSON pointer *ptr* can not be resolved. See example below.

Strong guarantee: if an exception is thrown, there are no changes in the JSON value.

Constant.

Since

version 2.0.0

{The behavior is shown in the example.,at_json_pointer_const}

6.9.4.8 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> reference nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::back ( ) [inline]

access the last element

Returns a reference to the last element in the container. For a JSON container `c`, the expression `c.back()` is equivalent to

```
auto tmp = c.end();
--tmp;
return *tmp;
```

Returns

In case of a structured type (array or object), a reference to the last element is returned. In case of number, string, or boolean values, a reference to the value is returned.

Constant.

Precondition

The JSON value must not be `null` (would throw `std::out_of_range`) or an empty array or object (undefined behavior, **guarded by assertions**).

Postcondition

The JSON value remains unchanged.

Exceptions

invalid_iterator.214	when called on a `null` value. See example below.

{The following code shows an example for `back()`.,back}

See also

front() – access the first element

Since

version 1.0.0

6.9.4.9 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> const_reference nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::back ( ) const [inline]

access the last element

Returns a reference to the last element in the container. For a JSON container `c`, the expression `c.back()` is equivalent to

```
auto tmp = c.end();
--tmp;
return *tmp;
```

Returns

In case of a structured type (array or object), a reference to the last element is returned. In case of number, string, or boolean values, a reference to the value is returned.

Constant.

Precondition

The JSON value must not be `null` (would throw `std::out_of_range`) or an empty array or object (undefined behavior, **guarded by assertions**).

Postcondition

The JSON value remains unchanged.

Exceptions

invalid_iterator.214	when called on a `null` value. See example below.

{The following code shows an example for `back()`.,back}

See also

front() – access the first element

Since

version 1.0.0

6.9.4.10 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> iterator nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::begin ( ) [inline], [noexcept]

returns an iterator to the first element

Returns an iterator to the first element.

Returns

iterator to the first element

Constant.

This function helps `basic_json` satisfying the `Container` requirements:

- The complexity is constant.

{The following code shows an example for `begin()`.,begin}

See also

cbegin() – returns a const iterator to the beginning
end() – returns an iterator to the end
cend() – returns a const iterator to the end

Since

version 1.0.0

6.9.4.11 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> const_iterator nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::begin ( ) const [inline], [noexcept]

returns a const iterator to the first element

Returns a const iterator to the first element.

Returns

const iterator to the first element

Constant.

This function helps `basic_json` satisfying the `Container` requirements:

- The complexity is constant.
- Has the semantics of `const_cast<const basic_json&>(*this).begin()`.

{The following code shows an example for `cbegin()`.,cbegin}

See also

begin() – returns an iterator to the beginning
end() – returns an iterator to the end
cend() – returns a const iterator to the end

Since

version 1.0.0

6.9.4.12 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> const_iterator nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::cbegin ( ) const [inline], [noexcept]

returns a const iterator to the first element

Returns a const iterator to the first element.

Returns

const iterator to the first element

Constant.

This function helps `basic_json` satisfying the `Container` requirements:

- The complexity is constant.
- Has the semantics of `const_cast<const basic_json&>(*this).begin()`.

{The following code shows an example for `cbegin()`.,cbegin}

See also

begin() – returns an iterator to the beginning
end() – returns an iterator to the end
cend() – returns a const iterator to the end

Since

version 1.0.0

6.9.4.13 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> const_iterator nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::cend ( ) const [inline], [noexcept]

returns a const iterator to one past the last element

Returns a const iterator to one past the last element.

Returns

const iterator one past the last element

Constant.

This function helps `basic_json` satisfying the `Container` requirements:

- The complexity is constant.
- Has the semantics of `const_cast<const basic_json&>(*this).end()`.

{The following code shows an example for `cend()`.,cend}

See also

end() – returns an iterator to the end
begin() – returns an iterator to the beginning
cbegin() – returns a const iterator to the beginning

Since

version 1.0.0

6.9.4.14 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> void nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::clear ( ) [inline], [noexcept]

clears the contents

Clears the content of a JSON value and resets it to the default value as if basic_json(value_t) would have been called with the current value type from type():

Value type	initial value
null	`null`
boolean	`false`
string	`""`
number	`0`
object	`{}`
array	`[]`

Postcondition

Has the same effect as calling

```
*this = basic_json(type());
```

{The example below shows the effect of `clear()` to different JSON types.,clear}

Linear in the size of the JSON value.

All iterators, pointers and references related to this container are invalidated.

No-throw guarantee: this function never throws exceptions.

See also

basic_json(value_t) – constructor that creates an object with the same value than calling `clear()`

Since

version 1.0.0

6.9.4.15 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename KeyT > size_type nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::count (KeyT && *key*) const [inline]

returns the number of occurrences of a key in a JSON object

Returns the number of elements with key *key*. If ObjectType is the default `std::map` type, the return value will always be `0` (*key* was not found) or `1` (*key* was found).

Note

This method always returns `0` when executed on a JSON type that is not an object.

Parameters

in	*key*	key value of the element to count

Returns

Number of elements with key *key*. If the JSON value is not an object, the return value will be `0`.

Logarithmic in the size of the JSON object.

{The example shows how `count()` is used.,count}

Since

version 1.0.0

6.9.4.16 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> const_reverse_iterator nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::crbegin ( ) const [inline], [noexcept]

returns a const reverse iterator to the last element

Returns a const iterator to the reverse-beginning; that is, the last element.

Constant.

This function helps `basic_json` satisfying the `ReversibleContainer` requirements:

- The complexity is constant.
- Has the semantics of `const_cast<const basic_json&>(*this).rbegin()`.

{The following code shows an example for `crbegin()`.,crbegin}

See also

rbegin() – returns a reverse iterator to the beginning
rend() – returns a reverse iterator to the end
crend() – returns a const reverse iterator to the end

Since

version 1.0.0

6.9.4.17 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> const_reverse_iterator nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::crend ( ) const [inline], [noexcept]

returns a const reverse iterator to one before the first

Returns a const reverse iterator to the reverse-end; that is, one before the first element.

Constant.

This function helps basic_json satisfying the ReversibleContainer requirements:

- The complexity is constant.
- Has the semantics of const_cast<const basic_json&>(*this).rend().

{The following code shows an example for crend().,crend}

See also

rend() – returns a reverse iterator to the end
rbegin() – returns a reverse iterator to the beginning
crbegin() – returns a const reverse iterator to the beginning

Since

version 1.0.0

6.9.4.18 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> static basic_json nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::diff (const basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer > & *source*, const basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer > & *target*, const std::string & *path* = "") [inline], [static]

creates a diff as a JSON patch

Creates a JSON Patch so that value *source* can be changed into the value *target* by calling patch function.

Invariant

For two JSON values *source* and *target*, the following code yields always true:

```
source.patch(diff(source, target)) == target;
```

Note

Currently, only remove, add, and replace operations are generated.

Parameters

in	*source*	JSON value to compare from
in	*target*	JSON value to compare against
in	*path*	helper value to create JSON pointers

Returns

a JSON patch to convert the *source* to *target*

Linear in the lengths of *source* and *target*.

{The following code shows how a JSON patch is created as a diff for two JSON values.,diff}

See also

patch – apply a JSON patch
RFC 6902 (JSON Patch)

Since

version 2.0.0

6.9.4.19 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> string_t nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::dump (const int *indent* = -1, const char *indent_char* = ' ', const bool *ensure_ascii* = false) const [inline]

serialization

Serialization function for JSON values. The function tries to mimic Python's `json.dumps()` function, and currently supports its *indent* and *ensure_ascii* parameters.

Parameters

in	*indent*	If indent is nonnegative, then array elements and object members will be pretty-printed with that indent level. An indent level of 0 will only insert newlines. -1 (the default) selects the most compact representation.
in	*indent_char*	The character to use for indentation if *indent* is greater than 0. The default is (space).
in	*ensure_ascii*	If *ensure_ascii* is true, all non-ASCII characters in the output are escaped with sequences, and the result consists of ASCII characters only.

Returns

string containing the serialization of the JSON value

Linear.

Strong guarantee: if an exception is thrown, there are no changes in the JSON value.

{The following example shows the effect of different *indent*\, *indent_char*\, and *ensure_ascii* parameters to the result of the serialization.,dump}

See also

https://docs.python.org/2/library/json.html#json.dump

Since

version 1.0.0; indentation character *indent_char* and option *ensure_ascii* added in version 3.0.0

6.9.4.20 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<class... Args> std::pair<iterator, bool> nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::emplace (Args &&... *args*) [inline]

add an object to an object if key does not exist

Inserts a new element into a JSON object constructed in-place with the given *args* if there is no element with the key in the container. If the function is called on a JSON null value, an empty object is created before appending the value created from *args*.

Parameters

in	*args*	arguments to forward to a constructor of basic_json

Template Parameters

Args	compatible types to create a basic_json object

Returns

a pair consisting of an iterator to the inserted element, or the already-existing element if no insertion happened, and a bool denoting whether the insertion took place.

Exceptions

type_error.311	when called on a type other than JSON object or null; example: "`cannot use emplace() with number`"

Logarithmic in the size of the container, O(log(`size()`)).

{The example shows how `emplace()` can be used to add elements to a JSON object. Note how the `null` value was silently converted to a JSON object. Further note how no value is added if there was already one value stored with the same key.,emplace}

Since

version 2.0.8

6.9.4.21 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<class... Args> void nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::emplace_back (Args &&... *args*) [inline]

add an object to an array

Creates a JSON value from the passed parameters *args* to the end of the JSON value. If the function is called on a JSON null value, an empty array is created before appending the value created from *args*.

Parameters

in	*args*	arguments to forward to a constructor of basic_json

Template Parameters

Args	compatible types to create a basic_json object

Exceptions

type_error.311	when called on a type other than JSON array or null; example: "`cannot use emplace_back() with number`"

Amortized constant.

{The example shows how `push_back()` can be used to add elements to a JSON array. Note how the `null` value was silently converted to a JSON array.,emplace_back}

Since

version 2.0.8

6.9.4.22 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> bool nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::empty ( ) const [inline], [noexcept]

checks whether the container is empty.

Checks if a JSON value has no elements (i.e. whether its size is `0`).

Returns

The return value depends on the different types and is defined as follows:

Value type	return value
null	`true`
boolean	`false`
string	`false`
number	`false`
object	result of function `object_t::empty()`
array	result of function `array_t::empty()`

{The following code uses `empty()` to check if a JSON object contains any elements.,empty}

Constant, as long as array_t and object_t satisfy the Container concept; that is, their `empty()` functions have constant complexity.

No changes.

No-throw guarantee: this function never throws exceptions.

Note

This function does not return whether a string stored as JSON value is empty - it returns whether the JSON container itself is empty which is false in the case of a string.

This function helps `basic_json` satisfying the `Container` requirements:

- The complexity is constant.
- Has the semantics of `begin() == end()`.

See also

size() – returns the number of elements

Since

version 1.0.0

6.9.4.23 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> iterator nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::end ( ) [inline], [noexcept]

returns an iterator to one past the last element

Returns an iterator to one past the last element.

Returns

iterator one past the last element

Constant.

This function helps `basic_json` satisfying the `Container` requirements:

- The complexity is constant.

{The following code shows an example for `end()`.,end}

See also

cend() – returns a const iterator to the end
begin() – returns an iterator to the beginning
cbegin() – returns a const iterator to the beginning

Since

version 1.0.0

6.9.4.24 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> const_iterator nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::end ( ) const [inline], [noexcept]

returns a const iterator to one past the last element

Returns a const iterator to one past the last element.

Returns

const iterator one past the last element

Constant.

This function helps `basic_json` satisfying the `Container` requirements:

- The complexity is constant.
- Has the semantics of `const_cast<const basic_json&>(*this).end()`.

{The following code shows an example for `cend()`.,cend}

See also

end() – returns an iterator to the end
begin() – returns an iterator to the beginning
cbegin() – returns a const iterator to the beginning

Since

version 1.0.0

6.9.4.25 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<class IteratorType , typename std::enable_if< std::is_same< IteratorType, typename basic_json_t::iterator >::value orstd::is_same< IteratorType, typename basic_json_t::const_iterator >::value, int >::type = 0> IteratorType nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::erase (IteratorType *pos*) [inline]

remove element given an iterator

Removes the element specified by iterator *pos*. The iterator *pos* must be valid and dereferenceable. Thus the `end()` iterator (which is valid, but is not dereferenceable) cannot be used as a value for *pos*.

If called on a primitive type other than `null`, the resulting JSON value will be `null`.

Parameters

`in`	*pos*	iterator to the element to remove

Returns

Iterator following the last removed element. If the iterator *pos* refers to the last element, the `end()` iterator is returned.

Template Parameters

IteratorType	an iterator or const_iterator

Postcondition

Invalidates iterators and references at or after the point of the erase, including the `end()` iterator.

Exceptions

type_error.307	if called on a `null` value; example: `"cannot use erase() with null"`
invalid_iterator.202	if called on an iterator which does not belong to the current JSON value; example: `"iterator does not fit current value"`
invalid_iterator.205	if called on a primitive type with invalid iterator (i.e., any iterator which is not `begin()`); example: `"iterator out of range"`

The complexity depends on the type:

- objects: amortized constant
- arrays: linear in distance between *pos* and the end of the container
- strings: linear in the length of the string
- other types: constant

{The example shows the result of `erase()` for different JSON types.,erase__IteratorType}

See also

erase(IteratorType, IteratorType) – removes the elements in the given range
erase(const typename object_t::key_type&) – removes the element from an object at the given key
erase(const size_type) – removes the element from an array at the given index

Since

version 1.0.0

6.9.4.26 **template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<class IteratorType , typename std::enable_if< std::is_same< IteratorType, typename basic_json_t::iterator >::value orstd::is_same< IteratorType, typename basic_json_t::const_iterator >::value, int >::type = 0> IteratorType nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::erase (IteratorType *first*, IteratorType *last*)** `[inline]`

remove elements given an iterator range

Removes the element specified by the range `[first; last)`. The iterator *first* does not need to be dereferenceable if `first == last`: erasing an empty range is a no-op.

If called on a primitive type other than `null`, the resulting JSON value will be `null`.

Parameters

`in`	*first*	iterator to the beginning of the range to remove
`in`	*last*	iterator past the end of the range to remove

Returns

Iterator following the last removed element. If the iterator *second* refers to the last element, the `end()` iterator is returned.

Template Parameters

IteratorType	an iterator or const_iterator

Postcondition

Invalidates iterators and references at or after the point of the erase, including the `end()` iterator.

Exceptions

type_error.307	if called on a `null` value; example: `"cannot use erase() with null"`
invalid_iterator.203	if called on iterators which does not belong to the current JSON value; example: `"iterators do not fit current value"`
invalid_iterator.204	if called on a primitive type with invalid iterators (i.e., if `first != begin()` and `last != end()`); example: `"iterators out of range"`

The complexity depends on the type:

- objects: `log(size()) + std::distance(first, last)`
- arrays: linear in the distance between *first* and *last*, plus linear in the distance between *last* and end of the container
- strings: linear in the length of the string
- other types: constant

{The example shows the result of `erase()` for different JSON types.,erase__IteratorType_IteratorType}

See also

erase(IteratorType) – removes the element at a given position
erase(const typename object_t::key_type&) – removes the element from an object at the given key
erase(const size_type) – removes the element from an array at the given index

Since

version 1.0.0

6.9.4.27 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> size_type nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::erase (const typename object_t::key_type & *key*) [inline]

remove element from a JSON object given a key

Removes elements from a JSON object with the key value *key*.

Parameters

in	*key*	value of the elements to remove

Returns

Number of elements removed. If *ObjectType* is the default `std::map` type, the return value will always be `0` (*key* was not found) or `1` (*key* was found).

Postcondition

References and iterators to the erased elements are invalidated. Other references and iterators are not affected.

Exceptions

type_error.307	when called on a type other than JSON object; example: `"cannot use erase() with null"`

`log(size()) + count(key)`

{The example shows the effect of `erase()`.,erase__key_type}

See also

erase(IteratorType) – removes the element at a given position
erase(IteratorType, IteratorType) – removes the elements in the given range
erase(const size_type) – removes the element from an array at the given index

Since

version 1.0.0

6.9.4.28 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> void nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::erase (const size_type *idx*) [inline]

remove element from a JSON array given an index

Removes element from a JSON array at the index *idx*.

Parameters

`in`	*idx*	index of the element to remove

Exceptions

type_error.307	when called on a type other than JSON object; example: `"cannot use erase() with null"`
out_of_range.401	when `idx >= size()`; example: `"array index 17 is out of range"`

Linear in distance between *idx* and the end of the container.

{The example shows the effect of `erase()`.,erase__size_type}

See also

erase(IteratorType) – removes the element at a given position
erase(IteratorType, IteratorType) – removes the elements in the given range
erase(const typename object_t::key_type&) – removes the element from an object at the given key

Since

version 1.0.0

6.9.4.29 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename KeyT > iterator nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::find (KeyT && *key*) [inline]

find an element in a JSON object

Finds an element in a JSON object with key equivalent to *key*. If the element is not found or the JSON value is not an object, end() is returned.

Note

This method always returns end() when executed on a JSON type that is not an object.

Parameters

in	*key*	key value of the element to search for.

Returns

Iterator to an element with key equivalent to *key*. If no such element is found or the JSON value is not an object, past-the-end (see end()) iterator is returned.

Logarithmic in the size of the JSON object.

{The example shows how `find()` is used.,find__key_type}

Since

version 1.0.0

6.9.4.30 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename KeyT > const_iterator nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::find (KeyT && *key*) const [inline]

find an element in a JSON object

find an element in a JSON object Finds an element in a JSON object with key equivalent to *key*. If the element is not found or the JSON value is not an object, end() is returned.

Note

This method always returns end() when executed on a JSON type that is not an object.

Parameters

in	*key*	key value of the element to search for.

Returns

Iterator to an element with key equivalent to *key*. If no such element is found or the JSON value is not an object, past-the-end (see end()) iterator is returned.

Logarithmic in the size of the JSON object.

{The example shows how `find()` is used.,find__key_type}

Since

version 1.0.0

6.9.4.31 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> basic_json nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::flatten ( ) const [inline]

return flattened JSON value

The function creates a JSON object whose keys are JSON pointers (see RFC 6901) and whose values are all primitive. The original JSON value can be restored using the unflatten() function.

Returns

an object that maps JSON pointers to primitive values

Note

Empty objects and arrays are flattened to `null` and will not be reconstructed correctly by the unflatten() function.

Linear in the size the JSON value.

{The following code shows how a JSON object is flattened to an object whose keys consist of JSON pointers.,flatten}

See also

unflatten() for the reverse function

Since

version 2.0.0

6.9.4.32 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> static basic_json nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::from_cbor (detail::input_adapter *i,* const bool *strict* = `true`) [inline], [static]

create a JSON value from an input in CBOR format

Deserializes a given input *i* to a JSON value using the CBOR (Concise Binary Object Representation) serialization format.

The library maps CBOR types to JSON value types as follows:

CBOR type	JSON value type	first byte
Integer	number_unsigned	0x00..0x17
Unsigned integer	number_unsigned	0x18
Unsigned integer	number_unsigned	0x19
Unsigned integer	number_unsigned	0x1a
Unsigned integer	number_unsigned	0x1b
Negative integer	number_integer	0x20..0x37
Negative integer	number_integer	0x38
Negative integer	number_integer	0x39
Negative integer	number_integer	0x3a
Negative integer	number_integer	0x3b
Negative integer	number_integer	0x40..0x57
UTF-8 string	string	0x60..0x77
UTF-8 string	string	0x78
UTF-8 string	string	0x79
UTF-8 string	string	0x7a
UTF-8 string	string	0x7b
UTF-8 string	string	0x7f
array	array	0x80..0x97
array	array	0x98
array	array	0x99
array	array	0x9a
array	array	0x9b
array	array	0x9f
map	object	0xa0..0xb7
map	object	0xb8
map	object	0xb9
map	object	0xba
map	object	0xbb
map	object	0xbf
False	`false`	0xf4
True	`true`	0xf5
Nill	`null`	0xf6
Half-Precision Float	number_float	0xf9
Single-Precision Float	number_float	0xfa
Double-Precision Float	number_float	0xfb

Warning

The mapping is **incomplete** in the sense that not all CBOR types can be converted to a JSON value. The following CBOR types are not supported and will yield parse errors (parse_error.112):

- byte strings (0x40..0x5f)
- date/time (0xc0..0xc1)
- bignum (0xc2..0xc3)
- decimal fraction (0xc4)
- bigfloat (0xc5)
- tagged items (0xc6..0xd4, 0xd8..0xdb)
- expected conversions (0xd5..0xd7)
- simple values (0xe0..0xf3, 0xf8)
- undefined (0xf7)

CBOR allows map keys of any type, whereas JSON only allows strings as keys in object values. Therefore, CBOR maps with keys other than UTF-8 strings are rejected (parse_error.113).

Note

Any CBOR output created to_cbor can be successfully parsed by from_cbor.

Parameters

`in`	*i*	an input in CBOR format convertible to an input adapter
`in`	*strict*	whether to expect the input to be consumed until EOF (true by default)

Returns

deserialized JSON value

Exceptions

parse_error.110	if the given input ends prematurely or the end of file was not reached when *strict* was set to true
parse_error.112	if unsupported features from CBOR were used in the given input *v* or if the input is not valid CBOR
parse_error.113	if a string was expected as map key, but not found

Linear in the size of the input *i*.

{The example shows the deserialization of a byte vector in CBOR format to a JSON value.,from_cbor}

See also

http://cbor.io
to_cbor(const basic_json&) for the analogous serialization
from_msgpack(detail::input_adapter, const bool) for the related MessagePack format

Since

version 2.0.9; parameter *start_index* since 2.1.1; changed to consume input adapters, removed start_index parameter, and added *strict* parameter since 3.0.0

6.9.4.33 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename A1 , typename A2 , detail::enable_if_t< std::is_constructible< detail::input_adapter, A1, A2 >::value, int > = 0> static basic_json nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::from_cbor (A1 && *a1,* A2 && *a2,* const bool *strict* = `true`) [inline], [static]

create a JSON value from an input in CBOR format

Deserializes a given input *i* to a JSON value using the CBOR (Concise Binary Object Representation) serialization format.

The library maps CBOR types to JSON value types as follows:

CBOR type	JSON value type	first byte
Integer	number_unsigned	0x00..0x17
Unsigned integer	number_unsigned	0x18
Unsigned integer	number_unsigned	0x19
Unsigned integer	number_unsigned	0x1a
Unsigned integer	number_unsigned	0x1b
Negative integer	number_integer	0x20..0x37
Negative integer	number_integer	0x38
Negative integer	number_integer	0x39
Negative integer	number_integer	0x3a
Negative integer	number_integer	0x3b
Negative integer	number_integer	0x40..0x57
UTF-8 string	string	0x60..0x77
UTF-8 string	string	0x78
UTF-8 string	string	0x79
UTF-8 string	string	0x7a
UTF-8 string	string	0x7b
UTF-8 string	string	0x7f
array	array	0x80..0x97
array	array	0x98
array	array	0x99
array	array	0x9a
array	array	0x9b
array	array	0x9f
map	object	0xa0..0xb7
map	object	0xb8
map	object	0xb9
map	object	0xba
map	object	0xbb
map	object	0xbf
False	`false`	0xf4
True	`true`	0xf5
Nill	`null`	0xf6
Half-Precision Float	number_float	0xf9
Single-Precision Float	number_float	0xfa
Double-Precision Float	number_float	0xfb

Warning

The mapping is **incomplete** in the sense that not all CBOR types can be converted to a JSON value. The following CBOR types are not supported and will yield parse errors (parse_error.112):

- byte strings (0x40..0x5f)
- date/time (0xc0..0xc1)
- bignum (0xc2..0xc3)
- decimal fraction (0xc4)
- bigfloat (0xc5)
- tagged items (0xc6..0xd4, 0xd8..0xdb)
- expected conversions (0xd5..0xd7)
- simple values (0xe0..0xf3, 0xf8)
- undefined (0xf7)

CBOR allows map keys of any type, whereas JSON only allows strings as keys in object values. Therefore, CBOR maps with keys other than UTF-8 strings are rejected (parse_error.113).

Note

Any CBOR output created to_cbor can be successfully parsed by from_cbor.

Parameters

in	*i*	an input in CBOR format convertible to an input adapter
in	*strict*	whether to expect the input to be consumed until EOF (true by default)

Returns

deserialized JSON value

Exceptions

parse_error.110	if the given input ends prematurely or the end of file was not reached when *strict* was set to true
parse_error.112	if unsupported features from CBOR were used in the given input *v* or if the input is not valid CBOR
parse_error.113	if a string was expected as map key, but not found

Linear in the size of the input *i*.

{The example shows the deserialization of a byte vector in CBOR format to a JSON value.,from_cbor}

See also

http://cbor.io
to_cbor(const basic_json&) for the analogous serialization
from_msgpack(detail::input_adapter, const bool) for the related MessagePack format

Since

version 2.0.9; parameter *start_index* since 2.1.1; changed to consume input adapters, removed start_index parameter, and added *strict* parameter since 3.0.0

6.9.4.34 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> static basic_json nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::from_msgpack (detail::input_adapter *i,* const bool *strict* = `true`) `[inline],[static]`

create a JSON value from an input in MessagePack format

Deserializes a given input *i* to a JSON value using the MessagePack serialization format.

The library maps MessagePack types to JSON value types as follows:

MessagePack type	JSON value type	first byte
positive fixint	number_unsigned	0x00..0x7f
fixmap	object	0x80..0x8f
fixarray	array	0x90..0x9f
fixstr	string	0xa0..0xbf
nil	`null`	0xc0
false	`false`	0xc2
true	`true`	0xc3
float 32	number_float	0xca
float 64	number_float	0xcb
uint 8	number_unsigned	0xcc
uint 16	number_unsigned	0xcd
uint 32	number_unsigned	0xce
uint 64	number_unsigned	0xcf
int 8	number_integer	0xd0
int 16	number_integer	0xd1
int 32	number_integer	0xd2
int 64	number_integer	0xd3
str 8	string	0xd9
str 16	string	0xda
str 32	string	0xdb
array 16	array	0xdc
array 32	array	0xdd
map 16	object	0xde
map 32	object	0xdf
negative fixint	number_integer	0xe0-0xff

Warning

The mapping is **incomplete** in the sense that not all MessagePack types can be converted to a JSON value. The following MessagePack types are not supported and will yield parse errors:

- bin 8 - bin 32 (0xc4..0xc6)
- ext 8 - ext 32 (0xc7..0xc9)
- fixext 1 - fixext 16 (0xd4..0xd8)

Note

Any MessagePack output created to_msgpack can be successfully parsed by from_msgpack.

Parameters

in	*i*	an input in MessagePack format convertible to an input adapter
in	*strict*	whether to expect the input to be consumed until EOF (true by default)

Exceptions

parse_error.110	if the given input ends prematurely or the end of file was not reached when *strict* was set to true
parse_error.112	if unsupported features from MessagePack were used in the given input *i* or if the input is not valid MessagePack
parse_error.113	if a string was expected as map key, but not found

Linear in the size of the input *i*.

{The example shows the deserialization of a byte vector in MessagePack format to a JSON value.,from_msgpack}

See also

http://msgpack.org
to_msgpack(const basic_json&) for the analogous serialization
from_cbor(detail::input_adapter, const bool) for the related CBOR format

Since

version 2.0.9; parameter *start_index* since 2.1.1; changed to consume input adapters, removed start_index parameter, and added *strict* parameter since 3.0.0

6.9.4.35 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename A1 , typename A2 , detail::enable_if_t< std::is_constructible< detail::input_adapter, A1, A2 >::value, int > = 0> static basic_json nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::from_msgpack (A1 && *a1,* A2 && *a2,* const bool *strict* = true) [inline], [static]

create a JSON value from an input in MessagePack format

Deserializes a given input *i* to a JSON value using the MessagePack serialization format.

The library maps MessagePack types to JSON value types as follows:

MessagePack type	JSON value type	first byte
positive fixint	number_unsigned	0x00..0x7f
fixmap	object	0x80..0x8f
fixarray	array	0x90..0x9f
fixstr	string	0xa0..0xbf
nil	null	0xc0
false	false	0xc2
true	true	0xc3

MessagePack type	JSON value type	first byte
float 32	number_float	0xca
float 64	number_float	0xcb
uint 8	number_unsigned	0xcc
uint 16	number_unsigned	0xcd
uint 32	number_unsigned	0xce
uint 64	number_unsigned	0xcf
int 8	number_integer	0xd0
int 16	number_integer	0xd1
int 32	number_integer	0xd2
int 64	number_integer	0xd3
str 8	string	0xd9
str 16	string	0xda
str 32	string	0xdb
array 16	array	0xdc
array 32	array	0xdd
map 16	object	0xde
map 32	object	0xdf
negative fixint	number_integer	0xe0-0xff

Warning

The mapping is **incomplete** in the sense that not all MessagePack types can be converted to a JSON value. The following MessagePack types are not supported and will yield parse errors:

- bin 8 - bin 32 (0xc4..0xc6)
- ext 8 - ext 32 (0xc7..0xc9)
- fixext 1 - fixext 16 (0xd4..0xd8)

Note

Any MessagePack output created to_msgpack can be successfully parsed by from_msgpack.

Parameters

in	*i*	an input in MessagePack format convertible to an input adapter
in	*strict*	whether to expect the input to be consumed until EOF (true by default)

Exceptions

parse_error.110	if the given input ends prematurely or the end of file was not reached when *strict* was set to true
parse_error.112	if unsupported features from MessagePack were used in the given input *i* or if the input is not valid MessagePack
parse_error.113	if a string was expected as map key, but not found

Linear in the size of the input *i*.

{The example shows the deserialization of a byte vector in MessagePack format to a JSON value.,from_msgpack}

See also

http://msgpack.org
to_msgpack(const basic_json&) for the analogous serialization
from_cbor(detail::input_adapter, const bool) for the related CBOR format

Since

version 2.0.9; parameter *start_index* since 2.1.1; changed to consume input adapters, removed start_index parameter, and added *strict* parameter since 3.0.0

6.9.4.36 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> reference nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::front ( ) [inline]

access the first element

Returns a reference to the first element in the container. For a JSON container `c`, the expression `c.front()` is equivalent to `*c.begin()`.

Returns

In case of a structured type (array or object), a reference to the first element is returned. In case of number, string, or boolean values, a reference to the value is returned.

Constant.

Precondition

The JSON value must not be `null` (would throw `std::out_of_range`) or an empty array or object (undefined behavior, **guarded by assertions**).

Postcondition

The JSON value remains unchanged.

Exceptions

invalid_iterator.214	when called on `null` value

{The following code shows an example for `front()`.,front}

See also

back() – access the last element

Since

version 1.0.0

6.9.4.37 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> const_reference nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::front ( ) const [inline]

access the first element

Returns a reference to the first element in the container. For a JSON container `c`, the expression `c.front()` is equivalent to `*c.begin()`.

Returns

In case of a structured type (array or object), a reference to the first element is returned. In case of number, string, or boolean values, a reference to the value is returned.

Constant.

Precondition

The JSON value must not be `null` (would throw `std::out_of_range`) or an empty array or object (undefined behavior, **guarded by assertions**).

Postcondition

The JSON value remains unchanged.

Exceptions

invalid_iterator.214	when called on `null` value

{The following code shows an example for `front()`.,front}

See also

back() – access the last element

Since

version 1.0.0

6.9.4.38 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename BasicJsonType , detail::enable_if_t< std::is_same< typename std::remove_const< BasicJsonType >::type, basic_json_t >::value, int > = 0> basic_json nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::get ( ) const [inline]

get special-case overload

This overloads avoids a lot of template boilerplate, it can be seen as the identity method

Template Parameters

BasicJsonType	== basic_json

Returns

a copy of *this

Constant.

Since

version 2.1.0

6.9.4.39 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename ValueTypeCV , typename ValueType = detail::uncvref_t<ValueTypeCV>, detail::enable_if_t< not std::is_same< basic_json_t, ValueType >::value anddetail::has_from_json< basic_json_t, ValueType >::value andnot detail::has_non_default_from_json< basic_json_t, ValueType >::value, int > = 0> ValueType nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::get ( ) const [inline], [noexcept]

get a value (explicit)

Explicit type conversion between the JSON value and a compatible value which is `CopyConstructible` and `DefaultConstructible`. The value is converted by calling the json_serializer<ValueType> `from_json()` method.

The function is equivalent to executing

```
ValueType ret;
JSONSerializer<ValueType>::from_json(*this, ret);
return ret;
```

This overloads is chosen if:

- *ValueType* is not basic_json,
- json_serializer<ValueType> has a `from_json()` method of the form `void from_json(const basic_json&, ValueType&)`, and
- json_serializer<ValueType> does not have a `from_json()` method of the form `ValueType from_json(const basic_json&)`

Template Parameters

ValueTypeCV	the provided value type
ValueType	the returned value type

Returns

copy of the JSON value, converted to *ValueType*

Exceptions

what	json_serializer<ValueType> `from_json()` method throws

{The example below shows several conversions from JSON values to other types. There a few things to note: (1) Floating-point numbers can be converted to integers\, (2) A JSON array can be converted to a standard `std::vector<short>`\, (3) A JSON object can be converted to C++ associative containers such as `std::unordered_map<std::string\, json>`.,get__ValueType_const}

Since

version 2.1.0

6.9.4.40 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename ValueTypeCV , typename ValueType = detail::uncvref_t<ValueTypeCV>, detail::enable_if_t< not std::is_same< basic_json_t, ValueType >::value anddetail::has_non_default_from_json< basic_json_t, ValueType >::value, int > = 0> ValueType nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::get ( ) const [inline], [noexcept]

get a value (explicit); special case

Explicit type conversion between the JSON value and a compatible value which is **not** `CopyConstructible` and **not** `DefaultConstructible`. The value is converted by calling the json_serializer<ValueType> `from_json()` method.

The function is equivalent to executing

```
return JSONSerializer<ValueTypeCV>::from_json(*this);
```

This overloads is chosen if:

- *ValueType* is not basic_json and
- json_serializer<ValueType> has a `from_json()` method of the form `ValueType from_json(const basic_json&)`

Note

If json_serializer<ValueType> has both overloads of `from_json()`, this one is chosen.

Template Parameters

ValueTypeCV	the provided value type
ValueType	the returned value type

Returns

copy of the JSON value, converted to *ValueType*

Exceptions

what	json_serializer<ValueType> `from_json()` method throws

Since

version 2.1.0

6.9.4.41 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename PointerType , typename std::enable_if< std::is_pointer< PointerType >::value, int >::type = 0> PointerType nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::get ( ) `[inline]`, `[noexcept]`

get a pointer value (explicit)

Explicit pointer access to the internally stored JSON value. No copies are made.

Warning

The pointer becomes invalid if the underlying JSON object changes.

Template Parameters

PointerType	pointer type; must be a pointer to array_t, object_t, string_t, boolean_t, number_integer_t, number_unsigned_t, or number_float_t.

Returns

pointer to the internally stored JSON value if the requested pointer type *PointerType* fits to the JSON value; `nullptr` otherwise

Constant.

{The example below shows how pointers to internal values of a JSON value can be requested. Note that no type conversions are made and a `nullptr` is returned if the value and the requested pointer type does not match.,get↩__PointerType}

See also

get_ptr() for explicit pointer-member access

Since

version 1.0.0

6.9.4.42 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename PointerType , typename std::enable_if< std::is_pointer< PointerType >::value, int >::type = 0> constexpr const PointerType nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::get ( ) const [inline], [noexcept]

get a pointer value (explicit)

get a pointer value (explicit) Explicit pointer access to the internally stored JSON value. No copies are made.

Warning

The pointer becomes invalid if the underlying JSON object changes.

Template Parameters

PointerType	pointer type; must be a pointer to array_t, object_t, string_t, boolean_t, number_integer_t, number_unsigned_t, or number_float_t.

Returns

pointer to the internally stored JSON value if the requested pointer type *PointerType* fits to the JSON value; `nullptr` otherwise

Constant.

{The example below shows how pointers to internal values of a JSON value can be requested. Note that no type conversions are made and a `nullptr` is returned if the value and the requested pointer type does not match.,get__PointerType}

See also

get_ptr() for explicit pointer-member access

Since

version 1.0.0

6.9.4.43 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename PointerType , typename std::enable_if< std::is_pointer< PointerType >::value, int >::type = 0> PointerType nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::get_ptr ( ) [inline], [noexcept]

get a pointer value (implicit)

Implicit pointer access to the internally stored JSON value. No copies are made.

Warning

Writing data to the pointee of the result yields an undefined state.

Template Parameters

PointerType	pointer type; must be a pointer to array_t, object_t, string_t, boolean_t, number_integer_t, number_unsigned_t, or number_float_t. Enforced by a static assertion.

Returns

pointer to the internally stored JSON value if the requested pointer type *PointerType* fits to the JSON value; `nullptr` otherwise

Constant.

{The example below shows how pointers to internal values of a JSON value can be requested. Note that no type conversions are made and a `nullptr` is returned if the value and the requested pointer type does not match.,get↩ _ptr}

Since

version 1.0.0

6.9.4.44 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename PointerType , typename std::enable_if< std::is_pointer< PointerType >::value andstd::is_const< typename std::remove_pointer< PointerType >::type >::value, int >::type = 0> constexpr const PointerType nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::get_ptr ( ) const [inline], [noexcept]

get a pointer value (implicit)

get a pointer value (implicit) Implicit pointer access to the internally stored JSON value. No copies are made.

Warning

Writing data to the pointee of the result yields an undefined state.

Template Parameters

PointerType	pointer type; must be a pointer to array_t, object_t, string_t, boolean_t, number_integer_t, number_unsigned_t, or number_float_t. Enforced by a static assertion.

Returns

pointer to the internally stored JSON value if the requested pointer type *PointerType* fits to the JSON value; `nullptr` otherwise

Constant.

{The example below shows how pointers to internal values of a JSON value can be requested. Note that no type conversions are made and a `nullptr` is returned if the value and the requested pointer type does not match.,get_ptr}

Since

version 1.0.0

6.9.4.45 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename ReferenceType , typename std::enable_if< std::is_reference< ReferenceType >::value, int >::type = 0> ReferenceType nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::get_ref ( ) [inline]

get a reference value (implicit)

Implicit reference access to the internally stored JSON value. No copies are made.

Warning

Writing data to the referee of the result yields an undefined state.

Template Parameters

ReferenceType	reference type; must be a reference to array_t, object_t, string_t, boolean_t, number_integer_t, or number_float_t. Enforced by static assertion.

Returns

reference to the internally stored JSON value if the requested reference type *ReferenceType* fits to the JSON value; throws type_error.303 otherwise

Exceptions

type_error.303	in case passed type *ReferenceType* is incompatible with the stored JSON value; see example below

Constant.

{The example shows several calls to `get_ref()`.,get_ref}

Since

version 1.1.0

6.9.4.46 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename ReferenceType , typename std::enable_if< std::is_reference< ReferenceType >::value andstd::is_const< typename std::remove_reference< ReferenceType >::type >::value, int >::type = 0> ReferenceType nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::get_ref ( ) const [inline]

get a reference value (implicit)

get a reference value (implicit) Implicit reference access to the internally stored JSON value. No copies are made.

Warning

Writing data to the referee of the result yields an undefined state.

Template Parameters

ReferenceType	reference type; must be a reference to array_t, object_t, string_t, boolean_t, number_integer_t, or number_float_t. Enforced by static assertion.

Returns

reference to the internally stored JSON value if the requested reference type *ReferenceType* fits to the JSON value; throws type_error.303 otherwise

Exceptions

type_error.303	in case passed type *ReferenceType* is incompatible with the stored JSON value; see example below

Constant.

{The example shows several calls to `get_ref()`.,get_ref}

Since

version 1.1.0

6.9.4.47 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> iterator nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::insert (const_iterator *pos*, const basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer > & *val*) [inline]

inserts element

Inserts element *val* before iterator *pos*.

Parameters

in	*pos*	iterator before which the content will be inserted; may be the end() iterator
in	*val*	element to insert

Returns

iterator pointing to the inserted *val*.

Exceptions

type_error.309	if called on JSON values other than arrays; example: "cannot use insert() with string"
invalid_iterator.202	if *pos* is not an iterator of *this; example: "iterator does not fit current value"

Constant plus linear in the distance between *pos* and end of the container.

{The example shows how insert() is used.,insert}

Since

version 1.0.0

6.9.4.48 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> iterator nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::insert (const_iterator *pos*, basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer > && *val*) [inline]

inserts element

inserts element Inserts element *val* before iterator *pos*.

Parameters

in	*pos*	iterator before which the content will be inserted; may be the end() iterator
in	*val*	element to insert

Returns

iterator pointing to the inserted *val*.

Exceptions

type_error.309	if called on JSON values other than arrays; example: "`cannot use insert() with string`"
invalid_iterator.202	if *pos* is not an iterator of ∗this; example: "`iterator does not fit current value`"

Constant plus linear in the distance between *pos* and end of the container.

{The example shows how `insert()` is used.,insert}

Since

version 1.0.0

6.9.4.49 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> iterator nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::insert (const_iterator *pos*, size_type *cnt*, const basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer > & *val*) [inline]

inserts elements

Inserts *cnt* copies of *val* before iterator *pos*.

Parameters

in	*pos*	iterator before which the content will be inserted; may be the end() iterator
in	*cnt*	number of copies of *val* to insert
in	*val*	element to insert

Returns

iterator pointing to the first element inserted, or *pos* if `cnt==0`

Exceptions

type_error.309	if called on JSON values other than arrays; example: "`cannot use insert() with string`"
invalid_iterator.202	if *pos* is not an iterator of ∗this; example: "`iterator does not fit current value`"

Linear in *cnt* plus linear in the distance between *pos* and end of the container.

{The example shows how `insert()` is used.,insert__count}

Since

version 1.0.0

6.9.4.50 template< template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> iterator nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::insert (const_iterator *pos*, const_iterator *first*, const_iterator *last*) [inline]

inserts elements

Inserts elements from range `[first, last)` before iterator *pos*.

Parameters

`in`	*pos*	iterator before which the content will be inserted; may be the end() iterator
`in`	*first*	begin of the range of elements to insert
`in`	*last*	end of the range of elements to insert

Exceptions

type_error.309	if called on JSON values other than arrays; example: "`cannot use insert() with string`"
invalid_iterator.202	if *pos* is not an iterator of ∗this; example: "`iterator does not fit current value`"
invalid_iterator.210	if *first* and *last* do not belong to the same JSON value; example: "`iterators do not fit`"
invalid_iterator.211	if *first* or *last* are iterators into container for which insert is called; example: "`passed iterators may not belong to container`"

Returns

iterator pointing to the first element inserted, or *pos* if `first==last`

Linear in `std::distance(first, last)` plus linear in the distance between *pos* and end of the container.

{The example shows how `insert()` is used.,insert__range}

Since

version 1.0.0

6.9.4.51 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> iterator nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::insert (const_iterator *pos*, initializer_list_t *ilist*) [inline]

inserts elements

Inserts elements from initializer list *ilist* before iterator *pos*.

Parameters

in	*pos*	iterator before which the content will be inserted; may be the end() iterator
in	*ilist*	initializer list to insert the values from

Exceptions

type_error.309	if called on JSON values other than arrays; example: "`cannot use insert() with string`"
invalid_iterator.202	if *pos* is not an iterator of *this; example: "`iterator does not fit current value`"

Returns

iterator pointing to the first element inserted, or *pos* if `ilist` is empty

Linear in `ilist.size()` plus linear in the distance between *pos* and end of the container.

{The example shows how `insert()` is used.,insert__ilist}

Since

version 1.0.0

6.9.4.52 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> void nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::insert (const_iterator *first*, const_iterator *last*) [inline]

inserts elements

Inserts elements from range `[first, last)`.

Parameters

in	*first*	begin of the range of elements to insert
in	*last*	end of the range of elements to insert

Exceptions

type_error.309	if called on JSON values other than objects; example: "`cannot use insert() with string`"
invalid_iterator.202	if iterator *first* or *last* does does not point to an object; example: "`iterators first and last must point to objects`"
invalid_iterator.210	if *first* and *last* do not belong to the same JSON value; example: "`iterators do not fit`"

Logarithmic: `O(N*log(size() + N))`, where `N` is the number of elements to insert.

{The example shows how `insert()` is used.,insert__range_object}

Since

version 3.0.0

6.9.4.53 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> constexpr bool nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::is_array ( ) const [inline], [noexcept]

return whether value is an array

This function returns true if and only if the JSON value is an array.

Returns

`true` if type is array, `false` otherwise.

Constant.

No-throw guarantee: this member function never throws exceptions.

{The following code exemplifies `is_array()` for all JSON types.,is_array}

Since

version 1.0.0

6.9.4.54 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> constexpr bool nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::is_boolean ( ) const [inline], [noexcept]

return whether value is a boolean

This function returns true if and only if the JSON value is a boolean.

Returns

`true` if type is boolean, `false` otherwise.

Constant.

No-throw guarantee: this member function never throws exceptions.

{The following code exemplifies `is_boolean()` for all JSON types.,is_boolean}

Since

version 1.0.0

6.9.4.55 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> constexpr bool nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::is_discarded ( ) const [inline], [noexcept]

return whether value is discarded

This function returns true if and only if the JSON value was discarded during parsing with a callback function (see parser_callback_t).

Note

This function will always be `false` for JSON values after parsing. That is, discarded values can only occur during parsing, but will be removed when inside a structured value or replaced by null in other cases.

Returns

`true` if type is discarded, `false` otherwise.

Constant.

No-throw guarantee: this member function never throws exceptions.

{The following code exemplifies `is_discarded()` for all JSON types.,is_discarded}

Since

version 1.0.0

6.9.4.56 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> constexpr bool nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::is_null ( ) const [inline], [noexcept]

return whether value is null

This function returns true if and only if the JSON value is null.

Returns

`true` if type is null, `false` otherwise.

Constant.

No-throw guarantee: this member function never throws exceptions.

{The following code exemplifies `is_null()` for all JSON types.,is_null}

Since

version 1.0.0

6.9.4.57 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> constexpr bool nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::is_number ( ) const [inline], [noexcept]

return whether value is a number

This function returns true if and only if the JSON value is a number. This includes both integer (signed and unsigned) and floating-point values.

Returns

`true` if type is number (regardless whether integer, unsigned integer or floating-type), `false` otherwise.

Constant.

No-throw guarantee: this member function never throws exceptions.

{The following code exemplifies `is_number()` for all JSON types.,is_number}

See also

is_number_integer() – check if value is an integer or unsigned integer number
is_number_unsigned() – check if value is an unsigned integer number
is_number_float() – check if value is a floating-point number

Since

version 1.0.0

6.9.4.58 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> constexpr bool nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::is_number_float ( ) const [inline], [noexcept]

return whether value is a floating-point number

This function returns true if and only if the JSON value is a floating-point number. This excludes signed and unsigned integer values.

Returns

`true` if type is a floating-point number, `false` otherwise.

Constant.

No-throw guarantee: this member function never throws exceptions.

{The following code exemplifies `is_number_float()` for all JSON types.,is_number_float}

See also

is_number() – check if value is number
is_number_integer() – check if value is an integer number
is_number_unsigned() – check if value is an unsigned integer number

Since

version 1.0.0

6.9.4.59 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> constexpr bool nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::is_number_integer ( ) const [inline], [noexcept]

return whether value is an integer number

This function returns true if and only if the JSON value is a signed or unsigned integer number. This excludes floating-point values.

Returns

`true` if type is an integer or unsigned integer number, `false` otherwise.

Constant.

No-throw guarantee: this member function never throws exceptions.

{The following code exemplifies `is_number_integer()` for all JSON types.,is_number_integer}

See also

is_number() – check if value is a number
is_number_unsigned() – check if value is an unsigned integer number
is_number_float() – check if value is a floating-point number

Since

version 1.0.0

6.9.4.60 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> constexpr bool nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::is_number_unsigned ( ) const [inline], [noexcept]

return whether value is an unsigned integer number

This function returns true if and only if the JSON value is an unsigned integer number. This excludes floating-point and signed integer values.

Returns

`true` if type is an unsigned integer number, `false` otherwise.

Constant.

No-throw guarantee: this member function never throws exceptions.

{The following code exemplifies `is_number_unsigned()` for all JSON types.,is_number_unsigned}

See also

is_number() – check if value is a number
is_number_integer() – check if value is an integer or unsigned integer number
is_number_float() – check if value is a floating-point number

Since

version 2.0.0

6.9.4.61 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> constexpr bool nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::is_object ( ) const [inline], [noexcept]

return whether value is an object

This function returns true if and only if the JSON value is an object.

Returns

`true` if type is object, `false` otherwise.

Constant.

No-throw guarantee: this member function never throws exceptions.

{The following code exemplifies `is_object()` for all JSON types.,is_object}

Since

version 1.0.0

6.9.4.62 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> constexpr bool nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::is_primitive ( ) const [inline], [noexcept]

return whether type is primitive

This function returns true if and only if the JSON type is primitive (string, number, boolean, or null).

Returns

`true` if type is primitive (string, number, boolean, or null), `false` otherwise.

Constant.

No-throw guarantee: this member function never throws exceptions.

{The following code exemplifies `is_primitive()` for all JSON types.,is_primitive}

See also

is_structured() – returns whether JSON value is structured
is_null() – returns whether JSON value is `null`
is_string() – returns whether JSON value is a string
is_boolean() – returns whether JSON value is a boolean
is_number() – returns whether JSON value is a number

Since

version 1.0.0

6.9.4.63 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> constexpr bool nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::is_string ( ) const [inline], [noexcept]

return whether value is a string

This function returns true if and only if the JSON value is a string.

Returns

`true` if type is string, `false` otherwise.

Constant.

No-throw guarantee: this member function never throws exceptions.

{The following code exemplifies `is_string()` for all JSON types.,is_string}

Since

version 1.0.0

6.9.4.64 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> constexpr bool nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::is_structured ( ) const [inline], [noexcept]

return whether type is structured

This function returns true if and only if the JSON type is structured (array or object).

Returns

`true` if type is structured (array or object), `false` otherwise.

Constant.

No-throw guarantee: this member function never throws exceptions.

{The following code exemplifies `is_structured()` for all JSON types.,is_structured}

See also

is_primitive() – returns whether value is primitive
is_array() – returns whether value is an array
is_object() – returns whether value is an object

Since

version 1.0.0

6.9.4.65 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> static iteration_proxy<iterator> nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::iterator_wrapper (reference *cont*) [inline], [static]

wrapper to access iterator member functions in range-based for

This function allows to access iterator::key() and iterator::value() during range-based for loops. In these loops, a reference to the JSON values is returned, so there is no access to the underlying iterator.

{The following code shows how the wrapper is used,iterator_wrapper}

Note

The name of this function is not yet final and may change in the future.

6.9.4.66 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> static iteration_proxy<const_iterator> nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::iterator_wrapper (const_reference *cont*) [inline], [static]

wrapper to access iterator member functions in range-based for

This function allows to access iterator::key() and iterator::value() during range-based for loops. In these loops, a reference to the JSON values is returned, so there is no access to the underlying iterator.

{The following code shows how the wrapper is used,iterator_wrapper}

Note

The name of this function is not yet final and may change in the future.

6.9.4.67 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> size_type nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::max_size ( ) const [inline], [noexcept]

returns the maximum possible number of elements

Returns the maximum number of elements a JSON value is able to hold due to system or library implementation limitations, i.e. `std::distance(begin(), end())` for the JSON value.

Returns

The return value depends on the different types and is defined as follows:

Value type	return value
null	0 (same as `size()`)
boolean	1 (same as `size()`)
string	1 (same as `size()`)
number	1 (same as `size()`)
object	result of function `object_t::max_size()`
array	result of function `array_t::max_size()`

{The following code calls `max_size()` on the different value types. Note the output is implementation specific.,max_size}

Constant, as long as array_t and object_t satisfy the Container concept; that is, their `max_size()` functions have constant complexity.

No changes.

No-throw guarantee: this function never throws exceptions.

This function helps `basic_json` satisfying the `Container` requirements:

- The complexity is constant.
- Has the semantics of returning `b.size()` where `b` is the largest possible JSON value.

See also

size() – returns the number of elements

Since

version 1.0.0

6.9.4.68 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> static basic_json nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::meta ( ) [inline], [static]

returns version information on the library

This function returns a JSON object with information about the library, including the version number and information on the platform and compiler.

Returns

JSON object holding version information

key	description
`compiler`	Information on the used compiler. It is an object with the following keys: `c++` (the used C++ standard), `family` (the compiler family; possible values are `clang`, `icc`, `gcc`, `ilecpp`, `msvc`, `pgcpp`, `sunpro`, and `unknown`), and `version` (the compiler version).
`copyright`	The copyright line for the library as string.
`name`	The name of the library as string.
`platform`	The used platform as string. Possible values are `win32`, `linux`, `apple`, `unix`, and `unknown`.
`url`	The URL of the project as string.
`version`	The version of the library. It is an object with the following keys: `major`, `minor`, and `patch` as defined by `Semantic Versioning`, and `string` (the version string).

{The following code shows an example output of the `meta()` function.,meta}

Strong guarantee: if an exception is thrown, there are no changes to any JSON value.

Constant.

Since

2.1.0

6.9.4.69 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> static basic_json nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::object (initializer_list_t *init* = {}) [inline], [static]

explicitly create an object from an initializer list

Creates a JSON object value from a given initializer list. The initializer lists elements must be pairs, and their first elements must be strings. If the initializer list is empty, the empty object {} is created.

Note

This function is only added for symmetry reasons. In contrast to the related function array(initializer_list_t), there are no cases which can only be expressed by this function. That is, any initializer list *init* can also be passed to the initializer list constructor basic_json(initializer_list_t, bool, value_t).

Parameters

in	*init*	initializer list to create an object from (optional)

Returns

JSON object value

Exceptions

type_error.301	if *init* is not a list of pairs whose first elements are strings. In this case, no object can be created. When such a value is passed to basic_json(initializer_list_t, bool, value_t), an array would have been created from the passed initializer list *init*. See example below.

Linear in the size of *init*.

Strong guarantee: if an exception is thrown, there are no changes to any JSON value.

{The following code shows an example for the object function.,object}

See also

basic_json(initializer_list_t, bool, value_t) – create a JSON value from an initializer list
array(initializer_list_t) – create a JSON array value from an initializer list

Since

version 1.0.0

6.9.4.70 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> constexpr nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::operator value_t ( ) const [inline], [noexcept]

return the type of the JSON value (implicit)

Implicitly return the type of the JSON value as a value from the value_t enumeration.

Returns

the type of the JSON value

Constant.

No-throw guarantee: this member function never throws exceptions.

{The following code exemplifies the value_t operator for all JSON types.,operator__value_t}

See also

type() – return the type of the JSON value (explicit)
type_name() – return the type as string

Since

version 1.0.0

6.9.4.71 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename ValueType , typename std::enable_if< not std::is_pointer< ValueType >::value andnot std::is_same< ValueType, detail::json_ref< basic_json >>::value andnot std::is_same< ValueType, typename string_t::value_type >::valueand not std::is_same< ValueType, std::initializer_list< typename string_t::value_type >>::value, int >::type = 0> nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::operator ValueType ( ) const [inline]

get a value (implicit)

Implicit type conversion between the JSON value and a compatible value. The call is realized by calling get() const.

Template Parameters

ValueType	non-pointer type compatible to the JSON value, for instance `int` for JSON integer numbers, `bool` for JSON booleans, or `std::vector` types for JSON arrays. The character type of string_t as well as an initializer list of this type is excluded to avoid ambiguities as these types implicitly convert to `std::string`.

Returns

copy of the JSON value, converted to type *ValueType*

Exceptions

type_error.302	in case passed type *ValueType* is incompatible to the JSON value type (e.g., the JSON value is of type boolean, but a string is requested); see example below

Linear in the size of the JSON value.

{The example below shows several conversions from JSON values to other types. There a few things to note↩ : (1) Floating-point numbers can be converted to integers\, (2) A JSON array can be converted to a standard `std::vector<short>`\, (3) A JSON object can be converted to C++ associative containers such as `std↩ ::unordered_map<std::string\, json>`.,operator__ValueType}

Since

version 1.0.0

6.9.4.72 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> reference nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::operator+= (basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer > && *val*) [inline]

add an object to an array

add an object to an array Appends the given element *val* to the end of the JSON value. If the function is called on a JSON null value, an empty array is created before appending *val*.

Parameters

in	*val*	the value to add to the JSON array

Exceptions

type_error.308	when called on a type other than JSON array or null; example: `"cannot use push_back() with number"`

Amortized constant.

{The example shows how `push_back()` and += can be used to add elements to a JSON array. Note how the `null` value was silently converted to a JSON array.,push_back}

Since

version 1.0.0

6.9.4.73 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> reference nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::operator+= (const basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer > & *val*) [inline]

add an object to an array

add an object to an array Appends the given element *val* to the end of the JSON value. If the function is called on a JSON null value, an empty array is created before appending *val*.

Parameters

in	*val*	the value to add to the JSON array

Exceptions

type_error.308	when called on a type other than JSON array or null; example: "cannot use push_back() with number"

Amortized constant.

{The example shows how push_back() and += can be used to add elements to a JSON array. Note how the null value was silently converted to a JSON array.,push_back}

Since

version 1.0.0

6.9.4.74 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> reference nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::operator+= (const typename object_t::value_type & *val*) [inline]

add an object to an object

add an object to an object Inserts the given element *val* to the JSON object. If the function is called on a JSON null value, an empty object is created before inserting *val*.

Parameters

in	*val*	the value to add to the JSON object

Exceptions

type_error.308	when called on a type other than JSON object or null; example: `"cannot use push_back() with number"`

Logarithmic in the size of the container, O(log(`size()`)).

{The example shows how `push_back()` and += can be used to add elements to a JSON object. Note how the `null` value was silently converted to a JSON object.,push_back__object_t__value}

Since

version 1.0.0

6.9.4.75 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> reference nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::operator+= (initializer_list_t *init*) [inline]

add an object to an object

add an object to an object This function allows to use `push_back` with an initializer list. In case

1. the current value is an object,
2. the initializer list *init* contains only two elements, and
3. the first element of *init* is a string,

init is converted into an object element and added using push_back(const typename object_t::value_type&). Otherwise, *init* is converted to a JSON value and added using push_back(basic_json&&).

Parameters

`in`	*init*	an initializer list

Linear in the size of the initializer list *init*.

Note

This function is required to resolve an ambiguous overload error, because pairs like `{"key", "value"}` can be both interpreted as `object_t::value_type` or `std::initializer_list<basic_json>`, see `https://github.com/nlohmann/json/issues/235` for more information.

{The example shows how initializer lists are treated as objects when possible.,push_back__initializer_list}

6.9.4.76 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> reference& nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::operator= (basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer > *other*) [inline], [noexcept]

copy assignment

Copy assignment operator. Copies a JSON value via the "copy and swap" strategy: It is expressed in terms of the copy constructor, destructor, and the `swap()` member function.

Parameters

in	*other*	value to copy from

Linear.

This function helps `basic_json` satisfying the `Container` requirements:

- The complexity is linear.

{The code below shows and example for the copy assignment. It creates a copy of value `a` which is then swapped with `b`. Finally\, the copy of `a` (which is the null value after the swap) is destroyed.,basic_json__copyassignment}

Since

version 1.0.0

6.9.4.77 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> reference nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::operator[] (size_type *idx*) [inline]

access specified array element

Returns a reference to the element at specified location *idx*.

Note

If *idx* is beyond the range of the array (i.e., `idx >= size()`), then the array is silently filled up with `null` values to make `idx` a valid reference to the last stored element.

Parameters

in	*idx*	index of the element to access

Returns

reference to the element at index *idx*

Exceptions

type_error.305	if the JSON value is not an array or null; in that cases, using the [] operator with an index makes no sense.

Constant if *idx* is in the range of the array. Otherwise linear in `idx - size()`.

{The example below shows how array elements can be read and written using `[]` operator. Note the addition of `null` values.,operatorarray__size_type}

Since

version 1.0.0

6.9.4.78 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> const_reference nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::operator[] (size_type *idx*) const [inline]

access specified array element

Returns a const reference to the element at specified location *idx*.

Parameters

`in`	*idx*	index of the element to access

Returns

const reference to the element at index *idx*

Exceptions

type_error.305	if the JSON value is not an array; in that cases, using the [] operator with an index makes no sense.

Constant.

{The example below shows how array elements can be read using the `[]` operator.,operatorarray__size_type_↩ const}

Since

version 1.0.0

6.9.4.79 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> reference nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::operator[ ] (const typename object_t::key_type & *key*) [inline]

access specified object element

Returns a reference to the element at with specified key *key*.

Note

If *key* is not found in the object, then it is silently added to the object and filled with a `null` value to make `key` a valid reference. In case the value was `null` before, it is converted to an object.

Parameters

`in`	*key*	key of the element to access

Returns

reference to the element at key *key*

Exceptions

type_error.305	if the JSON value is not an object or null; in that cases, using the [] operator with a key makes no sense.

Logarithmic in the size of the container.

{The example below shows how object elements can be read and written using the `[]` operator.,operatorarray__↩ key_type}

See also

at(const typename object_t::key_type&) for access by reference with range checking
value() for access by value with a default value

Since

version 1.0.0

6.9.4.80 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> const_reference nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::operator[ ] (const typename object_t::key_type & *key*) const [inline]

read-only access specified object element

Returns a const reference to the element at with specified key *key*. No bounds checking is performed.

Warning

If the element with key *key* does not exist, the behavior is undefined.

Parameters

in	*key*	key of the element to access

Returns

const reference to the element at key *key*

Precondition

The element with key *key* must exist. **This precondition is enforced with an assertion.**

Exceptions

type_error.305	if the JSON value is not an object; in that cases, using the [] operator with a key makes no sense.

Logarithmic in the size of the container.

{The example below shows how object elements can be read using the `[]` operator.,operatorarray__key_type_↩ const}

See also

at(const typename object_t::key_type&) for access by reference with range checking
value() for access by value with a default value

Since

version 1.0.0

6.9.4.81 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename T > reference nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::operator[ ] (T * *key*) [inline]

access specified object element

Returns a reference to the element at with specified key *key*.

Note

If *key* is not found in the object, then it is silently added to the object and filled with a `null` value to make `key` a valid reference. In case the value was `null` before, it is converted to an object.

Parameters

in	*key*	key of the element to access

Returns

reference to the element at key *key*

Exceptions

type_error.305	if the JSON value is not an object or null; in that cases, using the [] operator with a key makes no sense.

Logarithmic in the size of the container.

{The example below shows how object elements can be read and written using the `[]` operator.,operatorarray__↩ key_type}

See also

at(const typename object_t::key_type&) for access by reference with range checking
value() for access by value with a default value

Since

version 1.1.0

6.9.4.82 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename T > const_reference nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::operator[ ] (T * *key*) const [inline]

read-only access specified object element

Returns a const reference to the element at with specified key *key*. No bounds checking is performed.

Warning

If the element with key *key* does not exist, the behavior is undefined.

Parameters

in	*key*	key of the element to access

Returns

const reference to the element at key *key*

Precondition

The element with key *key* must exist. **This precondition is enforced with an assertion.**

Exceptions

type_error.305	if the JSON value is not an object; in that cases, using the [] operator with a key makes no sense.

Logarithmic in the size of the container.

{The example below shows how object elements can be read using the `[]` operator.,operatorarray__key_type_↩ const}

See also

at(const typename object_t::key_type&) for access by reference with range checking
value() for access by value with a default value

Since

version 1.1.0

6.9.4.83 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> reference nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::operator[] (const json_pointer & *ptr*) [inline]

access specified element via JSON Pointer

Uses a JSON pointer to retrieve a reference to the respective JSON value. No bound checking is performed. Similar to operator[](const typename object_t::key_type&), `null` values are created in arrays and objects if necessary.

In particular:

- If the JSON pointer points to an object key that does not exist, it is created an filled with a `null` value before a reference to it is returned.
- If the JSON pointer points to an array index that does not exist, it is created an filled with a `null` value before a reference to it is returned. All indices between the current maximum and the given index are also filled with `null`.
- The special value `-` is treated as a synonym for the index past the end.

Parameters

in	*ptr*	a JSON pointer

Returns

reference to the element pointed to by *ptr*

Constant.

Exceptions

parse_error.106	if an array index begins with '0'
parse_error.109	if an array index was not a number
out_of_range.404	if the JSON pointer can not be resolved

{The behavior is shown in the example.,operatorjson_pointer}

Since

version 2.0.0

6.9.4.84 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> const_reference nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::operator[ ] (const json_pointer & *ptr*) const [inline]

access specified element via JSON Pointer

Uses a JSON pointer to retrieve a reference to the respective JSON value. No bound checking is performed. The function does not change the JSON value; no `null` values are created. In particular, the the special value - yields an exception.

Parameters

in	*ptr*	JSON pointer to the desired element

Returns

const reference to the element pointed to by *ptr*

Constant.

Exceptions

parse_error.106	if an array index begins with '0'

Exceptions

parse_error.109	if an array index was not a number
out_of_range.402	if the array index '-' is used
out_of_range.404	if the JSON pointer can not be resolved

{The behavior is shown in the example.,operatorjson_pointer_const}

Since

version 2.0.0

6.9.4.85 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> static basic_json nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::parse (detail::input_adapter *i,* const parser_callback_t *cb* = `nullptr`, const bool *allow_exceptions* = `true`) `[inline]`, `[static]`

deserialize from a compatible input

This function reads from a compatible input. Examples are:

- an array of 1-byte values
- strings with character/literal type with size of 1 byte
- input streams
- container with contiguous storage of 1-byte values. Compatible container types include `std::vector`, `std::string`, `std::array`, `std::valarray`, and `std::initializer_list`. Furthermore, C-style arrays can be used with `std::begin()`/`std::end()`. User-defined containers can be used as long as they implement random-access iterators and a contiguous storage.

Precondition

Each element of the container has a size of 1 byte. Violating this precondition yields undefined behavior. **This precondition is enforced with a static assertion.**
The container storage is contiguous. Violating this precondition yields undefined behavior. **This precondition is enforced with an assertion.**
Each element of the container has a size of 1 byte. Violating this precondition yields undefined behavior. **This precondition is enforced with a static assertion.**

Warning

There is no way to enforce all preconditions at compile-time. If the function is called with a noncompliant container and with assertions switched off, the behavior is undefined and will most likely yield segmentation violation.

Parameters

in	*i*	input to read from
in	*cb*	a parser callback function of type parser_callback_t which is used to control the deserialization by filtering unwanted values (optional)

Returns

result of the deserialization

Exceptions

parse_error.101	if a parse error occurs; example: ""unexpected end of input; expected string literal""
parse_error.102	if to_unicode fails or surrogate error
parse_error.103	if to_unicode fails

Linear in the length of the input. The parser is a predictive LL(1) parser. The complexity can be higher if the parser callback function *cb* has a super-linear complexity.

Note

A UTF-8 byte order mark is silently ignored.

{The example below demonstrates the `parse()` function reading from an array.,parse__array__parser_callback_t}

{The example below demonstrates the `parse()` function with and without callback function.,parse__string__parser_callback_t}

{The example below demonstrates the `parse()` function with and without callback function.,parse__istream__parser_callback_t}

{The example below demonstrates the `parse()` function reading from a contiguous container.,parse__contiguouscontainer__parser_callback_t}

Since

version 2.0.3 (contiguous containers)

6.9.4.86 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> static basic_json nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::parse (detail::input_adapter & *i*, const parser_callback_t *cb* = `nullptr`, const bool *allow_exceptions* = `true`) `[inline]`, `[static]`

create an empty value with a given type parse(detail::input_adapter, const parser_callback_t)

Create an empty JSON value with a given type. The value will be default initialized with an empty value which depends on the type:

Value type	initial value
null	`null`
boolean	`false`
string	`""`
number	`0`
object	`{}`
array	`[]`

Parameters

`in`	*v*	the type of the value to create

Constant.

Strong guarantee: if an exception is thrown, there are no changes to any JSON value.

{The following code shows the constructor for different value_t values,basic_json__value_t}

See also

clear() – restores the postcondition of this constructor

Since

version 1.0.0 parse(detail::input_adapter, const parser_callback_t)

6.9.4.87 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<class IteratorType , typename std::enable_if< std::is_base_of< std::random_access_iterator_tag, typename std::iterator_traits< IteratorType >::iterator_category >::value, int >::type = 0> static basic_json nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::parse (IteratorType *first,* IteratorType *last,* const parser_callback_t *cb* = `nullptr`, const bool *allow_exceptions* = `true`) `[inline],[static]`

deserialize from an iterator range with contiguous storage

This function reads from an iterator range of a container with contiguous storage of 1-byte values. Compatible container types include `std::vector`, `std::string`, `std::array`, `std::valarray`, and `std::initializer_list`. Furthermore, C-style arrays can be used with `std::begin()`/`std::end()`. User-defined containers can be used as long as they implement random-access iterators and a contiguous storage.

Precondition

The iterator range is contiguous. Violating this precondition yields undefined behavior. **This precondition is enforced with an assertion.**
Each element in the range has a size of 1 byte. Violating this precondition yields undefined behavior. **This precondition is enforced with a static assertion.**

Warning

There is no way to enforce all preconditions at compile-time. If the function is called with noncompliant iterators and with assertions switched off, the behavior is undefined and will most likely yield segmentation violation.

Template Parameters

IteratorType	iterator of container with contiguous storage

Parameters

in	*first*	begin of the range to parse (included)
in	*last*	end of the range to parse (excluded)
in	*cb*	a parser callback function of type parser_callback_t which is used to control the deserialization by filtering unwanted values (optional)
in	*allow_exceptions*	whether to throw exceptions in case of a parse error (optional, true by default)

Returns

result of the deserialization

Exceptions

parse_error.101	in case of an unexpected token
parse_error.102	if to_unicode fails or surrogate error
parse_error.103	if to_unicode fails

Linear in the length of the input. The parser is a predictive LL(1) parser. The complexity can be higher if the parser callback function *cb* has a super-linear complexity.

Note

A UTF-8 byte order mark is silently ignored.

{The example below demonstrates the `parse()` function reading from an iterator range.,parse__iteratortype__↩ parser_callback_t}

Since

version 2.0.3

6.9.4.88 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> basic_json nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::patch (const basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer > & *json_patch*) const [inline]

applies a JSON patch

JSON Patch defines a JSON document structure for expressing a sequence of operations to apply to a JSON) document. With this function, a JSON Patch is applied to the current JSON value by executing all operations from the patch.

Parameters

in	*json_patch*	JSON patch document

Returns

patched document

Note

The application of a patch is atomic: Either all operations succeed and the patched document is returned or an exception is thrown. In any case, the original value is not changed: the patch is applied to a copy of the value.

Exceptions

parse_error.104	if the JSON patch does not consist of an array of objects
parse_error.105	if the JSON patch is malformed (e.g., mandatory attributes are missing); example: `"operation add must have member path"`
out_of_range.401	if an array index is out of range.
out_of_range.403	if a JSON pointer inside the patch could not be resolved successfully in the current JSON value; example: `"key baz not found"`
out_of_range.405	if JSON pointer has no parent ("add", "remove", "move")
other_error.501	if "test" operation was unsuccessful

Linear in the size of the JSON value and the length of the JSON patch. As usually only a fraction of the JSON value is affected by the patch, the complexity can usually be neglected.

{The following code shows how a JSON patch is applied to a value.,patch}

See also

diff – create a JSON patch by comparing two JSON values
RFC 6902 (JSON Patch)
RFC 6901 (JSON Pointer)

Since

version 2.0.0

6.9.4.89 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> void nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::push_back (basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer > && *val*) [inline]

add an object to an array

Appends the given element *val* to the end of the JSON value. If the function is called on a JSON null value, an empty array is created before appending *val*.

Parameters

in	*val*	the value to add to the JSON array

Exceptions

type_error.308	when called on a type other than JSON array or null; example: "`cannot use push_back() with number`"

Amortized constant.

{The example shows how `push_back()` and += can be used to add elements to a JSON array. Note how the `null` value was silently converted to a JSON array.,push_back}

Since

version 1.0.0

6.9.4.90 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> void nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::push_back (const basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer > & *val*) [inline]

add an object to an array

add an object to an array Appends the given element *val* to the end of the JSON value. If the function is called on a JSON null value, an empty array is created before appending *val*.

Parameters

in	*val*	the value to add to the JSON array

Exceptions

type_error.308	when called on a type other than JSON array or null; example: "`cannot use push_back() with number`"

Amortized constant.

{The example shows how `push_back()` and += can be used to add elements to a JSON array. Note how the `null` value was silently converted to a JSON array.,push_back}

Since

version 1.0.0

6.9.4.91 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> void nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::push_back (const typename object_t::value_type & *val*) [inline]

add an object to an object

Inserts the given element *val* to the JSON object. If the function is called on a JSON null value, an empty object is created before inserting *val*.

Parameters

in	*val*	the value to add to the JSON object

Exceptions

type_error.308	when called on a type other than JSON object or null; example: "cannot use push_back() with number"

Logarithmic in the size of the container, O(log(size())).

{The example shows how push_back() and += can be used to add elements to a JSON object. Note how the null value was silently converted to a JSON object.,push_back__object_t__value}

Since

version 1.0.0

6.9.4.92 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> void nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::push_back (initializer_list_t *init*) [inline]

add an object to an object

This function allows to use push_back with an initializer list. In case

1. the current value is an object,
2. the initializer list *init* contains only two elements, and
3. the first element of *init* is a string,

init is converted into an object element and added using push_back(const typename object_t::value_type&). Otherwise, *init* is converted to a JSON value and added using push_back(basic_json&&).

Parameters

in	*init*	an initializer list

Linear in the size of the initializer list *init*.

Note

This function is required to resolve an ambiguous overload error, because pairs like `{"key", "value"}` can be both interpreted as `object_t::value_type` or `std::initializer_list<basic_json>`, see https://github.com/nlohmann/json/issues/235 for more information.

{The example shows how initializer lists are treated as objects when possible.,push_back__initializer_list}

6.9.4.93 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> reverse_iterator nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::rbegin ( ) [inline], [noexcept]

returns an iterator to the reverse-beginning

Returns an iterator to the reverse-beginning; that is, the last element.

Constant.

This function helps `basic_json` satisfying the `ReversibleContainer` requirements:

- The complexity is constant.
- Has the semantics of `reverse_iterator(end())`.

{The following code shows an example for `rbegin()`.,rbegin}

See also

crbegin() – returns a const reverse iterator to the beginning
rend() – returns a reverse iterator to the end
crend() – returns a const reverse iterator to the end

Since

version 1.0.0

6.9.4.94 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> const_reverse_iterator nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::rbegin ( ) const [inline], [noexcept]

returns a const reverse iterator to the last element

Returns a const iterator to the reverse-beginning; that is, the last element.

Constant.

This function helps `basic_json` satisfying the `ReversibleContainer` requirements:

- The complexity is constant.
- Has the semantics of `const_cast<const basic_json&>(*this).rbegin()`.

{The following code shows an example for `crbegin()`.,crbegin}

See also

rbegin() – returns a reverse iterator to the beginning
rend() – returns a reverse iterator to the end
crend() – returns a const reverse iterator to the end

Since

version 1.0.0

6.9.4.95 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> reverse_iterator nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::rend ( ) [inline], [noexcept]

returns an iterator to the reverse-end

Returns an iterator to the reverse-end; that is, one before the first element.

Constant.

This function helps `basic_json` satisfying the `ReversibleContainer` requirements:

- The complexity is constant.
- Has the semantics of `reverse_iterator(begin())`.

{The following code shows an example for `rend()`.,rend}

See also

crend() – returns a const reverse iterator to the end
rbegin() – returns a reverse iterator to the beginning
crbegin() – returns a const reverse iterator to the beginning

Since

version 1.0.0

6.9.4.96 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> const_reverse_iterator nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::rend ( ) const [inline], [noexcept]

returns a const reverse iterator to one before the first

Returns a const reverse iterator to the reverse-end; that is, one before the first element.

Constant.

This function helps `basic_json` satisfying the `ReversibleContainer` requirements:

- The complexity is constant.
- Has the semantics of `const_cast<const basic_json&>(*this).rend()`.

{The following code shows an example for `crend()`.,crend}

See also

rend() – returns a reverse iterator to the end
rbegin() – returns a reverse iterator to the beginning
crbegin() – returns a const reverse iterator to the beginning

Since

version 1.0.0

6.9.4.97 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> size_type nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::size ( ) const [inline], [noexcept]

returns the number of elements

Returns the number of elements in a JSON value.

Returns

The return value depends on the different types and is defined as follows:

Value type	return value
null	0
boolean	1
string	1
number	1
object	result of function object_t::size()
array	result of function array_t::size()

{The following code calls `size()` on the different value types.,size}

Constant, as long as array_t and object_t satisfy the Container concept; that is, their size() functions have constant complexity.

No changes.

No-throw guarantee: this function never throws exceptions.

Note

This function does not return the length of a string stored as JSON value - it returns the number of elements in the JSON value which is 1 in the case of a string.

This function helps `basic_json` satisfying the `Container` requirements:

- The complexity is constant.
- Has the semantics of `std::distance(begin(), end())`.

See also

empty() – checks whether the container is empty
max_size() – returns the maximal number of elements

Since

version 1.0.0

6.9.4.98 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> void nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::swap (reference *other*) [inline], [noexcept]

exchanges the values

Exchanges the contents of the JSON value with those of *other*. Does not invoke any move, copy, or swap operations on individual elements. All iterators and references remain valid. The past-the-end iterator is invalidated.

Parameters

`in,out`	*other*	JSON value to exchange the contents with

Constant.

{The example below shows how JSON values can be swapped with `swap()`.,swap__reference}

Since

version 1.0.0

6.9.4.99 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> void nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::swap (array_t & *other*) [inline]

exchanges the values

Exchanges the contents of a JSON array with those of *other*. Does not invoke any move, copy, or swap operations on individual elements. All iterators and references remain valid. The past-the-end iterator is invalidated.

Parameters

in,out	*other*	array to exchange the contents with

Exceptions

type_error.310	when JSON value is not an array; example: "`cannot use swap() with string`"

Constant.

{The example below shows how arrays can be swapped with `swap()`.,swap__array_t}

Since

version 1.0.0

6.9.4.100 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> void nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::swap (object_t & *other*) [inline]

exchanges the values

Exchanges the contents of a JSON object with those of *other*. Does not invoke any move, copy, or swap operations on individual elements. All iterators and references remain valid. The past-the-end iterator is invalidated.

Parameters

in,out	*other*	object to exchange the contents with

Exceptions

type_error.310	when JSON value is not an object; example: "`cannot use swap() with string`"

Constant.

{The example below shows how objects can be swapped with `swap()`.,swap__object_t}

Since

version 1.0.0

6.9.4.101 **template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> void nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::swap (string_t & *other*)** `[inline]`

exchanges the values

Exchanges the contents of a JSON string with those of *other*. Does not invoke any move, copy, or swap operations on individual elements. All iterators and references remain valid. The past-the-end iterator is invalidated.

Parameters

`in,out`	*other*	string to exchange the contents with

Exceptions

type_error.310	when JSON value is not a string; example: "`cannot use swap() with boolean`"

Constant.

{The example below shows how strings can be swapped with `swap()`.,swap__string_t}

Since

version 1.0.0

6.9.4.102 **template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> static std::vector<uint8_t> nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::to_cbor (const basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer > & *j*)** `[inline]`, `[static]`

create a CBOR serialization of a given JSON value

Serializes a given JSON value *j* to a byte vector using the CBOR (Concise Binary Object Representation) serialization format. CBOR is a binary serialization format which aims to be more compact than JSON itself, yet more efficient to parse.

The library uses the following mapping from JSON values types to CBOR types according to the CBOR specification (RFC 7049):

JSON value type	value/range	CBOR type	first byte
null	`null`	Null	0xf6
boolean	`true`	True	0xf5
boolean	`false`	False	0xf4
number_integer	-9223372036854775808..-2147483649	Negative integer (8 bytes follow)	0x3b
number_integer	-2147483648..-32769	Negative integer (4 bytes follow)	0x3a
number_integer	-32768..-129	Negative integer (2 bytes follow)	0x39
number_integer	-128..-25	Negative integer (1 byte follow)	0x38
number_integer	-24..-1	Negative integer	0x20..0x37
number_integer	0..23	Integer	0x00..0x17
number_integer	24..255	Unsigned integer (1 byte follow)	0x18
number_integer	256..65535	Unsigned integer (2 bytes follow)	0x19
number_integer	65536..4294967295	Unsigned integer (4 bytes follow)	0x1a
number_integer	4294967296..18446744073709551615	Unsigned integer (8 bytes follow)	0x1b
number_unsigned	0..23	Integer	0x00..0x17
number_unsigned	24..255	Unsigned integer (1 byte follow)	0x18
number_unsigned	256..65535	Unsigned integer (2 bytes follow)	0x19
number_unsigned	65536..4294967295	Unsigned integer (4 bytes follow)	0x1a
number_unsigned	4294967296..18446744073709551615	Unsigned integer (8 bytes follow)	0x1b
number_float	*any value*	Double-Precision Float	0xfb
string	*length*: 0..23	UTF-8 string	0x60..0x77
string	*length*: 23..255	UTF-8 string (1 byte follow)	0x78
string	*length*: 256..65535	UTF-8 string (2 bytes follow)	0x79
string	*length*: 65536..4294967295	UTF-8 string (4 bytes follow)	0x7a
string	*length*↩ : 4294967296..18446744073709551615	UTF-8 string (8 bytes follow)	0x7b
array	*size*: 0..23	array	0x80..0x97
array	*size*: 23..255	array (1 byte follow)	0x98
array	*size*: 256..65535	array (2 bytes follow)	0x99
array	*size*: 65536..4294967295	array (4 bytes follow)	0x9a
array	*size*: 4294967296..18446744073709551615	array (8 bytes follow)	0x9b
object	*size*: 0..23	map	0xa0..0xb7
object	*size*: 23..255	map (1 byte follow)	0xb8
object	*size*: 256..65535	map (2 bytes follow)	0xb9
object	*size*: 65536..4294967295	map (4 bytes follow)	0xba
object	*size*: 4294967296..18446744073709551615	map (8 bytes follow)	0xbb

Note

The mapping is **complete** in the sense that any JSON value type can be converted to a CBOR value.
If NaN or Infinity are stored inside a JSON number, they are serialized properly. This behavior differs from the dump() function which serializes NaN or Infinity to `null`.
The following CBOR types are not used in the conversion:

- byte strings (0x40..0x5f)
- UTF-8 strings terminated by "break" (0x7f)
- arrays terminated by "break" (0x9f)
- maps terminated by "break" (0xbf)
- date/time (0xc0..0xc1)
- bignum (0xc2..0xc3)
- decimal fraction (0xc4)
- bigfloat (0xc5)
- tagged items (0xc6..0xd4, 0xd8..0xdb)
- expected conversions (0xd5..0xd7)
- simple values (0xe0..0xf3, 0xf8)
- undefined (0xf7)
- half and single-precision floats (0xf9-0xfa)
- break (0xff)

Parameters

in	*j*	JSON value to serialize

Returns

MessagePack serialization as byte vector

Linear in the size of the JSON value *j*.

{The example shows the serialization of a JSON value to a byte vector in CBOR format.,to_cbor}

See also

http://cbor.io
from_cbor(const std::vector<uint8_t>&, const size_t) for the analogous deserialization
to_msgpack(const basic_json&) for the related MessagePack format

Since

version 2.0.9

6.9.4.103 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> static std::vector<uint8_t> nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::to_msgpack (const basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer > & *j*) [inline], [static]

create a MessagePack serialization of a given JSON value

Serializes a given JSON value *j* to a byte vector using the MessagePack serialization format. MessagePack is a binary serialization format which aims to be more compact than JSON itself, yet more efficient to parse.

The library uses the following mapping from JSON values types to MessagePack types according to the Message↩ Pack specification:

JSON value type	value/range	MessagePack type	first byte
null	`null`	nil	0xc0
boolean	`true`	true	0xc3
boolean	`false`	false	0xc2
number_integer	-9223372036854775808..-2147483649	int64	0xd3
number_integer	-2147483648..-32769	int32	0xd2
number_integer	-32768..-129	int16	0xd1
number_integer	-128..-33	int8	0xd0
number_integer	-32..-1	negative fixint	0xe0..0xff
number_integer	0..127	positive fixint	0x00..0x7f
number_integer	128..255	uint 8	0xcc
number_integer	256..65535	uint 16	0xcd
number_integer	65536..4294967295	uint 32	0xce
number_integer	4294967296..18446744073709551615	uint 64	0xcf
number_unsigned	0..127	positive fixint	0x00..0x7f
number_unsigned	128..255	uint 8	0xcc
number_unsigned	256..65535	uint 16	0xcd
number_unsigned	65536..4294967295	uint 32	0xce
number_unsigned	4294967296..18446744073709551615	uint 64	0xcf
number_float	*any value*	float 64	0xcb
string	*length*: 0..31	fixstr	0xa0..0xbf
string	*length*: 32..255	str 8	0xd9
string	*length*: 256..65535	str 16	0xda
string	*length*: 65536..4294967295	str 32	0xdb
array	*size*: 0..15	fixarray	0x90..0x9f
array	*size*: 16..65535	array 16	0xdc
array	*size*: 65536..4294967295	array 32	0xdd
object	*size*: 0..15	fix map	0x80..0x8f
object	*size*: 16..65535	map 16	0xde
object	*size*: 65536..4294967295	map 32	0xdf

Note

The mapping is **complete** in the sense that any JSON value type can be converted to a MessagePack value.
The following values can **not** be converted to a MessagePack value:

- strings with more than 4294967295 bytes
- arrays with more than 4294967295 elements
- objects with more than 4294967295 elements

The following MessagePack types are not used in the conversion:

- bin 8 - bin 32 (0xc4..0xc6)
- ext 8 - ext 32 (0xc7..0xc9)
- float 32 (0xca)
- fixext 1 - fixext 16 (0xd4..0xd8)

Any MessagePack output created to_msgpack can be successfully parsed by from_msgpack.
If NaN or Infinity are stored inside a JSON number, they are serialized properly. This behavior differs from the dump() function which serializes NaN or Infinity to `null`.

Parameters

in	*j*	JSON value to serialize

Returns

MessagePack serialization as byte vector

Linear in the size of the JSON value *j*.

{The example shows the serialization of a JSON value to a byte vector in MessagePack format.,to_msgpack}

See also

http://msgpack.org
from_msgpack(const std::vector<uint8_t>&, const size_t) for the analogous deserialization
to_cbor(const basic_json& for the related CBOR format

Since

version 2.0.9

6.9.4.104 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> constexpr value_t nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::type ( ) const [inline], [noexcept]

return the type of the JSON value (explicit)

Return the type of the JSON value as a value from the value_t enumeration.

Returns

the type of the JSON value

Value type	return value
null	value_t::null
boolean	value_t::boolean
string	value_t::string
number (integer)	value_t::number_integer
number (unsigned integer)	value_t::number_unsigned
number (floating-point)	value_t::number_float
object	value_t::object
array	value_t::array
discarded	value_t::discarded

Constant.

No-throw guarantee: this member function never throws exceptions.

{The following code exemplifies `type()` for all JSON types.,type}

See also

operator value_t() – return the type of the JSON value (implicit)
type_name() – return the type as string

Since

version 1.0.0

6.9.4.105 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> const char* nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::type_name ( ) const [inline], [noexcept]

return the type as string

Returns the type name as string to be used in error messages - usually to indicate that a function was called on a wrong JSON type.

Returns

a string representation of a the *m_type* member:

Value type	return value
null	`"null"`
boolean	`"boolean"`
string	`"string"`
number	`"number"` (for all number types)
object	`"object"`
array	`"array"`
discarded	`"discarded"`

No-throw guarantee: this function never throws exceptions.

Constant.

{The following code exemplifies `type_name()` for all JSON types.,type_name}

See also

type() – return the type of the JSON value
operator value_t() – return the type of the JSON value (implicit)

Since

version 1.0.0, public since 2.1.0, `const char*` and `noexcept` since 3.0.0

6.9.4.106 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> basic_json nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::unflatten ( ) const [inline]

unflatten a previously flattened JSON value

The function restores the arbitrary nesting of a JSON value that has been flattened before using the flatten() function. The JSON value must meet certain constraints:

1. The value must be an object.
2. The keys must be JSON pointers (see RFC 6901)
3. The mapped values must be primitive JSON types.

Returns

the original JSON from a flattened version

Note

Empty objects and arrays are flattened by flatten() to `null` values and can not unflattened to their original type. Apart from this example, for a JSON value `j`, the following is always true: `j == j.flatten().unflatten()`.

Linear in the size the JSON value.

Exceptions

type_error.314	if value is not an object
type_error.315	if object values are not primitive

{The following code shows how a flattened JSON object is unflattened into the original nested JSON object.,unflatten}

See also

flatten() for the reverse function

Since

version 2.0.0

6.9.4.107 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> void nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::update (const_reference *j*) [inline]

updates a JSON object from another object, overwriting existing keys

Inserts all values from JSON object *j* and overwrites existing keys.

Parameters

in	*j*	JSON object to read values from

Exceptions

type_error.312	if called on JSON values other than objects; example: "`cannot use update() with string`"

O(N∗log(size() + N)), where N is the number of elements to insert.

{The example shows how `update()` is used.,update}

See also

https://docs.python.org/3.6/library/stdtypes.html#dict.update

Since

version 3.0.0

6.9.4.108 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> void nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::update (const_iterator *first*, const_iterator *last*) [inline]

updates a JSON object from another object, overwriting existing keys

Inserts all values from from range `[first, last)` and overwrites existing keys.

Parameters

in	*first*	begin of the range of elements to insert
in	*last*	end of the range of elements to insert

Exceptions

type_error.312	if called on JSON values other than objects; example: "`cannot use update() with string`"
invalid_iterator.202	if iterator *first* or *last* does does not point to an object; example: "`iterators first and last must point to objects`"
invalid_iterator.210	if *first* and *last* do not belong to the same JSON value; example: "`iterators do not fit`"

O(N∗log(size() + N)), where N is the number of elements to insert.

{The example shows how `update()` is used__range.,update}

See also

https://docs.python.org/3.6/library/stdtypes.html#dict.update

Since

version 3.0.0

6.9.4.109 **template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<class ValueType , typename std::enable_if< std::is_convertible< basic_json_t, ValueType >::value, int >::type = 0> ValueType nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::value (const typename object_t::key_type & *key,* const ValueType & *default_value*) const** `[inline]`

access specified object element with default value

Returns either a copy of an object's element at the specified key *key* or a given default value if no element with key *key* exists.

The function is basically equivalent to executing

```
try {
    return at(key);
} catch(out_of_range) {
    return default_value;
}
```

Note

Unlike at(const typename object_t::key_type&), this function does not throw if the given key *key* was not found. Unlike operator[ ](const typename object_t::key_type& key), this function does not implicitly add an element to the position defined by *key*. This function is furthermore also applicable to const objects.

Parameters

in	*key*	key of the element to access
in	*default_value*	the value to return if *key* is not found

Template Parameters

ValueType	type compatible to JSON values, for instance `int` for JSON integer numbers, `bool` for JSON booleans, or `std::vector` types for JSON arrays. Note the type of the expected value at *key* and the default value *default_value* must be compatible.

Returns

copy of the element at key *key* or *default_value* if *key* is not found

Exceptions

type_error.306	if the JSON value is not an objec; in that cases, using `value()` with a key makes no sense.

Logarithmic in the size of the container.

{The example below shows how object elements can be queried with a default value.,basic_json__value}

See also

at(const typename object_t::key_type&) for access by reference with range checking
operator[](const typename object_t::key_type&) for unchecked access by reference

Since

version 1.0.0

6.9.4.110 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> string_t nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::value (const typename object_t::key_type & *key,* const char * *default_value*) const [inline]

overload for a default value of type const char*

6.9.4.111 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<class ValueType , typename std::enable_if< std::is_convertible< basic_json_t, ValueType >::value, int >::type = 0> ValueType nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::value (const json_pointer & *ptr,* const ValueType & *default_value*) const [inline]

access specified object element via JSON Pointer with default value

Returns either a copy of an object's element at the specified key *key* or a given default value if no element with key *key* exists.

The function is basically equivalent to executing

```
try {
    return at(ptr);
} catch(out_of_range) {
    return default_value;
}
```

Note

Unlike at(const json_pointer&), this function does not throw if the given key *key* was not found.

Parameters

in	*ptr*	a JSON pointer to the element to access
in	*default_value*	the value to return if *ptr* found no value

Template Parameters

ValueType	type compatible to JSON values, for instance `int` for JSON integer numbers, `bool` for JSON booleans, or `std::vector` types for JSON arrays. Note the type of the expected value at *key* and the default value *default_value* must be compatible.

Returns

copy of the element at key *key* or *default_value* if *key* is not found

Exceptions

type_error.306	if the JSON value is not an objec; in that cases, using `value()` with a key makes no sense.

Logarithmic in the size of the container.

{The example below shows how object elements can be queried with a default value.,basic_json__value_ptr}

See also

operator[ ](const json_pointer&) for unchecked access by reference

Since

version 2.0.2

6.9.4.112 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> string_t nlohmann::basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer >::value (const json_pointer & *ptr,* const char * *default_value*) const [inline]

overload for a default value of type const char*

6.9.5 Friends And Related Function Documentation

6.9.5.1 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> bool operator!= (const_reference *lhs,* const_reference *rhs*) [friend]

comparison: not equal

Compares two JSON values for inequality by calculating `not (lhs == rhs)`.

Parameters

in	*lhs*	first JSON value to consider
in	*rhs*	second JSON value to consider

Returns

whether the values *lhs* and *rhs* are not equal

Linear.

No-throw guarantee: this function never throws exceptions.

{The example demonstrates comparing several JSON types.,operator__notequal}

Since

version 1.0.0

6.9.5.2 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename ScalarType , typename std::enable_if< std::is_scalar< ScalarType >::value, int >::type = 0> bool operator!= (const_reference *lhs,* const ScalarType *rhs*) [friend]

comparison: not equal

comparison: not equal Compares two JSON values for inequality by calculating `not (lhs == rhs)`.

Parameters

in	*lhs*	first JSON value to consider
in	*rhs*	second JSON value to consider

Returns

whether the values *lhs* and *rhs* are not equal

Linear.

No-throw guarantee: this function never throws exceptions.

{The example demonstrates comparing several JSON types.,operator__notequal}

Since

version 1.0.0

6.9.5.3 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename ScalarType , typename std::enable_if< std::is_scalar< ScalarType >::value, int >::type = 0> bool operator!= (const ScalarType *lhs,* const_reference *rhs*) [friend]

comparison: not equal

comparison: not equal Compares two JSON values for inequality by calculating `not (lhs == rhs)`.

Parameters

in	*lhs*	first JSON value to consider
in	*rhs*	second JSON value to consider

Returns

whether the values *lhs* and *rhs* are not equal

Linear.

No-throw guarantee: this function never throws exceptions.

{The example demonstrates comparing several JSON types.,operator__notequal}

Since

version 1.0.0

6.9.5.4 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> bool operator< (const_reference *lhs*, const_reference *rhs*) [friend]

comparison: less than

Compares whether one JSON value *lhs* is less than another JSON value *rhs* according to the following rules:

- If *lhs* and *rhs* have the same type, the values are compared using the default < operator.
- Integer and floating-point numbers are automatically converted before comparison
- In case *lhs* and *rhs* have different types, the values are ignored and the order of the types is considered, see operator<(const value_t, const value_t).

Parameters

in	*lhs*	first JSON value to consider
in	*rhs*	second JSON value to consider

Returns

whether *lhs* is less than *rhs*

Linear.

No-throw guarantee: this function never throws exceptions.

{The example demonstrates comparing several JSON types.,operator__less}

Since

version 1.0.0

6.9.5.5 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename ScalarType , typename std::enable_if< std::is_scalar< ScalarType >::value, int >::type = 0> bool operator< (const_reference *lhs*, const ScalarType *rhs*) [friend]

comparison: less than

comparison: less than Compares whether one JSON value *lhs* is less than another JSON value *rhs* according to the following rules:

- If *lhs* and *rhs* have the same type, the values are compared using the default < operator.
- Integer and floating-point numbers are automatically converted before comparison
- In case *lhs* and *rhs* have different types, the values are ignored and the order of the types is considered, see operator<(const value_t, const value_t).

Parameters

in	*lhs*	first JSON value to consider
in	*rhs*	second JSON value to consider

Returns

whether *lhs* is less than *rhs*

Linear.

No-throw guarantee: this function never throws exceptions.

{The example demonstrates comparing several JSON types.,operator__less}

Since

version 1.0.0

6.9.5.6 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename ScalarType , typename std::enable_if< std::is_scalar< ScalarType >::value, int >::type = 0> bool operator< (const ScalarType *lhs,* const_reference *rhs*) [friend]

comparison: less than

comparison: less than Compares whether one JSON value *lhs* is less than another JSON value *rhs* according to the following rules:

- If *lhs* and *rhs* have the same type, the values are compared using the default < operator.
- Integer and floating-point numbers are automatically converted before comparison
- In case *lhs* and *rhs* have different types, the values are ignored and the order of the types is considered, see operator<(const value_t, const value_t).

Parameters

in	*lhs*	first JSON value to consider
in	*rhs*	second JSON value to consider

Returns

whether *lhs* is less than *rhs*

Linear.

No-throw guarantee: this function never throws exceptions.

{The example demonstrates comparing several JSON types.,operator__less}

Since

version 1.0.0

6.9.5.7 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> std::ostream& operator<< (std::ostream & *o,* const basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer > & *j*) [friend]

serialize to stream

Serialize the given JSON value *j* to the output stream *o*. The JSON value will be serialized using the dump member function.

- The indentation of the output can be controlled with the member variable `width` of the output stream *o*. For instance, using the manipulator `std::setw(4)` on *o* sets the indentation level to `4` and the serialization result is the same as calling `dump(4)`.
- The indentation characrer can be controlled with the member variable `fill` of the output stream *o*. For instance, the manipulator 'std::setfill('\t')' sets indentation to use a tab character rather than the default space character.

Parameters

`in,out`	*o*	stream to serialize to
`in`	*j*	JSON value to serialize

Returns

the stream *o*

Linear.

{The example below shows the serialization with different parameters to `width` to adjust the indentation level.,operator_serialize}

Since

version 1.0.0; indentaction character added in version 3.0.0

6.9.5.8 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> JSON_DEPRECATED friend std::istream& operator<< (basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer > & *j,* std::istream & *i*) [friend]

deserialize from stream

Deprecated This stream operator is deprecated and will be removed in a future version of the library. Please use operator>>(std::istream&, basic_json&) instead; that is, replace calls like `j << i;` with `i >> j;`.

Since

version 1.0.0; deprecated since version 3.0.0

6.9.5.9 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> bool operator<= (const_reference *lhs*, const_reference *rhs*) [friend]

comparison: less than or equal

Compares whether one JSON value *lhs* is less than or equal to another JSON value by calculating `not (rhs < lhs)`.

Parameters

in	*lhs*	first JSON value to consider
in	*rhs*	second JSON value to consider

Returns

whether *lhs* is less than or equal to *rhs*

Linear.

No-throw guarantee: this function never throws exceptions.

{The example demonstrates comparing several JSON types.,operator__greater}

Since

version 1.0.0

6.9.5.10 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename ScalarType , typename std::enable_if< std::is_scalar< ScalarType >::value, int >::type = 0> bool operator<= (const_reference *lhs*, const ScalarType *rhs*) [friend]

comparison: less than or equal

comparison: less than or equal Compares whether one JSON value *lhs* is less than or equal to another JSON value by calculating `not (rhs < lhs)`.

Parameters

in	*lhs*	first JSON value to consider
in	*rhs*	second JSON value to consider

Returns

whether *lhs* is less than or equal to *rhs*

Linear.

No-throw guarantee: this function never throws exceptions.

{The example demonstrates comparing several JSON types.,operator__greater}

Since

version 1.0.0

6.9.5.11 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename ScalarType , typename std::enable_if< std::is_scalar< ScalarType >::value, int >::type = 0> bool operator<= (const ScalarType *lhs,* const_reference *rhs*) [friend]

comparison: less than or equal

comparison: less than or equal Compares whether one JSON value *lhs* is less than or equal to another JSON value by calculating `not (rhs < lhs)`.

Parameters

in	*lhs*	first JSON value to consider
in	*rhs*	second JSON value to consider

Returns

whether *lhs* is less than or equal to *rhs*

Linear.

No-throw guarantee: this function never throws exceptions.

{The example demonstrates comparing several JSON types.,operator__greater}

Since

version 1.0.0

6.9.5.12 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> bool operator== (const_reference *lhs*, const_reference *rhs*) [friend]

comparison: equal

Compares two JSON values for equality according to the following rules:

- Two JSON values are equal if (1) they are from the same type and (2) their stored values are the same according to their respective `operator==`.
- Integer and floating-point numbers are automatically converted before comparison. Note than two NaN values are always treated as unequal.
- Two JSON null values are equal.

Note

Floating-point inside JSON values numbers are compared with `json::number_float_t::operator==` which is `double::operator==` by default. To compare floating-point while respecting an epsilon, an alternative `comparison function` could be used, for instance

```
template <typename T, typename = typename std::enable_if<std::is_floating_point<T>::value, T>
      ::type>
inline bool is_same(T a, T b, T epsilon = std::numeric_limits<T>::epsilon()) noexcept
{
    return std::abs(a - b) <= epsilon;
}
```

NaN values never compare equal to themselves or to other NaN values.

Parameters

in	*lhs*	first JSON value to consider
in	*rhs*	second JSON value to consider

Returns

whether the values *lhs* and *rhs* are equal

No-throw guarantee: this function never throws exceptions.

Linear.

{The example demonstrates comparing several JSON types.,operator__equal}

Since

version 1.0.0

6.9.5.13 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename ScalarType , typename std::enable_if< std::is_scalar< ScalarType >::value, int >::type = 0> bool operator== (const_reference *lhs*, const ScalarType *rhs*) [friend]

comparison: equal

comparison: equal Compares two JSON values for equality according to the following rules:

- Two JSON values are equal if (1) they are from the same type and (2) their stored values are the same according to their respective `operator==`.
- Integer and floating-point numbers are automatically converted before comparison. Note than two NaN values are always treated as unequal.
- Two JSON null values are equal.

Note

Floating-point inside JSON values numbers are compared with `json::number_float_t::operator==` which is `double::operator==` by default. To compare floating-point while respecting an epsilon, an alternative `comparison function` could be used, for instance

```
template <typename T, typename = typename std::enable_if<std::is_floating_point<T>::value, T>
      ::type>
inline bool is_same(T a, T b, T epsilon = std::numeric_limits<T>::epsilon()) noexcept
{
    return std::abs(a - b) <= epsilon;
}
```

NaN values never compare equal to themselves or to other NaN values.

Parameters

in	*lhs*	first JSON value to consider
in	*rhs*	second JSON value to consider

Returns

whether the values *lhs* and *rhs* are equal

No-throw guarantee: this function never throws exceptions.

Linear.

{The example demonstrates comparing several JSON types.,operator__equal}

Since

version 1.0.0

6.9.5.14 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename ScalarType , typename std::enable_if< std::is_scalar< ScalarType >::value, int >::type = 0> bool operator== (const ScalarType *lhs,* const_reference *rhs*) [friend]

comparison: equal

comparison: equal Compares two JSON values for equality according to the following rules:

- Two JSON values are equal if (1) they are from the same type and (2) their stored values are the same according to their respective `operator==`.
- Integer and floating-point numbers are automatically converted before comparison. Note than two NaN values are always treated as unequal.
- Two JSON null values are equal.

Note

Floating-point inside JSON values numbers are compared with `json::number_float_t::operator==` which is `double::operator==` by default. To compare floating-point while respecting an epsilon, an alternative `comparison function` could be used, for instance

```
template <typename T, typename = typename std::enable_if<std::is_floating_point<T>::value, T>
      ::type>
inline bool is_same(T a, T b, T epsilon = std::numeric_limits<T>::epsilon()) noexcept
{
    return std::abs(a - b) <= epsilon;
}
```

NaN values never compare equal to themselves or to other NaN values.

Parameters

in	*lhs*	first JSON value to consider
in	*rhs*	second JSON value to consider

Returns

whether the values *lhs* and *rhs* are equal

No-throw guarantee: this function never throws exceptions.

Linear.

{The example demonstrates comparing several JSON types.,operator__equal}

Since

version 1.0.0

6.9.5.15 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> bool operator> (const_reference *lhs,* const_reference *rhs*) [friend]

comparison: greater than

Compares whether one JSON value *lhs* is greater than another JSON value by calculating `not (lhs <= rhs)`.

Parameters

in	*lhs*	first JSON value to consider
in	*rhs*	second JSON value to consider

Returns

whether *lhs* is greater than to *rhs*

Linear.

No-throw guarantee: this function never throws exceptions.

{The example demonstrates comparing several JSON types.,operator__lessequal}

Since

version 1.0.0

6.9.5.16 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename ScalarType , typename std::enable_if< std::is_scalar< ScalarType >::value, int >::type = 0> bool operator> (const_reference *lhs,* const ScalarType *rhs*) [friend]

comparison: greater than

comparison: greater than Compares whether one JSON value *lhs* is greater than another JSON value by calculating `not (lhs <= rhs)`.

Parameters

in	*lhs*	first JSON value to consider
in	*rhs*	second JSON value to consider

Returns

whether *lhs* is greater than to *rhs*

Linear.

No-throw guarantee: this function never throws exceptions.

{The example demonstrates comparing several JSON types.,operator__lessequal}

Since

version 1.0.0

6.9.5.17 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename ScalarType , typename std::enable_if< std::is_scalar< ScalarType >::value, int >::type = 0> bool operator> (const ScalarType *lhs,* const_reference *rhs*) [friend]

comparison: greater than

comparison: greater than Compares whether one JSON value *lhs* is greater than another JSON value by calculating `not (lhs <= rhs)`.

Parameters

in	*lhs*	first JSON value to consider
in	*rhs*	second JSON value to consider

Returns

whether *lhs* is greater than to *rhs*

Linear.

No-throw guarantee: this function never throws exceptions.

{The example demonstrates comparing several JSON types.,operator__lessequal}

Since

version 1.0.0

6.9.5.18 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> bool operator>= (const_reference *lhs,* const_reference *rhs*) [friend]

comparison: greater than or equal

Compares whether one JSON value *lhs* is greater than or equal to another JSON value by calculating `not (lhs < rhs)`.

Parameters

in	*lhs*	first JSON value to consider
in	*rhs*	second JSON value to consider

Returns

whether *lhs* is greater than or equal to *rhs*

Linear.

No-throw guarantee: this function never throws exceptions.

{The example demonstrates comparing several JSON types.,operator__greaterequal}

Since

version 1.0.0

6.9.5.19 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename ScalarType , typename std::enable_if< std::is_scalar< ScalarType >::value, int >::type = 0> bool operator>= (const_reference *lhs,* const ScalarType *rhs*) [friend]

comparison: greater than or equal

comparison: greater than or equal Compares whether one JSON value *lhs* is greater than or equal to another JSON value by calculating `not (lhs < rhs)`.

Parameters

in	*lhs*	first JSON value to consider
in	*rhs*	second JSON value to consider

Returns

whether *lhs* is greater than or equal to *rhs*

Linear.

No-throw guarantee: this function never throws exceptions.

{The example demonstrates comparing several JSON types.,operator__greaterequal}

Since

version 1.0.0

6.9.5.20 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> template<typename ScalarType , typename std::enable_if< std::is_scalar< ScalarType >::value, int >::type = 0> bool operator>= (const ScalarType *lhs,* const_reference *rhs*) [friend]

comparison: greater than or equal

comparison: greater than or equal Compares whether one JSON value *lhs* is greater than or equal to another JSON value by calculating `not (lhs < rhs)`.

Parameters

in	*lhs*	first JSON value to consider
in	*rhs*	second JSON value to consider

Returns

whether *lhs* is greater than or equal to *rhs*

Linear.

No-throw guarantee: this function never throws exceptions.

{The example demonstrates comparing several JSON types.,operator__greaterequal}

Since

version 1.0.0

6.9.5.21 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> JSON_DEPRECATED friend std::ostream& operator>> (const basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer > & *j,* std::ostream & *o*) [friend]

serialize to stream

Deprecated This stream operator is deprecated and will be removed in a future version of the library. Please use operator<<(std::ostream&, const basic_json&) instead; that is, replace calls like `j >> o;` with `o << j;`.

Since

version 1.0.0; deprecated since version 3.0.0

6.9.5.22 template<template< typename U, typename V, typename...Args > class ObjectType = std::map, template< typename U, typename...Args > class ArrayType = std::vector, class StringType = std::string, class BooleanType = bool, class NumberIntegerType = std::int64_t, class NumberUnsignedType = std::uint64_t, class NumberFloatType = double, template< typename U > class AllocatorType = std::allocator, template< typename T, typename SFINAE=void > class JSONSerializer = adl_serializer> std::istream& operator>> (std::istream & *i,* basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer > & *j*) [friend]

deserialize from stream

Deserializes an input stream to a JSON value.

Parameters

in,out	*i*	input stream to read a serialized JSON value from
in,out	*j*	JSON value to write the deserialized input to

Exceptions

parse_error.101	in case of an unexpected token
parse_error.102	if to_unicode fails or surrogate error
parse_error.103	if to_unicode fails

Linear in the length of the input. The parser is a predictive LL(1) parser.

Note

A UTF-8 byte order mark is silently ignored.

{The example below shows how a JSON value is constructed by reading a serialization from a stream.,operator_↩ deserialize}

See also

parse(std::istream&, const parser_callback_t) for a variant with a parser callback function to filter values while parsing

Since

version 1.0.0

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.10 nlohmann::detail::binary_reader< BasicJsonType > Class Template Reference

deserialization of CBOR and MessagePack values

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

Public Member Functions

- binary_reader (input_adapter_t adapter)
  create a binary reader
- BasicJsonType parse_cbor (const bool strict)
  create a JSON value from CBOR input
- BasicJsonType parse_msgpack (const bool strict)
  create a JSON value from MessagePack input

Static Public Member Functions

- static constexpr bool little_endianess (int num=1) noexcept
  determine system byte order

6.10.1 Detailed Description

template<typename BasicJsonType>
class nlohmann::detail::binary_reader< BasicJsonType >

deserialization of CBOR and MessagePack values

6.10.2 Constructor & Destructor Documentation

6.10.2.1 template<typename BasicJsonType > nlohmann::detail::binary_reader< BasicJsonType >::binary_reader (input_adapter_t *adapter*) [inline], [explicit]

create a binary reader

Parameters

in	*adapter*	input adapter to read from

6.10.3 Member Function Documentation

6.10.3.1 template<typename BasicJsonType > static constexpr bool nlohmann::detail::binary_reader< BasicJsonType >::little_endianess (int *num* = 1) [inline], [static], [noexcept]

determine system byte order

Returns

true if and only if system's byte order is little endian

Note

from http://stackoverflow.com/a/1001328/266378

6.10.3.2 template<typename BasicJsonType > BasicJsonType nlohmann::detail::binary_reader< BasicJsonType >::parse_cbor (const bool *strict*) [inline]

create a JSON value from CBOR input

Parameters

in	*strict*	whether to expect the input to be consumed completed

Returns

JSON value created from CBOR input

Exceptions

parse_error.110	if input ended unexpectedly or the end of file was not reached when *strict* was set to true
parse_error.112	if unsupported byte was read

6.10.3.3 template<typename BasicJsonType > BasicJsonType nlohmann::detail::binary_reader< BasicJsonType >::parse_msgpack (const bool *strict*) [inline]

create a JSON value from MessagePack input

Parameters

in	*strict*	whether to expect the input to be consumed completed

Returns

JSON value created from MessagePack input

Exceptions

parse_error.110	if input ended unexpectedly or the end of file was not reached when *strict* was set to true
parse_error.112	if unsupported byte was read

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.11 nlohmann::detail::binary_writer< BasicJsonType, CharType > Class Template Reference

serialization to CBOR and MessagePack values

`#include <json.hpp>`

Public Member Functions

- binary_writer (output_adapter_t< CharType > adapter)
  create a binary writer
- void write_cbor (const BasicJsonType &j)
  [in] j JSON value to serialize
- void write_msgpack (const BasicJsonType &j)
  [in] j JSON value to serialize

6.11.1 Detailed Description

template<typename BasicJsonType, typename CharType>
class nlohmann::detail::binary_writer< BasicJsonType, CharType >

serialization to CBOR and MessagePack values

6.11.2 Constructor & Destructor Documentation

6.11.2.1 template<typename BasicJsonType , typename CharType > nlohmann::detail::binary_writer< BasicJsonType, CharType >::binary_writer (output_adapter_t< CharType > *adapter*) [inline], [explicit]

create a binary writer

Parameters

in	*adapter*	output adapter to write to

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.12 tao::operators::bitwise< T, U > Class Template Reference

Inheritance diagram for tao::operators::bitwise< T, U >:

Collaboration diagram for tao::operators::bitwise< T, U >:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.13 tao::operators::bitwise_left< T, U > Class Template Reference

Inheritance diagram for tao::operators::bitwise_left< T, U >:

Collaboration diagram for tao::operators::bitwise_left< T, U >:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.14 commutative_addable Class Reference

Inheritance diagram for commutative_addable:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/su3.h

6.15 commutative_addable Class Reference

Inheritance diagram for commutative_addable:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.16 commutative_andable Class Reference

Inheritance diagram for commutative_andable:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.17 tao::operators::commutative_bitwise< T, U > Class Template Reference

Inheritance diagram for tao::operators::commutative_bitwise< T, U >:

Collaboration diagram for tao::operators::commutative_bitwise< T, U >:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.18 commutative_multipliable Class Reference

Inheritance diagram for commutative_multipliable:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.19 commutative_orable Class Reference

Inheritance diagram for commutative_orable:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.20 tao::operators::commutative_ring< T, U > Class Template Reference

Inheritance diagram for tao::operators::commutative_ring< T, U >:

Collaboration diagram for tao::operators::commutative_ring< T, U >:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.21 tao::operators::commutative_ring< T > Class Template Reference

Inheritance diagram for tao::operators::commutative_ring< T >:

Collaboration diagram for tao::operators::commutative_ring< T >:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.22 commutative_xorable Class Reference

Inheritance diagram for commutative_xorable:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.23 complex Struct Reference

Collaboration diagram for complex:

Public Member Functions

- void **printComplex** ()
- double **norm** ()

Public Attributes

- double **real**
- double **imag**

The documentation for this struct was generated from the following files:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/complex.h
- /home/giovanni/Desktop/LatticeYangMills/src/Math/complex.cpp

6.24 nlohmann::detail::conjunction<... > Struct Template Reference

Inheritance diagram for nlohmann::detail::conjunction<... >:

Collaboration diagram for nlohmann::detail::conjunction<... >:

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.25 nlohmann::detail::conjunction< B1 > Struct Template Reference

Inheritance diagram for nlohmann::detail::conjunction< B1 >:

Collaboration diagram for nlohmann::detail::conjunction< B1 >:

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.26 nlohmann::detail::conjunction< B1, Bn... > Struct Template Reference

Inheritance diagram for nlohmann::detail::conjunction< B1, Bn... >:

Collaboration diagram for nlohmann::detail::conjunction< B1, Bn... >:

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.27 tao::operators::decrementable< T > Class Template Reference

Inheritance diagram for tao::operators::decrementable< T >:

Friends

- T **operator--** (T &arg, int) noexcept(noexcept(T(arg),--arg, T(std::declval< T >())))

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.28 dividable Class Reference

Inheritance diagram for dividable:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.29 dividable_left Class Reference

Inheritance diagram for dividable_left:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.30 EnergyDensity Class Reference

Inheritance diagram for EnergyDensity:

Collaboration diagram for EnergyDensity:

Public Member Functions

- void **initObservable** (GluonField *lattice)
- void **compute** ()

Additional Inherited Members

The documentation for this class was generated from the following files:

- /home/giovanni/Desktop/LatticeYangMills/include/Observables/energydensity.h
- /home/giovanni/Desktop/LatticeYangMills/src/Observables/energydensity.cpp

6.31 tao::operators::equality_comparable< T, U > Class Template Reference

Inheritance diagram for tao::operators::equality_comparable< T, U >:

Friends

- bool **operator!=** (const T &lhs, const U &rhs) noexcept(noexcept(static_cast< bool >(lhs==rhs)))
- bool **operator==** (const U &lhs, const T &rhs) noexcept(noexcept(static_cast< bool >(rhs==lhs)))
- bool **operator!=** (const U &lhs, const T &rhs) noexcept(noexcept(static_cast< bool >(rhs!=lhs)))

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.32 tao::operators::equality_comparable< T > Class Template Reference

Friends

- bool **operator!=** (const T &lhs, const T &rhs) noexcept(noexcept(static_cast< bool >(lhs==rhs)))

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.33 tao::operators::equivalent< T, U > Class Template Reference

Friends

- bool **operator==** (const T &lhs, const U &rhs) noexcept(noexcept(static_cast< bool >(lhs< rhs), static_↩ cast< bool >(lhs > rhs)))

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.34 tao::operators::equivalent< T > Class Template Reference

Friends

- bool **operator==** (const T &lhs, const T &rhs) noexcept(noexcept(static_cast< bool >(lhs< rhs)))

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.35 nlohmann::detail::exception Class Reference

general exception of the basic_json class

`#include <json.hpp>`

Inheritance diagram for nlohmann::detail::exception:

Collaboration diagram for nlohmann::detail::exception:

Public Member Functions

- const char * what () const noexceptoverride
  returns the explanatory string

Public Attributes

- const int id
  the id of the exception

Protected Member Functions

- **exception** (int id_, const char *what_arg)

Static Protected Member Functions

- static std::string **name** (const std::string &ename, int id_)

6.35.1 Detailed Description

general exception of the basic_json class

This class is an extension of `std::exception` objects with a member *id* for exception ids. It is used as the base class for all exceptions thrown by the basic_json class. This class can hence be used as "wildcard" to catch exceptions.

Subclasses:

- parse_error for exceptions indicating a parse error
- invalid_iterator for exceptions indicating errors with iterators
- type_error for exceptions indicating executing a member function with a wrong type
- out_of_range for exceptions indicating access out of the defined range
- other_error for exceptions indicating other library errors

{The following code shows how arbitrary library exceptions can be caught.,exception}

Since

version 3.0.0

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.36 nlohmann::detail::external_constructor< value_t > Struct Template Reference

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.37 nlohmann::detail::external_constructor< value_t::array > Struct Template Reference

Static Public Member Functions

- template<typename BasicJsonType >
  static void **construct** (BasicJsonType &j, const typename BasicJsonType::array_t &arr)
- template<typename BasicJsonType >
  static void **construct** (BasicJsonType &j, typename BasicJsonType::array_t &&arr)
- template<typename BasicJsonType , typename CompatibleArrayType , enable_if_t< not std::is_same< CompatibleArrayType, typename BasicJsonType::array_t >::value, int > = 0>
  static void **construct** (BasicJsonType &j, const CompatibleArrayType &arr)
- template<typename BasicJsonType >
  static void **construct** (BasicJsonType &j, const std::vector< bool > &arr)
- template<typename BasicJsonType , typename T , enable_if_t< std::is_convertible< T, BasicJsonType >::value, int > = 0>
  static void **construct** (BasicJsonType &j, const std::valarray< T > &arr)

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.38 nlohmann::detail::external_constructor< value_t::boolean > Struct Template Reference

Static Public Member Functions

- template<typename BasicJsonType >
  static void **construct** (BasicJsonType &j, typename BasicJsonType::boolean_t b) noexcept

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.39 nlohmann::detail::external_constructor< value_t::number_float > Struct Template Reference

Static Public Member Functions

- template<typename BasicJsonType >
  static void **construct** (BasicJsonType &j, typename BasicJsonType::number_float_t val) noexcept

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.40 nlohmann::detail::external_constructor< value_t::number_integer > Struct Template Reference

Static Public Member Functions

- template<typename BasicJsonType >
  static void **construct** (BasicJsonType &j, typename BasicJsonType::number_integer_t val) noexcept

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.41 nlohmann::detail::external_constructor< value_t::number_unsigned > Struct Template Reference

Static Public Member Functions

- template<typename BasicJsonType >
  static void **construct** (BasicJsonType &j, typename BasicJsonType::number_unsigned_t val) noexcept

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.42 nlohmann::detail::external_constructor< value_t::object > Struct Template Reference

Static Public Member Functions

- template<typename BasicJsonType >
  static void **construct** (BasicJsonType &j, const typename BasicJsonType::object_t &obj)
- template<typename BasicJsonType >
  static void **construct** (BasicJsonType &j, typename BasicJsonType::object_t &&obj)
- template<typename BasicJsonType , typename CompatibleObjectType , enable_if_t< not std::is_same< CompatibleObjectType, typename BasicJsonType::object_t >::value, int > = 0>
  static void **construct** (BasicJsonType &j, const CompatibleObjectType &obj)

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.43 nlohmann::detail::external_constructor< value_t::string > Struct Template Reference

Static Public Member Functions

- template<typename BasicJsonType >
  static void **construct** (BasicJsonType &j, const typename BasicJsonType::string_t &s)
- template<typename BasicJsonType >
  static void **construct** (BasicJsonType &j, typename BasicJsonType::string_t &&s)

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.44 Field< T, N > Class Template Reference

Collaboration diagram for Field< T, N >:

Public Member Functions

- **Field** (std::array< int, 4 > size)
- Lattice< T > & **operator[ ]** (int mu)

Public Attributes

- int **m_dimensions**
- std::array< int, 4 > **m_size**

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/field.h

6.45 tao::operators::field< T, U > Class Template Reference

Inheritance diagram for tao::operators::field< T, U >:

Collaboration diagram for tao::operators::field< T, U >:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.46 tao::operators::field< T > Class Template Reference

Inheritance diagram for tao::operators::field< T >:

Collaboration diagram for tao::operators::field< T >:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.47 nlohmann::detail::from_json_fn Struct Reference

Public Member Functions

- template<typename BasicJsonType , typename T >
  void **operator()** (const BasicJsonType &j, T &val) const noexcept(noexcept(std::declval< from_json_↩ fn >().call(j, val, priority_tag< 1 >{})))

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.48 GaugeFieldFactory Class Reference

Inheritance diagram for GaugeFieldFactory:

Collaboration diagram for GaugeFieldFactory:

Public Member Functions

- **GaugeFieldFactory** (int MCSteps, int thermSteps, int NConf, double epsilon, std::string startType)
- void **generateConfigurations** ()
- std::vector< double > & **getObsValues** ()
- void **execute** ()

Additional Inherited Members

The documentation for this class was generated from the following files:

- /home/giovanni/Desktop/LatticeYangMills/include/Apps/gaugefieldfactory.h
- /home/giovanni/Desktop/LatticeYangMills/src/Apps/gaugefieldfactory.cpp

6.49 GaugeFieldReader Class Reference

Inheritance diagram for GaugeFieldReader:

Collaboration diagram for GaugeFieldReader:

Public Member Functions

- void **initGFR** ()
- void **sampleConfigurations** ()
- void **addObservable** (Observable *observable)
- const char * **getOutDir** ()
- std::array< int, 4 > & **getSize** ()
- std::vector< double > & **getObsValues** ()
- std::vector< Observable * > & **getObs** ()
- void **execute** ()

Additional Inherited Members

The documentation for this class was generated from the following files:

- /home/giovanni/Desktop/LatticeYangMills/include/Apps/gaugefieldreader.h
- /home/giovanni/Desktop/LatticeYangMills/src/Apps/gaugefieldreader.cpp

6.50 nlohmann::detail::has_from_json< BasicJsonType, T > Struct Template Reference

Collaboration diagram for nlohmann::detail::has_from_json< BasicJsonType, T >:

Static Public Attributes

- static constexpr bool **value**

6.50.1 Member Data Documentation

6.50.1.1 template<typename BasicJsonType , typename T > constexpr bool nlohmann::detail::has_from_json< BasicJsonType, T >::value [static]

Initial value:

```
= std::is_integral<decltype(
                                    detect(std::declval<typename BasicJsonType::template
      json_serializer<T, void>>()))>::value
```

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.51 nlohmann::detail::has_non_default_from_json< BasicJsonType, T > Struct Template Reference

Collaboration diagram for nlohmann::detail::has_non_default_from_json< BasicJsonType, T >:

Static Public Attributes

- static constexpr bool **value**

6.51.1 Member Data Documentation

6.51.1.1 template<typename BasicJsonType , typename T > constexpr bool nlohmann::detail::has_non_default_from_json< BasicJsonType, T >::value [static]

Initial value:

```
= std::is_integral<decltype(detect(
                                    std::declval<typename BasicJsonType::template json_serializer<T,
       void>>()))>::value
```

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.52 nlohmann::detail::has_to_json< BasicJsonType, T > Struct Template Reference

Collaboration diagram for nlohmann::detail::has_to_json< BasicJsonType, T >:

Static Public Attributes

- static constexpr bool **value**

6.52.1 Member Data Documentation

6.52.1.1 template<typename BasicJsonType , typename T > constexpr bool nlohmann::detail::has_to_json< BasicJsonType, T >::value [static]

Initial value:

```
= std::is_integral<decltype(detect(
                                    std::declval<typename BasicJsonType::template json_serializer<T,
       void>>()))>::value
```

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.53 std::hash< nlohmann::json > Struct Template Reference

hash value for JSON objects

`#include <json.hpp>`

Public Member Functions

- std::size_t operator() (const nlohmann::json &j) const
  return a hash value for a JSON object

6.53.1 Detailed Description

template<>
struct std::hash< nlohmann::json >

hash value for JSON objects

6.53.2 Member Function Documentation

6.53.2.1 std::size_t std::hash< nlohmann::json >::operator() (const nlohmann::json & *j*) const [inline]

return a hash value for a JSON object

Since

version 1.0.0

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.54 tao::operators::incrementable< T > Class Template Reference

Inheritance diagram for tao::operators::incrementable< T >:

Friends

- T **operator++** (T &arg, int) noexcept(noexcept(T(arg),++arg, T(std::declval< T >())))

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.55 nlohmann::detail::index_sequence< Ints > Struct Template Reference

Public Types

- using **type** = index_sequence
- using **value_type** = std::size_t

Static Public Member Functions

- static constexpr std::size_t **size** () noexcept

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.56 nlohmann::detail::input_adapter Class Reference

Public Member Functions

- input_adapter (std::istream &i)

  input adapter for input stream
- input_adapter (std::istream &&i)

  input adapter for input stream
- template<typename CharT , typename std::enable_if< std::is_pointer< CharT >::value andstd::is_integral< typename std::remove↩ _pointer< CharT >::type >::value andsizeof(typename std::remove_pointer< CharT >::type)==1, int >::type = 0>
  input_adapter (CharT b, std::size_t l)

  input adapter for buffer
- template<typename CharT , typename std::enable_if< std::is_pointer< CharT >::value andstd::is_integral< typename std::remove↩ _pointer< CharT >::type >::value andsizeof(typename std::remove_pointer< CharT >::type)==1, int >::type = 0>
  input_adapter (CharT b)

  input adapter for string literal
- template<class IteratorType , typename std::enable_if< std::is_same< typename std::iterator_traits< IteratorType >::iterator_category, std::random_access_iterator_tag >::value, int >::type = 0>
  input_adapter (IteratorType first, IteratorType last)

  input adapter for iterator range with contiguous storage
- template<class T , std::size_t N>
  input_adapter (T(&array)[N])

  input adapter for array
- template<class ContiguousContainer , typename std::enable_if< not std::is_pointer< ContiguousContainer >::value andstd::is↩ _base_of< std::random_access_iterator_tag, typename std::iterator_traits< decltype(std::begin(std::declval< ContiguousContainer const >()))>::iterator_category >::value, int >::type = 0>
  input_adapter (const ContiguousContainer &c)

  input adapter for contiguous container
- **operator input_adapter_t** ()

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.57 nlohmann::detail::input_adapter_protocol Struct Reference

abstract input adapter interface

`#include <json.hpp>`

Inheritance diagram for nlohmann::detail::input_adapter_protocol:

Public Member Functions

- virtual std::char_traits< char >::int_type get_character ()=0
  get a character [0,255] or std::char_traits<char>::eof().
- virtual void unget_character ()=0
  restore the last non-eof() character to input

6.57.1 Detailed Description

abstract input adapter interface

Produces a stream of std::char_traits<char>::int_type characters from a std::istream, a buffer, or some other input type. Accepts the return of exactly one non-EOF character for future input. The int_type characters returned consist of all valid char values as positive values (typically unsigned char), plus an EOF value outside that range, specified by the value of the function std::char_traits<char>::eof(). This value is typically -1, but could be any arbitrary value which is not a valid char value.

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.58 nlohmann::detail::input_buffer_adapter Class Reference

input adapter for buffer input

`#include <json.hpp>`

Inheritance diagram for nlohmann::detail::input_buffer_adapter:

Collaboration diagram for nlohmann::detail::input_buffer_adapter:

Public Member Functions

- **input_buffer_adapter** (const char *b, const std::size_t l)
- **input_buffer_adapter** (const input_buffer_adapter &)=delete
- input_buffer_adapter & **operator=** (input_buffer_adapter &)=delete
- std::char_traits< char >::int_type get_character () noexceptoverride
  get a character [0,255] or std::char_traits<char>::eof().
- void unget_character () noexceptoverride
  restore the last non-eof() character to input

6.58.1 Detailed Description

input adapter for buffer input

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.59 nlohmann::detail::input_stream_adapter Class Reference

`#include <json.hpp>`

Inheritance diagram for nlohmann::detail::input_stream_adapter:

Collaboration diagram for nlohmann::detail::input_stream_adapter:

Public Member Functions

- **input_stream_adapter** (std::istream &i)
- **input_stream_adapter** (const input_stream_adapter &)=delete
- input_stream_adapter & **operator=** (input_stream_adapter &)=delete
- std::char_traits< char >::int_type get_character () override

  get a character [0,255] or std::char_traits<char>::eof().
- void unget_character () override

  restore the last non-eof() character to input

6.59.1 Detailed Description

Input adapter for a (caching) istream. Ignores a UFT Byte Order Mark at beginning of input. Does not support changing the underlying std::streambuf in mid-input. Maintains underlying std::istream and std::streambuf to support subsequent use of standard std::istream operations to process any input characters following those used in parsing the JSON input. Clears the std::istream flags; any input errors (e.g., EOF) will be detected by the first subsequent call for input from the std::istream.

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.60 LatticeIO::InputConf Class Reference

Static Public Member Functions

- static void **readConf** (GluonField &lattice, int confNum)
- static void **readConf** (GluonField &lattice, const char *inputFile)
- static void **readSubLattice** (GluonField &lattice, int confNum)
- static void **setInputDir** (std::string inputDir)
- static void **getInputList** (std::vector< std::string > &inputConfList)

The documentation for this class was generated from the following files:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/inputconf.h
- /home/giovanni/Desktop/LatticeYangMills/src/InputOutput/inputconf.cpp

6.61 nlohmann::detail::internal_iterator< BasicJsonType > Struct Template Reference

an iterator value

`#include <json.hpp>`

Collaboration diagram for nlohmann::detail::internal_iterator< BasicJsonType >:

Public Attributes

- BasicJsonType::object_t::iterator object_iterator {}
  iterator for JSON objects
- BasicJsonType::array_t::iterator array_iterator {}
  iterator for JSON arrays
- primitive_iterator_t primitive_iterator {}
  generic iterator for all other types

6.61.1 Detailed Description

template<typename BasicJsonType>
struct nlohmann::detail::internal_iterator< BasicJsonType >

an iterator value

Note

This structure could easily be a union, but MSVC currently does not allow unions members with complex constructors, see https://github.com/nlohmann/json/pull/105.

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.62 nlohmann::detail::invalid_iterator Class Reference

exception indicating errors with iterators

`#include <json.hpp>`

Inheritance diagram for nlohmann::detail::invalid_iterator:

Collaboration diagram for nlohmann::detail::invalid_iterator:

Static Public Member Functions

- static invalid_iterator **create** (int id_, const std::string &what_arg)

Additional Inherited Members

6.62.1 Detailed Description

exception indicating errors with iterators

This exception is thrown if iterators passed to a library function do not match the expected semantics.

Exceptions have ids 2xx.

name / id	example message	description
json.exception.invalid_iterator.201	iterators are not compatible	The iterators passed to constructor basic_json(InputIT first, InputIT last) are not compatible, meaning they do not belong to the same container. Therefore, the range (*first*, *last*) is invalid.
json.exception.invalid_iterator.202	iterator does not fit current value	In an erase or insert function, the passed iterator *pos* does not belong to the JSON value for which the function was called. It hence does not define a valid position for the deletion/insertion.
json.exception.invalid_iterator.203	iterators do not fit current value	Either iterator passed to function erase(IteratorType first, IteratorType last) does not belong to the JSON value from which values shall be erased. It hence does not define a valid range to delete values from.
json.exception.invalid_iterator.204	iterators out of range	When an iterator range for a primitive type (number, boolean, or string) is passed to a constructor or an erase function, this range has to be exactly (begin(), end()), because this is the only way the single stored value is expressed. All other ranges are invalid.
json.exception.invalid_iterator.205	iterator out of range	When an iterator for a primitive type (number, boolean, or string) is passed to an erase function, the iterator has to be the begin() iterator, because it is the only way to address the stored value. All other iterators are invalid.
json.exception.invalid_iterator.206	cannot construct with iterators from null	The iterators passed to constructor basic_json(InputIT first, InputIT last) belong to a JSON null value and hence to not define a valid range.
json.exception.invalid_iterator.207	cannot use key() for non-object iterators	The key() member function can only be used on iterators belonging to a JSON object, because other types do not have a concept of a key.
json.exception.invalid_iterator.208	cannot use operator[] for object iterators	The operator[] to specify a concrete offset cannot be used on iterators belonging to a JSON object, because JSON objects are unordered.
json.exception.invalid_iterator.209	cannot use offsets with object iterators	The offset operators (+, -, +=, -=) cannot be used on iterators belonging to a JSON object, because JSON objects are unordered.

name / id	example message	description
json.exception.invalid_iterator.210	iterators do not fit	The iterator range passed to the insert function are not compatible, meaning they do not belong to the same container. Therefore, the range (*first*, *last*) is invalid.
json.exception.invalid_iterator.211	passed iterators may not belong to container	The iterator range passed to the insert function must not be a subrange of the container to insert to.
json.exception.invalid_iterator.212	cannot compare iterators of different containers	When two iterators are compared, they must belong to the same container.
json.exception.invalid_iterator.213	cannot compare order of object iterators	The order of object iterators cannot be compared, because JSON objects are unordered.
json.exception.invalid_iterator.214	cannot get value	Cannot get value for iterator: Either the iterator belongs to a null value or it is an iterator to a primitive type (number, boolean, or string), but the iterator is different to begin().

{The following code shows how an `invalid_iterator` exception can be caught.,invalid_iterator}

See also

exception for the base class of the library exceptions
parse_error for exceptions indicating a parse error
type_error for exceptions indicating executing a member function with a wrong type
out_of_range for exceptions indicating access out of the defined range
other_error for exceptions indicating other library errors

Since

version 3.0.0

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.63 nlohmann::detail::is_basic_json< typename > Struct Template Reference

Inheritance diagram for nlohmann::detail::is_basic_json< typename >:

Collaboration diagram for nlohmann::detail::is_basic_json< typename >:

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.64 nlohmann::detail::is_basic_json< NLOHMANN_BASIC_JSON_TPL > Struct Reference

Inheritance diagram for nlohmann::detail::is_basic_json< NLOHMANN_BASIC_JSON_TPL >:

Collaboration diagram for nlohmann::detail::is_basic_json< NLOHMANN_BASIC_JSON_TPL >:

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.65 nlohmann::detail::is_basic_json_nested_type< BasicJsonType, T > Struct Template Reference

Collaboration diagram for nlohmann::detail::is_basic_json_nested_type< BasicJsonType, T >:

Static Public Attributes

- static auto constexpr **value**

6.65.1 Member Data Documentation

6.65.1.1 template<typename BasicJsonType , typename T > auto constexpr nlohmann::detail::is_basic_json_nested_type< BasicJsonType, T >::value [static]

Initial value:

```
= std::is_same<T, typename BasicJsonType::iterator>::value or
                                std::is_same<T, typename BasicJsonType::const_iterator>::value or
                                std::is_same<T, typename BasicJsonType::reverse_iterator>::value or
                                std::is_same<T, typename BasicJsonType::const_reverse_iterator>::value
```

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.66 nlohmann::detail::is_compatible_array_type< BasicJsonType, CompatibleArrayType > Struct Template Reference

Collaboration diagram for nlohmann::detail::is_compatible_array_type< BasicJsonType, CompatibleArrayType >:

Static Public Attributes

- static auto constexpr **value**

6.66.1 Member Data Documentation

6.66.1.1 template<class BasicJsonType , class CompatibleArrayType > auto constexpr nlohmann::detail::is_compatible_array_type< BasicJsonType, CompatibleArrayType >::value [static]

Initial value:

```
=
        conjunction<negation<std::is_same<void, CompatibleArrayType>>,
        negation<is_compatible_object_type<
        BasicJsonType, CompatibleArrayType>>,
        negation<std::is_constructible<typename BasicJsonType::string_t,
        CompatibleArrayType>>,
        negation<is_basic_json_nested_type<BasicJsonType, CompatibleArrayType>>,
        has_value_type<CompatibleArrayType>,
        has_iterator<CompatibleArrayType>>::value
```

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.67 nlohmann::detail::is_compatible_integer_type< RealIntegerType, CompatibleNumberIntegerType > Struct Template Reference

Collaboration diagram for nlohmann::detail::is_compatible_integer_type< RealIntegerType, CompatibleNumberIntegerType >:

Static Public Attributes

- static constexpr auto **value**

6.67.1 Member Data Documentation

6.67.1.1 template<typename RealIntegerType , typename CompatibleNumberIntegerType > constexpr auto nlohmann::detail::is_compatible_integer_type< RealIntegerType, CompatibleNumberIntegerType >::value [static]

Initial value:

```
=
        is_compatible_integer_type_impl <
        std::is_integral<CompatibleNumberIntegerType>::value and
        not std::is_same<bool, CompatibleNumberIntegerType>::value,
        RealIntegerType, CompatibleNumberIntegerType > ::value
```

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.68 nlohmann::detail::is_compatible_integer_type_impl< bool, typename, typename > Struct Template Reference

Inheritance diagram for nlohmann::detail::is_compatible_integer_type_impl< bool, typename, typename >:

Collaboration diagram for nlohmann::detail::is_compatible_integer_type_impl< bool, typename, typename >:

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.69 nlohmann::detail::is_compatible_integer_type_impl< true, RealIntegerType, CompatibleNumberIntegerType > Struct Template Reference

Collaboration diagram for nlohmann::detail::is_compatible_integer_type_impl< true, RealIntegerType, Compatible↩ NumberIntegerType >:

Public Types

- using **RealLimits** = std::numeric_limits< RealIntegerType >
- using **CompatibleLimits** = std::numeric_limits< CompatibleNumberIntegerType >

Static Public Attributes

- static constexpr auto **value**

6.69.1 Member Data Documentation

6.69.1.1 template<typename RealIntegerType , typename CompatibleNumberIntegerType > constexpr auto nlohmann::detail::is_compatible_integer_type_impl< true, RealIntegerType, CompatibleNumberIntegerType >::value [static]

Initial value:

```
=
        std::is_constructible<RealIntegerType, CompatibleNumberIntegerType>::value and
        CompatibleLimits::is_integer and
        RealLimits::is_signed == CompatibleLimits::is_signed
```

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.70 nlohmann::detail::is_compatible_object_type< BasicJsonType, CompatibleObjectType > Struct Template Reference

Collaboration diagram for nlohmann::detail::is_compatible_object_type< BasicJsonType, CompatibleObjectType >:

Static Public Attributes

- static auto constexpr **value**

6.70.1 Member Data Documentation

6.70.1.1 template<class BasicJsonType , class CompatibleObjectType > auto constexpr nlohmann::detail::is_compatible_object_type< BasicJsonType, CompatibleObjectType >::value [static]

Initial value:

```
= is_compatible_object_type_impl <
                                conjunction<negation<std::is_same<void, CompatibleObjectType>>,
                                has_mapped_type<CompatibleObjectType>,
                                has_key_type<CompatibleObjectType>>::value,
                                typename BasicJsonType::object_t, CompatibleObjectType >::value
```

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.71 nlohmann::detail::is_compatible_object_type_impl< B, RealType, CompatibleObjectType > Struct Template Reference

Inheritance diagram for nlohmann::detail::is_compatible_object_type_impl< B, RealType, CompatibleObjectType >:

Collaboration diagram for nlohmann::detail::is_compatible_object_type_impl< B, RealType, CompatibleObjectType >:

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.72 nlohmann::detail::is_compatible_object_type_impl< true, RealType, CompatibleObjectType > Struct Template Reference

Collaboration diagram for nlohmann::detail::is_compatible_object_type_impl< true, RealType, CompatibleObjectType >:

Static Public Attributes

- static constexpr auto **value**

6.72.1 Member Data Documentation

6.72.1.1 template<class RealType , class CompatibleObjectType > constexpr auto nlohmann::detail::is_compatible_object_type_impl< true, RealType, CompatibleObjectType >::value [static]

Initial value:

```
=
        std::is_constructible<typename RealType::key_type, typename CompatibleObjectType::key_type>::value
      and
        std::is_constructible<typename RealType::mapped_type, typename
       CompatibleObjectType::mapped_type>::value
```

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.73 nlohmann::detail::iter_impl< BasicJsonType > Class Template Reference

a template for a bidirectional iterator for the basic_json class

`#include <json.hpp>`

Inheritance diagram for nlohmann::detail::iter_impl< BasicJsonType >:

Collaboration diagram for nlohmann::detail::iter_impl< BasicJsonType >:

Public Types

- using value_type = typename BasicJsonType::value_type

  the type of the values when the iterator is dereferenced
- using difference_type = typename BasicJsonType::difference_type

  a type to represent differences between iterators
- using pointer = typename std::conditional< std::is_const< BasicJsonType >::value, typename BasicJson↩ Type::const_pointer, typename BasicJsonType::pointer >::type

  defines a pointer to the type iterated over (value_type)
- using reference = typename std::conditional< std::is_const< BasicJsonType >::value, typename Basic↩ JsonType::const_reference, typename BasicJsonType::reference >::type

  defines a reference to the type iterated over (value_type)

Public Member Functions

- iter_impl ()=default
  default constructor
- iter_impl (pointer object) noexcept
  constructor for a given JSON instance
- iter_impl (const iter_impl< typename std::remove_const< BasicJsonType >::type > &other) noexcept
  converting constructor
- iter_impl & operator= (const iter_impl< typename std::remove_const< BasicJsonType >::type > &other) noexcept
  converting assignment
- reference operator* () const
  return a reference to the value pointed to by the iterator
- pointer operator-> () const
  dereference the iterator
- iter_impl operator++ (int)
  post-increment (it++)
- iter_impl & operator++ ()
  pre-increment (++it)
- iter_impl operator-- (int)
  post-decrement (it–)
- iter_impl & operator-- ()
  pre-decrement (–it)
- bool operator== (const iter_impl &other) const
  comparison: equal
- bool operator!= (const iter_impl &other) const
  comparison: not equal
- bool operator< (const iter_impl &other) const
  comparison: smaller
- bool operator<= (const iter_impl &other) const
  comparison: less than or equal
- bool operator> (const iter_impl &other) const
  comparison: greater than
- bool operator>= (const iter_impl &other) const
  comparison: greater than or equal
- iter_impl & operator+= (difference_type i)
  add to iterator
- iter_impl & operator-= (difference_type i)
  subtract from iterator
- iter_impl operator+ (difference_type i) const
  add to iterator
- iter_impl operator- (difference_type i) const
  subtract from iterator
- difference_type operator- (const iter_impl &other) const
  return difference
- reference operator[ ] (difference_type n) const
  access to successor
- object_t::key_type key () const
  return the key of an object iterator
- reference value () const
  return the value of an iterator

Friends

- iter_impl operator+ (difference_type i, const iter_impl &it)

  addition of distance and iterator

6.73.1 Detailed Description

template<typename BasicJsonType>
class nlohmann::detail::iter_impl< BasicJsonType >

a template for a bidirectional iterator for the basic_json class

This class implements a both iterators (iterator and const_iterator) for the basic_json class.

Note

An iterator is called *initialized* when a pointer to a JSON value has been set (e.g., by a constructor or a copy assignment). If the iterator is default-constructed, it is *uninitialized* and most methods are undefined. The library uses assertions to detect calls on uninitialized iterators.**

The class satisfies the following concept requirements:

- `BidirectionalIterator`: The iterator that can be moved can be moved in both directions (i.e. incremented and decremented).

Since

version 1.0.0, simplified in version 2.0.9, change to bidirectional iterators in version 3.0.0 (see https://github.com/nlohmann/json/issues/593)

6.73.2 Constructor & Destructor Documentation

6.73.2.1 template<typename BasicJsonType> iter_impl< typename std::conditional< std::is_const< BasicJsonType >::value, typename std::remove_const< BasicJsonType >::type, const BasicJsonType >::type > ( ) [default]

default constructor

allow basic_json to access private members

6.73.2.2 template<typename BasicJsonType> nlohmann::detail::iter_impl< BasicJsonType >::iter_impl (pointer *object*) [inline], [explicit], [noexcept]

constructor for a given JSON instance

Parameters

in	*object*	pointer to a JSON object for this iterator

Precondition

object != nullptr

Postcondition

The iterator is initialized; i.e. `m_object != nullptr`.

6.73.2.3 template<typename BasicJsonType> nlohmann::detail::iter_impl< BasicJsonType >::iter_impl (const iter_impl< typename std::remove_const< BasicJsonType >::type > & *other*) [inline], [noexcept]

converting constructor

Note

The conventional copy constructor and copy assignment are implicitly defined. Combined with the following converting constructor and assignment, they support: (1) copy from iterator to iterator, (2) copy from const iterator to const iterator, and (3) conversion from iterator to const iterator. However conversion from const iterator to iterator is not defined.

Parameters

in	*other*	non-const iterator to copy from

Note

It is not checked whether *other* is initialized.

6.73.3 Member Function Documentation

6.73.3.1 template<typename BasicJsonType> object_t::key_type nlohmann::detail::iter_impl< BasicJsonType >::key ( ) const [inline]

return the key of an object iterator

Precondition

The iterator is initialized; i.e. `m_object != nullptr`.

6.73.3.2 template<typename BasicJsonType> bool nlohmann::detail::iter_impl< BasicJsonType >::operator!= (const iter_impl< BasicJsonType > & *other*) const [inline]

comparison: not equal

Precondition

The iterator is initialized; i.e. `m_object != nullptr`.

6.73.3.3 template<typename BasicJsonType> reference nlohmann::detail::iter_impl< BasicJsonType >::operator* ( ) const [inline]

return a reference to the value pointed to by the iterator

Precondition

The iterator is initialized; i.e. `m_object != nullptr`.

6.73.3.4 template<typename BasicJsonType> iter_impl nlohmann::detail::iter_impl< BasicJsonType >::operator+ (difference_type *i*) const [inline]

add to iterator

Precondition

The iterator is initialized; i.e. `m_object != nullptr`.

6.73.3.5 template<typename BasicJsonType> iter_impl nlohmann::detail::iter_impl< BasicJsonType >::operator++ (int) [inline]

post-increment (it++)

Precondition

The iterator is initialized; i.e. `m_object != nullptr`.

6.73.3.6 template<typename BasicJsonType> iter_impl& nlohmann::detail::iter_impl< BasicJsonType >::operator++ ( ) [inline]

pre-increment (++it)

Precondition

The iterator is initialized; i.e. `m_object != nullptr`.

6.73.3.7 template<typename BasicJsonType> iter_impl& nlohmann::detail::iter_impl< BasicJsonType >::operator+= (difference_type *i*) [inline]

add to iterator

Precondition

The iterator is initialized; i.e. `m_object != nullptr`.

6.73.3.8 template<typename BasicJsonType> iter_impl nlohmann::detail::iter_impl< BasicJsonType >::operator- (difference_type *i*) const [inline]

subtract from iterator

Precondition

The iterator is initialized; i.e. `m_object != nullptr`.

6.73.3.9 template<typename BasicJsonType> difference_type nlohmann::detail::iter_impl< BasicJsonType >::operator- (const iter_impl< BasicJsonType > & *other*) const [inline]

return difference

Precondition

The iterator is initialized; i.e. `m_object != nullptr`.

6.73.3.10 template<typename BasicJsonType> iter_impl nlohmann::detail::iter_impl< BasicJsonType >::operator-- (int) [inline]

post-decrement (it–)

Precondition

The iterator is initialized; i.e. `m_object != nullptr`.

6.73.3.11 template<typename BasicJsonType> iter_impl& nlohmann::detail::iter_impl< BasicJsonType >::operator-- ( ) [inline]

pre-decrement (–it)

Precondition

The iterator is initialized; i.e. `m_object != nullptr`.

6.73.3.12 template<typename BasicJsonType> iter_impl& nlohmann::detail::iter_impl< BasicJsonType >::operator-= (difference_type *i*) [inline]

subtract from iterator

Precondition

The iterator is initialized; i.e. `m_object != nullptr`.

6.73.3.13 template<typename BasicJsonType> pointer nlohmann::detail::iter_impl< BasicJsonType >::operator-> ( ) const [inline]

dereference the iterator

Precondition

The iterator is initialized; i.e. `m_object != nullptr`.

6.73.3.14 template<typename BasicJsonType> bool nlohmann::detail::iter_impl< BasicJsonType >::operator< (const iter_impl< BasicJsonType > & *other*) const [inline]

comparison: smaller

Precondition

The iterator is initialized; i.e. `m_object != nullptr`.

6.73.3.15 template<typename BasicJsonType> bool nlohmann::detail::iter_impl< BasicJsonType >::operator<= (const iter_impl< BasicJsonType > & *other*) const [inline]

comparison: less than or equal

Precondition

The iterator is initialized; i.e. `m_object != nullptr`.

6.73.3.16 template<typename BasicJsonType> iter_impl& nlohmann::detail::iter_impl< BasicJsonType >::operator= (const iter_impl< typename std::remove_const< BasicJsonType >::type > & *other*) [inline], [noexcept]

converting assignment

Parameters

`in,out`	*other*	non-const iterator to copy from

Returns

const/non-const iterator

Note

It is not checked whether *other* is initialized.

6.73.3.17 template<typename BasicJsonType> bool nlohmann::detail::iter_impl< BasicJsonType >::operator== (const iter_impl< BasicJsonType > & *other*) const [inline]

comparison: equal

Precondition

The iterator is initialized; i.e. `m_object != nullptr`.

6.73.3.18 template<typename BasicJsonType> bool nlohmann::detail::iter_impl< BasicJsonType >::operator> (const iter_impl< BasicJsonType > & *other*) const [inline]

comparison: greater than

Precondition

The iterator is initialized; i.e. `m_object != nullptr`.

6.73.3.19 template<typename BasicJsonType> bool nlohmann::detail::iter_impl< BasicJsonType >::operator>= (const iter_impl< BasicJsonType > & *other*) const [inline]

comparison: greater than or equal

Precondition

The iterator is initialized; i.e. `m_object != nullptr`.

6.73.3.20 template<typename BasicJsonType> reference nlohmann::detail::iter_impl< BasicJsonType >::operator[] (difference_type *n*) const [inline]

access to successor

Precondition

The iterator is initialized; i.e. `m_object != nullptr`.

6.73.3.21 template<typename BasicJsonType> reference nlohmann::detail::iter_impl< BasicJsonType >::value ( ) const [inline]

return the value of an iterator

Precondition

The iterator is initialized; i.e. `m_object != nullptr`.

6.73.4 Friends And Related Function Documentation

6.73.4.1 template<typename BasicJsonType> iter_impl operator+ (difference_type *i*, const iter_impl< BasicJsonType > & *it*) [friend]

addition of distance and iterator

Precondition

The iterator is initialized; i.e. `m_object != nullptr`.

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.74 nlohmann::detail::iteration_proxy< IteratorType > Class Template Reference

proxy class for the iterator_wrapper functions

`#include <json.hpp>`

Public Member Functions

- iteration_proxy (typename IteratorType::reference cont)
  construct iteration proxy from a container
- iteration_proxy_internal begin () noexcept
  return iterator begin (needed for range-based for)
- iteration_proxy_internal end () noexcept
  return iterator end (needed for range-based for)

6.74.1 Detailed Description

template<typename IteratorType>
class nlohmann::detail::iteration_proxy< IteratorType >

proxy class for the iterator_wrapper functions

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.75 nlohmann::json_pointer Class Reference

JSON Pointer.

`#include <json.hpp>`

Public Member Functions

- json_pointer (const std::string &s="")
  create JSON pointer
- std::string to_string () const noexcept
  return a string representation of the JSON pointer
- operator std::string () const
  return a string representation of the JSON pointer

Friends

- class basic_json
  allow basic_json to access private members
- bool **operator==** (json_pointer const &lhs, json_pointer const &rhs) noexcept
- bool **operator!=** (json_pointer const &lhs, json_pointer const &rhs) noexcept

6.75.1 Detailed Description

JSON Pointer.

A JSON pointer defines a string syntax for identifying a specific value within a JSON document. It can be used with functions `at` and `operator[]`. Furthermore, JSON pointers are the base for JSON patches.

See also

RFC 6901

Since

version 2.0.0

6.75.2 Constructor & Destructor Documentation

6.75.2.1 nlohmann::json_pointer::json_pointer (const std::string & *s* = "") [inline], [explicit]

create JSON pointer

Create a JSON pointer according to the syntax described in Section 3 of RFC6901.

Parameters

in	*s*	string representing the JSON pointer; if omitted, the empty string is assumed which references the whole JSON value

Exceptions

parse_error.107	if the given JSON pointer *s* is nonempty and does not begin with a slash (/); see example below

Exceptions

parse_error.108	if a tilde (~) in the given JSON pointer *s* is not followed by `0` (representing ~) or `1` (representing `/`); see example below

{The example shows the construction several valid JSON pointers as well as the exceptional behavior.,json_pointer}

Since

version 2.0.0

6.75.3 Member Function Documentation

6.75.3.1 nlohmann::json_pointer::operator std::string ( ) const [inline]

return a string representation of the JSON pointer

Invariant

For each JSON pointer `ptr`, it holds:

```
ptr == json_pointer(ptr.to_string());
```

Returns

a string representation of the JSON pointer

{The example shows the result of `to_string`., json_pointer__to_string}

Since

version 2.0.0

6.75.3.2 std::string nlohmann::json_pointer::to_string ( ) const [inline], [noexcept]

return a string representation of the JSON pointer

Invariant

For each JSON pointer `ptr`, it holds:

```
ptr == json_pointer(ptr.to_string());
```

Returns

a string representation of the JSON pointer

{The example shows the result of `to_string`., json_pointer__to_string}

Since

version 2.0.0

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.76 nlohmann::detail::json_ref< BasicJsonType > Class Template Reference

Public Types

- using **value_type** = BasicJsonType

Public Member Functions

- **json_ref** (value_type &&value)
- **json_ref** (const value_type &value)
- **json_ref** (std::initializer_list< json_ref > init)
- template<class... Args>
  json_ref (Args &&...args)
- **json_ref** (json_ref &&)=default
- **json_ref** (const json_ref &)=delete
- json_ref & **operator=** (const json_ref &)=delete
- value_type **moved_or_copied** () const
- value_type const & **operator*** () const
- value_type const * **operator->** () const

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.77 nlohmann::detail::json_reverse_iterator< Base > Class Template Reference

a template for a reverse iterator class

`#include <json.hpp>`

Inheritance diagram for nlohmann::detail::json_reverse_iterator< Base >:

Collaboration diagram for nlohmann::detail::json_reverse_iterator< Base >:

Public Types

- using **difference_type** = std::ptrdiff_t
- using base_iterator = std::reverse_iterator< Base >

  shortcut to the reverse iterator adaptor
- using reference = typename Base::reference

  the reference type for the pointed-to element

Public Member Functions

- json_reverse_iterator (const typename base_iterator::iterator_type &it) noexcept
  create reverse iterator from iterator
- json_reverse_iterator (const base_iterator &it) noexcept
  create reverse iterator from base class
- json_reverse_iterator operator++ (int)
  post-increment (it++)
- json_reverse_iterator & operator++ ()
  pre-increment (++it)
- json_reverse_iterator operator-- (int)
  post-decrement (it–)
- json_reverse_iterator & operator-- ()
  pre-decrement (–it)
- json_reverse_iterator & operator+= (difference_type i)
  add to iterator
- json_reverse_iterator operator+ (difference_type i) const
  add to iterator
- json_reverse_iterator operator- (difference_type i) const
  subtract from iterator
- difference_type operator- (const json_reverse_iterator &other) const
  return difference
- reference operator[ ] (difference_type n) const
  access to successor
- auto key () const -> decltype(std::declval< Base >().key())
  return the key of an object iterator
- reference value () const
  return the value of an iterator

6.77.1 Detailed Description

template<typename Base>
class nlohmann::detail::json_reverse_iterator< Base >

a template for a reverse iterator class

Template Parameters

Base	the base iterator type to reverse. Valid types are iterator (to create reverse_iterator) and const_iterator (to create const_reverse_iterator).

The class satisfies the following concept requirements:

- `BidirectionalIterator`: The iterator that can be moved can be moved in both directions (i.e. incremented and decremented).
- `OutputIterator`: It is possible to write to the pointed-to element (only if *Base* is iterator).

Since

version 1.0.0

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.78 Lattice< T > Class Template Reference

Collaboration diagram for Lattice< T >:

Public Member Functions

- **Lattice** (const Lattice &other) noexcept
- **Lattice** (Lattice &&other) noexcept
- **Lattice** (std::array< int, 4 > sizeArray)
- void **allocate** (std::array< int, 4 > sizeArray)
- T & **at** (int x, int y, int z, int t)
- T & **at** (const std::vector< int > &site)
- T & **at** (const std::array< int, 4 > &site)
- T & **at** (int i)
- const T & **at** (int x, int y, int z, int t) const
- const T & **at** (const std::vector< int > &site) const
- const T & **at** (const std::array< int, 4 > &site) const
- const T & **at** (int i) const
- Lattice & **operator=** (const Lattice &other) noexcept
- Lattice & **operator=** (Lattice &&other) noexcept
- Lattice & **operator+=** (const Lattice &other) noexcept
- Lattice & **operator+=** (Lattice &&other) noexcept
- Lattice & **operator-=** (const Lattice &other) noexcept
- Lattice & **operator-=** (Lattice &&other) noexcept
- Lattice & **operator*=** (const Lattice &other) noexcept
- Lattice & **operator*=** (Lattice &&other) noexcept
- Lattice & **operator+=** (double scalar) noexcept
- Lattice & **operator-=** (double scalar) noexcept
- Lattice & **operator*=** (double scalar) noexcept

Public Attributes

- std::vector< T > **lattice**
- std::array< int, 4 > **size**
- int **sites**

Friends

- Lattice **operator+** (Lattice lhs, const Lattice &rhs) noexcept
- Lattice **operator+** (Lattice lhs, Lattice &&rhs) noexcept
- Lattice **operator-** (Lattice lhs, const Lattice &rhs) noexcept
- Lattice **operator-** (Lattice lhs, Lattice &&rhs) noexcept
- Lattice **operator*** (Lattice lhs, const Lattice &rhs) noexcept
- Lattice **operator*** (Lattice lhs, Lattice &&rhs) noexcept
- Lattice **operator+** (Lattice lhs, double scalar) noexcept
- Lattice **operator-** (Lattice lhs, double scalar) noexcept
- Lattice **operator*** (Lattice lhs, double scalar) noexcept

The documentation for this class was generated from the following files:

- /home/giovanni/Desktop/LatticeYangMills/include/Actions/action.h
- /home/giovanni/Desktop/LatticeYangMills/include/Math/lattice.h

6.79 LatticeUnits Struct Reference

Collaboration diagram for LatticeUnits:

Static Public Member Functions

- static void **initialize** (double beta)
- static double **plaquette** (double value)
- static double **energyDensity** (double value)
- static double **topologicalCharge** (double value)
- static double **calculateLatticeSpacing** (double beta)

Static Public Attributes

- static double **beta** = 0
- static double **latticeVolume** = 0
- static double **latticeSpacing** = 0
- static std::array< int, 4 > **size**
- static int **latticeSites** = 0
- static int **Nc** = 0

The documentation for this struct was generated from the following files:

- /home/giovanni/Desktop/LatticeYangMills/include/Utils/latticeunits.h
- /home/giovanni/Desktop/LatticeYangMills/src/Utils/latticeunits.cpp

6.80 left_shiftable Class Reference

Inheritance diagram for left_shiftable:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.81 std::less< ::nlohmann::detail::value_t > Struct Template Reference

`#include <json.hpp>`

Public Member Functions

- bool operator() (nlohmann::detail::value_t lhs, nlohmann::detail::value_t rhs) const noexcept

  compare two value_t enum values

6.81.1 Detailed Description

template<>
struct std::less< ::nlohmann::detail::value_t >

specialization for std::less<value_t>

Note

: do not remove the space after '<', see https://github.com/nlohmann/json/pull/679

6.81.2 Member Function Documentation

6.81.2.1 bool std::less< ::nlohmann::detail::value_t >::operator() (nlohmann::detail::value_t *lhs*, nlohmann::detail::value_t *rhs*) const [inline], [noexcept]

compare two value_t enum values

Since

version 3.0.0

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.82 tao::operators::less_than_comparable< T, U > Class Template Reference

Inheritance diagram for tao::operators::less_than_comparable< T, U >:

Friends

- bool **operator**<= (const T &lhs, const U &rhs) noexcept(noexcept(static_cast< bool >(lhs > rhs)))
- bool **operator**>= (const T &lhs, const U &rhs) noexcept(noexcept(static_cast< bool >(lhs< rhs)))
- bool **operator**< (const U &lhs, const T &rhs) noexcept(noexcept(static_cast< bool >(rhs > lhs)))
- bool **operator**> (const U &lhs, const T &rhs) noexcept(noexcept(static_cast< bool >(rhs< lhs)))
- bool **operator**<= (const U &lhs, const T &rhs) noexcept(noexcept(static_cast< bool >(rhs >=lhs)))
- bool **operator**>= (const U &lhs, const T &rhs) noexcept(noexcept(static_cast< bool >(rhs<=lhs)))

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.83 tao::operators::less_than_comparable< T > Class Template Reference

Friends

- bool **operator**> (const T &lhs, const T &rhs) noexcept(noexcept(static_cast< bool >(rhs< lhs)))
- bool **operator**<= (const T &lhs, const T &rhs) noexcept(noexcept(static_cast< bool >(rhs< lhs)))
- bool **operator**>= (const T &lhs, const T &rhs) noexcept(noexcept(static_cast< bool >(lhs< rhs)))

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.84 nlohmann::detail::lexer< BasicJsonType > Class Template Reference

lexical analysis

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

Public Types

- enum token_type {
  token_type::uninitialized, token_type::literal_true, token_type::literal_false, token_type::literal_null,
  token_type::value_string, token_type::value_unsigned, token_type::value_integer, token_type::value_float,
  token_type::begin_array, token_type::begin_object, token_type::end_array, token_type::end_object,
  token_type::name_separator, token_type::value_separator, token_type::parse_error, token_type::end_of_↩
  input,
  token_type::literal_or_value }

  token types for the parser

Public Member Functions

- **lexer** (detail::input_adapter_t adapter)
- **lexer** (const lexer &)=delete
- lexer & **operator=** (lexer &)=delete
- constexpr number_integer_t get_number_integer () const noexcept

  return integer value
- constexpr number_unsigned_t get_number_unsigned () const noexcept

  return unsigned integer value
- constexpr number_float_t get_number_float () const noexcept

  return floating-point value
- std::string move_string ()

  return current string value (implicitly resets the token; useful only once)
- constexpr std::size_t get_position () const noexcept

  return position of last read token
- std::string get_token_string () const
- constexpr const char * get_error_message () const noexcept

  return syntax error message
- token_type **scan** ()

Static Public Member Functions

- static const char * token_type_name (const token_type t) noexcept

  return name of values of type token_type (only used for errors)

6.84.1 Detailed Description

template<typename BasicJsonType>
class nlohmann::detail::lexer< BasicJsonType >

lexical analysis

This class organizes the lexical analysis during JSON deserialization.

6.84.2 Member Enumeration Documentation

6.84.2.1 template<typename BasicJsonType > enum nlohmann::detail::lexer::token_type [strong]

token types for the parser

Enumerator

uninitialized indicating the scanner is uninitialized

literal_true the `true` literal

literal_false the `false` literal

literal_null the `null` literal

value_string a string – use get_string() for actual value

value_unsigned an unsigned integer – use get_number_unsigned() for actual value

value_integer a signed integer – use get_number_integer() for actual value

value_float an floating point number – use get_number_float() for actual value

begin_array the character for array begin `[`

begin_object the character for object begin `{`

end_array the character for array end `]`

end_object the character for object end `}`

name_separator the name separator `:`

value_separator the value separator `,`

parse_error indicating a parse error

end_of_input indicating the end of the input buffer

literal_or_value a literal or the begin of a value (only for diagnostics)

6.84.3 Member Function Documentation

6.84.3.1 template<typename BasicJsonType > std::string nlohmann::detail::lexer< BasicJsonType >::get_token_string ( ) const [inline]

return the last read token (for errors only). Will never contain EOF (an arbitrary value that is not a valid char value, often -1), because 255 may legitimately occur. May contain NUL, which should be escaped.

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.85 nlohmann::detail::make_index_sequence< N > Struct Template Reference

Inheritance diagram for nlohmann::detail::make_index_sequence< N >:

Collaboration diagram for nlohmann::detail::make_index_sequence< N >:

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.86 nlohmann::detail::make_index_sequence< 0 > Struct Template Reference

Inheritance diagram for nlohmann::detail::make_index_sequence< 0 >:

Collaboration diagram for nlohmann::detail::make_index_sequence< 0 >:

Additional Inherited Members

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.87 nlohmann::detail::make_index_sequence< 1 > Struct Template Reference

Inheritance diagram for nlohmann::detail::make_index_sequence< 1 >:

Collaboration diagram for nlohmann::detail::make_index_sequence< 1 >:

Additional Inherited Members

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.88 nlohmann::detail::merge_and_renumber< Sequence1, Sequence2 > Struct Template Reference

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.89 nlohmann::detail::merge_and_renumber< index_sequence< I1... >, index_sequence< I2... > > Struct Template Reference

Inheritance diagram for nlohmann::detail::merge_and_renumber< index_sequence< I1... >, index_sequence< I2... > >:

Collaboration diagram for nlohmann::detail::merge_and_renumber< index_sequence< I1... >, index_sequence< I2... > >:

Additional Inherited Members

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.90 multipliable Class Reference

Inheritance diagram for multipliable:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.91 multipliable Class Reference

Inheritance diagram for multipliable:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/su3.h

6.92 nlohmann::detail::negation< B > Struct Template Reference

Inheritance diagram for nlohmann::detail::negation< B >:

Collaboration diagram for nlohmann::detail::negation< B >:

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.93 Observable Class Reference

Inheritance diagram for Observable:

Collaboration diagram for Observable:

Public Member Functions

- virtual void **compute** ()=0
- virtual void **initObservable** (GluonField *field)=0
- const char * **getName** ()
- double **value** ()

Public Attributes

- double **plaq**
- double **energy**
- double **topc**

Protected Member Functions

- void **gatherResults** ()

Protected Attributes

- GluonField * **m_field** = nullptr
- double **m_value**
- std::string **m_name**

The documentation for this class was generated from the following files:

- /home/giovanni/Desktop/LatticeYangMills/include/Observables/observable.h
- /home/giovanni/Desktop/LatticeYangMills/src/Observables/observable.cpp

6.94 orable Class Reference

Inheritance diagram for orable:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.95 orable_left Class Reference

Inheritance diagram for orable_left:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.96 tao::operators::ordered_commutative_ring< T, U > Class Template Reference

Inheritance diagram for tao::operators::ordered_commutative_ring< T, U >:

Collaboration diagram for tao::operators::ordered_commutative_ring< T, U >:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.97 tao::operators::ordered_field< T, U > Class Template Reference

Inheritance diagram for tao::operators::ordered_field< T, U >:

Collaboration diagram for tao::operators::ordered_field< T, U >:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.98 tao::operators::ordered_ring< T, U > Class Template Reference

Inheritance diagram for tao::operators::ordered_ring< T, U >:

Collaboration diagram for tao::operators::ordered_ring< T, U >:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.99 nlohmann::detail::other_error Class Reference

exception indicating other library errors

`#include <json.hpp>`

Inheritance diagram for nlohmann::detail::other_error:

Collaboration diagram for nlohmann::detail::other_error:

Static Public Member Functions

- static other_error **create** (int id_, const std::string &what_arg)

Additional Inherited Members

6.99.1 Detailed Description

exception indicating other library errors

This exception is thrown in case of errors that cannot be classified with the other exception types.

Exceptions have ids 5xx.

name / id	example message	description
json.exception.other_error.501	unsuccessful: {"op":"test","path"↩ :"/baz", "value":"bar"}	A JSON Patch operation 'test' failed. The unsuccessful operation is also printed.
json.exception.other_error.502	invalid object size for conversion	Some conversions to user-defined types impose constraints on the object size (e.g. std::pair)

See also

exception for the base class of the library exceptions
parse_error for exceptions indicating a parse error
invalid_iterator for exceptions indicating errors with iterators

type_error for exceptions indicating executing a member function with a wrong type
out_of_range for exceptions indicating access out of the defined range

{The following code shows how an other_error exception can be caught.,other_error}

Since

version 3.0.0

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.100 nlohmann::detail::out_of_range Class Reference

exception indicating access out of the defined range

`#include <json.hpp>`

Inheritance diagram for nlohmann::detail::out_of_range:

Collaboration diagram for nlohmann::detail::out_of_range:

Static Public Member Functions

- static out_of_range **create** (int id_, const std::string &what_arg)

Additional Inherited Members

6.100.1 Detailed Description

exception indicating access out of the defined range

This exception is thrown in case a library function is called on an input parameter that exceeds the expected range, for instance in case of array indices or nonexisting object keys.

Exceptions have ids 4xx.

name / id	example message	description
json.exception.out_of_range.401	array index 3 is out of range	The provided array index *i* is larger than *size-1*.
json.exception.out_of_range.402	array index '-' (3) is out of range	The special array index - in a JSON Pointer never describes a valid element of the array, but the index past the end. That is, it can only be used to add elements at this position, but not to read it.
json.exception.out_of_range.403	key 'foo' not found	The provided key was not found in the JSON object.
json.exception.out_of_range.404	unresolved reference token 'foo'	A reference token in a JSON Pointer could not be resolved.
json.exception.out_of_range.405	JSON pointer has no parent	The JSON Patch operations 'remove' and 'add' can not be applied

{The following code shows how an out_of_range exception can be caught.,out_of_range}

See also

exception for the base class of the library exceptions
parse_error for exceptions indicating a parse error
invalid_iterator for exceptions indicating errors with iterators
type_error for exceptions indicating executing a member function with a wrong type
other_error for exceptions indicating other library errors

Since

version 3.0.0

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.101 nlohmann::detail::output_adapter< CharType > Class Template Reference

Public Member Functions

- **output_adapter** (std::vector< CharType > &vec)
- **output_adapter** (std::basic_ostream< CharType > &s)
- **output_adapter** (std::basic_string< CharType > &s)
- **operator output_adapter_t< CharType >** ()

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.102 nlohmann::detail::output_adapter_protocol< CharType > Struct Template Reference

abstract output adapter interface

`#include <json.hpp>`

Inheritance diagram for nlohmann::detail::output_adapter_protocol< CharType >:

Public Member Functions

- virtual void **write_character** (CharType c)=0
- virtual void **write_characters** (const CharType *s, std::size_t length)=0

6.102.1 Detailed Description

template<typename CharType>
struct nlohmann::detail::output_adapter_protocol< CharType >

abstract output adapter interface

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.103 nlohmann::detail::output_stream_adapter< CharType > Class Template Reference

output adapter for output streams

`#include <json.hpp>`

Inheritance diagram for nlohmann::detail::output_stream_adapter< CharType >:

Collaboration diagram for nlohmann::detail::output_stream_adapter< CharType >:

Public Member Functions

- **output_stream_adapter** (std::basic_ostream< CharType > &s)
- void **write_character** (CharType c) override
- void **write_characters** (const CharType *s, std::size_t length) override

6.103.1 Detailed Description

template<typename CharType>
class nlohmann::detail::output_stream_adapter< CharType >

output adapter for output streams

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.104 nlohmann::detail::output_string_adapter< CharType > Class Template Reference

output adapter for basic_string

`#include <json.hpp>`

Inheritance diagram for nlohmann::detail::output_string_adapter< CharType >:

Collaboration diagram for nlohmann::detail::output_string_adapter< CharType >:

Public Member Functions

- **output_string_adapter** (std::basic_string< CharType > &s)
- void **write_character** (CharType c) override
- void **write_characters** (const CharType *s, std::size_t length) override

6.104.1 Detailed Description

template<typename CharType>
class nlohmann::detail::output_string_adapter< CharType >

output adapter for basic_string

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.105 nlohmann::detail::output_vector_adapter< CharType > Class Template Reference

output adapter for byte vectors

`#include <json.hpp>`

Inheritance diagram for nlohmann::detail::output_vector_adapter< CharType >:

Collaboration diagram for nlohmann::detail::output_vector_adapter< CharType >:

Public Member Functions

- **output_vector_adapter** (std::vector< CharType > &vec)
- void **write_character** (CharType c) override
- void **write_characters** (const CharType *s, std::size_t length) override

6.105.1 Detailed Description

template<typename CharType>
class nlohmann::detail::output_vector_adapter< CharType >

output adapter for byte vectors

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.106 LatticeIO::OutputConf Class Reference

Static Public Member Functions

- static void **writeConf** (GluonField &lattice, int confNum)
- static void **writeSubLattice** (GluonField &lattice, int confNum)
- static void **setOutputDir** (std::string outputDir)

The documentation for this class was generated from the following files:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/outputconf.h
- /home/giovanni/Desktop/LatticeYangMills/src/InputOutput/outputconf.cpp

6.107 LatticeIO::OutputObs Class Reference

Static Public Member Functions

- static void **initialize** (std::vector< Observable * > &obsList)
- static void **writeObs** (std::vector< Observable * > &obsList, int MCSteps)
- static void **writeFlowObs** (int confNum, std::vector< Observable * > &obsList, std::vector< std::vector< double >> &obsMatrix)
- static void **setOutputDir** (std::string outputDir)

The documentation for this class was generated from the following files:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/outputobs.h
- /home/giovanni/Desktop/LatticeYangMills/src/InputOutput/outputobs.cpp

6.108 LatticeIO::OutputTerm Class Reference

Static Public Member Functions

- static void **printInitialConditions** ()
- static void **writeObs** (int confNum, std::vector< Observable * > &obsList)
- static void **printThermStep** (int step, std::vector< Observable * > &obsList, double acceptRatio)
- static void **printGenStep** (int confNum, std::vector< Observable * > &obsList, double acceptRatio)
- static void **writeFlowObs** (double flowTime, std::vector< Observable * > &obsList)

The documentation for this class was generated from the following files:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/outputterm.h
- /home/giovanni/Desktop/LatticeYangMills/src/InputOutput/outputterm.cpp

6.109 Parallel Class Reference

Static Public Member Functions

- static void **initialize** ()
- static void **createGeometry** (std::array< int, 4 > latticeSize, std::array< int, 4 > subLatticeSize)
- static void **finalize** ()
- static int **rank** ()
- static int **numProcs** ()
- static int **activeProcs** ()
- static bool **isActive** ()
- static MPI_Comm **cartCoordComm** ()
- static std::array< int, 4 > & **subBlocks** ()
- static std::array< int, 4 > & **rankCoord** ()
- static std::array< int, 4 > & **latticeSubSize** ()
- static std::array< int, 4 > & **latticeFullSize** ()
- static std::array< int, 4 > & **parity** ()
- static int **getNeighbor** (int direction, int sign)
- static int **getSecondNeighbor** (int direction1, int sign1, int direction2, int sign2)
- static void **openFile** (MPI_File &file, const char *fileName)
- static void **closeFile** (MPI_File &file)

The documentation for this class was generated from the following files:

- /home/giovanni/Desktop/LatticeYangMills/include/ParallelTools/parallel.h
- /home/giovanni/Desktop/LatticeYangMills/src/ParallelTools/parallel.cpp

6.110 nlohmann::detail::parse_error Class Reference

exception indicating a parse error

`#include <json.hpp>`

Inheritance diagram for nlohmann::detail::parse_error:

Collaboration diagram for nlohmann::detail::parse_error:

Static Public Member Functions

- static parse_error create (int id_, std::size_t byte_, const std::string &what_arg)
  create a parse error exception

Public Attributes

- const std::size_t byte
  byte index of the parse error

Additional Inherited Members

6.110.1 Detailed Description

exception indicating a parse error

This excpetion is thrown by the library when a parse error occurs. Parse errors can occur during the deserialization of JSON text, CBOR, MessagePack, as well as when using JSON Patch.

Member *byte* holds the byte index of the last read character in the input file.

Exceptions have ids 1xx.

name / id	example message	description
json.exception.parse_error.101	parse error at 2: unexpected end of input; expected string literal	This error indicates a syntax error while deserializing a JSON text. The error message describes that an unexpected token (character) was encountered, and the member *byte* indicates the error position.
json.exception.parse_error.102	parse error at 14: missing or wrong low surrogate	JSON uses the `\uxxxx` format to describe Unicode characters. Code points above above 0xFFFF are split into two `\uxxxx` entries ("surrogate pairs"). This error indicates that the surrogate pair is incomplete or contains an invalid code point.
json.exception.parse_error.103	parse error: code points above 0x10FFFF are invalid	Unicode supports code points up to 0x10FFFF. Code points above 0x10FFFF are invalid.
json.exception.parse_error.104	parse error: JSON patch must be an array of objects	RFC 6902 requires a JSON Patch document to be a JSON document that represents an array of objects.
json.exception.parse_error.105	parse error: operation must have string member 'op'	An operation of a JSON Patch document must contain exactly one "op" member, whose value indicates the operation to perform. Its value must be one of "add", "remove", "replace", "move", "copy", or "test"; other values are errors.
json.exception.parse_error.106	parse error: array index '01' must not begin with '0'	An array index in a JSON Pointer (RFC 6901) may be `0` or any number wihtout a leading `0`.
json.exception.parse_error.107	parse error: JSON pointer must be empty or begin with '/' - was: 'foo'	A JSON Pointer must be a Unicode string containing a sequence of zero or more reference tokens, each prefixed by a `/` character.
json.exception.parse_error.108	parse error: escape character '~' must be followed with '0' or '1'	In a JSON Pointer, only `~0` and `~1` are valid escape sequences.
json.exception.parse_error.109	parse error: array index 'one' is not a number	A JSON Pointer array index must be a number.
json.exception.parse_error.110	parse error at 1: cannot read 2 bytes from vector	When parsing CBOR or MessagePack, the byte vector ends before the complete value has been read.
json.exception.parse_error.112	parse error at 1: error reading CBOR; last byte: 0xf8	Not all types of CBOR or MessagePack are supported. This exception occurs if an unsupported byte was read.
json.exception.parse_error.113	parse error at 2: expected a CBOR string; last byte: 0x98	While parsing a map key, a value that is not a string has been read.

Note

For an input with n bytes, 1 is the index of the first character and n+1 is the index of the terminating null byte or the end of file. This also holds true when reading a byte vector (CBOR or MessagePack).

{The following code shows how a `parse_error` exception can be caught.,parse_error}

See also

exception for the base class of the library exceptions
invalid_iterator for exceptions indicating errors with iterators
type_error for exceptions indicating executing a member function with a wrong type
out_of_range for exceptions indicating access out of the defined range
other_error for exceptions indicating other library errors

Since

version 3.0.0

6.110.2 Member Function Documentation

6.110.2.1 static parse_error nlohmann::detail::parse_error::create (int *id_*, std::size_t *byte_*, const std::string & *what_arg*) [inline], [static]

create a parse error exception

Parameters

in	*id_*	the id of the exception
in	*byte_*	the byte index where the error occurred (or 0 if the position cannot be determined)
in	*what_arg*	the explanatory string

Returns

parse_error object

6.110.3 Member Data Documentation

6.110.3.1 const std::size_t nlohmann::detail::parse_error::byte

byte index of the parse error

The byte index of the last read character in the input file.

Note

For an input with n bytes, 1 is the index of the first character and n+1 is the index of the terminating null byte or the end of file. This also holds true when reading a byte vector (CBOR or MessagePack).

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.111 nlohmann::detail::parser< BasicJsonType > Class Template Reference

syntax analysis

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

Public Types

- enum parse_event_t : uint8_t {
  parse_event_t::object_start, parse_event_t::object_end, parse_event_t::array_start, parse_event_t::array_↩
  end,
  parse_event_t::key, parse_event_t::value }
- using **parser_callback_t** = std::function< bool(int depth, parse_event_t event, BasicJsonType &parsed)>

Public Member Functions

- parser (detail::input_adapter_t adapter, const parser_callback_t cb=nullptr, const bool allow_exceptions_↩
  =true)

  a parser reading from an input adapter
- void parse (const bool strict, BasicJsonType &result)

  public parser interface
- bool accept (const bool strict=true)

  public accept interface

6.111.1 Detailed Description

template<typename BasicJsonType>
class nlohmann::detail::parser< BasicJsonType >

syntax analysis

This class implements a recursive decent parser.

6.111.2 Member Enumeration Documentation

6.111.2.1 template<typename BasicJsonType > enum nlohmann::detail::parser::parse_event_t : uint8_t [strong]

Enumerator

object_start the parser read { and started to process a JSON object

object_end the parser read } and finished processing a JSON object

array_start the parser read [and started to process a JSON array

array_end the parser read] and finished processing a JSON array

key the parser read a key of a value in an object

value the parser finished reading a JSON value

6.111.3 Member Function Documentation

6.111.3.1 template<typename BasicJsonType > bool nlohmann::detail::parser< BasicJsonType >::accept (const bool *strict* = true) [inline]

public accept interface

Parameters

in	*strict*	whether to expect the last token to be EOF

Returns

whether the input is a proper JSON text

6.111.3.2 template<typename BasicJsonType > void nlohmann::detail::parser< BasicJsonType >::parse (const bool *strict*, BasicJsonType & *result*) [inline]

public parser interface

Parameters

in	*strict*	whether to expect the last token to be EOF
in,out	*result*	parsed JSON value

Exceptions

parse_error.101	in case of an unexpected token
parse_error.102	if to_unicode fails or surrogate error
parse_error.103	if to_unicode fails

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.112 tao::operators::partially_ordered< T, U > Class Template Reference

Friends

- bool **operator<=** (const T &lhs, const U &rhs) noexcept(noexcept(static_cast< bool >(lhs< rhs), static_↩ cast< bool >(lhs==rhs)))
- bool **operator>=** (const T &lhs, const U &rhs) noexcept(noexcept(static_cast< bool >(lhs > rhs), static_↩ cast< bool >(lhs==rhs)))
- bool **operator<** (const U &lhs, const T &rhs) noexcept(noexcept(static_cast< bool >(rhs > lhs)))
- bool **operator>** (const U &lhs, const T &rhs) noexcept(noexcept(static_cast< bool >(rhs< lhs)))
- bool **operator<=** (const U &lhs, const T &rhs) noexcept(noexcept(static_cast< bool >(rhs >=lhs)))
- bool **operator>=** (const U &lhs, const T &rhs) noexcept(noexcept(static_cast< bool >(rhs<=lhs)))

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.113 tao::operators::partially_ordered< T > Class Template Reference

Friends

- bool **operator>** (const T &lhs, const T &rhs) noexcept(noexcept(static_cast< bool >(rhs< lhs)))
- bool **operator<=** (const T &lhs, const T &rhs) noexcept(noexcept(static_cast< bool >(lhs< rhs), static_↩ cast< bool >(lhs==rhs)))
- bool **operator>=** (const T &lhs, const T &rhs) noexcept(noexcept(static_cast< bool >(rhs< lhs), static_↩ cast< bool >(lhs==rhs)))

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.114 Plaquette Class Reference

Inheritance diagram for Plaquette:

Collaboration diagram for Plaquette:

Public Member Functions

- void **initObservable** (Lattice *lattice)
- void **compute** ()

Additional Inherited Members

The documentation for this class was generated from the following files:

- /home/giovanni/Desktop/LatticeYangMills/include/Observables/plaquette.h
- /home/giovanni/Desktop/LatticeYangMills/src/Observables/plaquette.cpp

6.115 Point Class Reference

Collaboration diagram for Point:

Public Member Functions

- SU3 & **operator[ ]** (int i)
- SU3 **operator[ ]** (int i) const

Public Attributes

- SU3 **m_links** [4]

The documentation for this class was generated from the following files:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/point.h
- /home/giovanni/Desktop/LatticeYangMills/src/Math/point.cpp

6.116 nlohmann::detail::primitive_iterator_t Class Reference

an iterator for primitive JSON types

`#include <json.hpp>`

Public Types

- using **difference_type** = std::ptrdiff_t

Public Member Functions

- constexpr difference_type **get_value** () const noexcept
- void set_begin () noexcept

  set iterator to a defined beginning
- void set_end () noexcept

  set iterator to a defined past the end
- constexpr bool is_begin () const noexcept

  return whether the iterator can be dereferenced
- constexpr bool is_end () const noexcept

  return whether the iterator is at end
- primitive_iterator_t **operator+** (difference_type i)
- primitive_iterator_t & **operator++** ()
- primitive_iterator_t **operator++** (int)
- primitive_iterator_t & **operator--** ()
- primitive_iterator_t **operator--** (int)
- primitive_iterator_t & **operator+=** (difference_type n)
- primitive_iterator_t & **operator-=** (difference_type n)

Friends

- constexpr bool **operator==** (primitive_iterator_t lhs, primitive_iterator_t rhs) noexcept
- constexpr bool **operator**< (primitive_iterator_t lhs, primitive_iterator_t rhs) noexcept
- constexpr difference_type **operator-** (primitive_iterator_t lhs, primitive_iterator_t rhs) noexcept
- std::ostream & **operator**<< (std::ostream &os, primitive_iterator_t it)

6.116.1 Detailed Description

an iterator for primitive JSON types

This class models an iterator for primitive JSON types (boolean, number, string). It's only purpose is to allow the iterator/const_iterator classes to "iterate" over primitive values. Internally, the iterator is modeled by a `difference_type` variable. Value begin_value (`0`) models the begin, end_value (`1`) models past the end.

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.117 nlohmann::detail::priority_tag< N > Struct Template Reference

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.118 nlohmann::detail::priority_tag< 0 > Struct Template Reference

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.119 PureGauge Class Reference

Inheritance diagram for PureGauge:

Collaboration diagram for PureGauge:

Public Member Functions

- **PureGauge** (GluonField *lattice, double beta)
- PureGauge (double beta)

  Initializer for the PureGauge Action class.
- double **compute** (int x, int y, int z, int t, int mu, SU3 &newLink)
- void computeStaples (int mu)

  Computes the staples along the given directionfor all links in the given direction.
- Lattice< SU3 > computeDerivative (int mu)

  Computes the derivative of all links along the given direction.
- void **computeStaplez** (GluonField *lattice)
- void **computeOtherStaples** (int x, int y, int z, int t, int mu)

Additional Inherited Members

6.119.1 Constructor & Destructor Documentation

6.119.1.1 PureGauge::PureGauge (double *beta*)

Initializer for the PureGauge Action class.

Parameters

beta	The $/beta$ value of the action

6.119.2 Member Function Documentation

6.119.2.1 Lattice< SU3 > PureGauge::computeDerivative (int *mu*) [virtual]

Computes the derivative of all links along the given direction.

Parameters

mu	The index of the directions to compute the staples of

Returns

m_omega Lattice<SU3> containing the derivative of the GluonField

Implements Action.

6.119.2.2 void PureGauge::computeStaples (int *mu*) [virtual]

Computes the staples along the given directionfor all links in the given direction.

Parameters

mu	The index of the directions to compute the staples of

Implements Action.

The documentation for this class was generated from the following files:

- /home/giovanni/Desktop/LatticeYangMills/include/Actions/puregauge.h
- /home/giovanni/Desktop/LatticeYangMills/src/Actions/puregauge.cpp

6.120 Random Class Reference

Static Public Member Functions

- static double **randUniform** ()
- static SU3 **randSU3** ()
- static SU3 **randSU3Transf** (double epsilon)

The documentation for this class was generated from the following files:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/random.h
- /home/giovanni/Desktop/LatticeYangMills/src/Math/random.cpp

6.121 right_shiftable Class Reference

Inheritance diagram for right_shiftable:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.122 tao::operators::ring< T, U > Class Template Reference

Inheritance diagram for tao::operators::ring< T, U >:

Collaboration diagram for tao::operators::ring< T, U >:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.123 tao::operators::ring< T > Class Template Reference

Inheritance diagram for tao::operators::ring< T >:

Collaboration diagram for tao::operators::ring< T >:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.124 nlohmann::detail::serializer< BasicJsonType > Class Template Reference

Public Member Functions

- serializer (output_adapter_t< char > s, const char ichar)
- **serializer** (const serializer &)=delete
- serializer & **operator=** (const serializer &)=delete
- void dump (const BasicJsonType &val, const bool pretty_print, const bool ensure_ascii, const unsigned int indent_step, const unsigned int current_indent=0)

  internal implementation of the serialization function

6.124.1 Constructor & Destructor Documentation

6.124.1.1 template<typename BasicJsonType > nlohmann::detail::serializer< BasicJsonType >::serializer (output_adapter_t< char > *s,* const char *ichar*) [inline]

Parameters

in	*s*	output stream to serialize to
in	*ichar*	indentation character to use

6.124.2 Member Function Documentation

6.124.2.1 template<typename BasicJsonType > void nlohmann::detail::serializer< BasicJsonType >::dump (const BasicJsonType & *val,* const bool *pretty_print,* const bool *ensure_ascii,* const unsigned int *indent_step,* const unsigned int *current_indent* = `0`) `[inline]`

internal implementation of the serialization function

This function is called by the public member function dump and organizes the serialization internally. The indentation level is propagated as additional parameter. In case of arrays and objects, the function is called recursively.

- strings and object keys are escaped using `escape_string()`
- integer numbers are converted implicitly via `operator<<`
- floating-point numbers are converted to a string using `"%g"` format

Parameters

in	*val*	value to serialize
in	*pretty_print*	whether the output shall be pretty-printed
in	*indent_step*	the indent level
in	*current_indent*	the current indent level (only used internally)

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.125 tao::operators::shiftable< T, U > Class Template Reference

Inheritance diagram for tao::operators::shiftable< T, U >:

Collaboration diagram for tao::operators::shiftable< T, U >:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.126 nlohmann::detail::static_const< T > Struct Template Reference

Collaboration diagram for nlohmann::detail::static_const< T >:

Static Public Attributes

- static constexpr T **value** {}

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.127 SU3 Struct Reference

Inheritance diagram for SU3:

Collaboration diagram for SU3:

Public Member Functions

- **SU3** (double value) noexcept
- **SU3** (const SU3 &source) noexcept
- **SU3** (SU3 &&source) noexcept
- SU3 & **operator=** (const SU3 &other) noexcept
- SU3 & **operator=** (SU3 &&other) noexcept
- SU3 & **operator+=** (const SU3 &other) noexcept
- SU3 & **operator+=** (SU3 &&other) noexcept
- SU3 & **operator-=** (const SU3 &other) noexcept
- SU3 & **operator-=** (SU3 &&other) noexcept
- SU3 & **operator*=** (const SU3 &other) noexcept
- SU3 & **operator*=** (SU3 &&other) noexcept
- SU3 & **operator+=** (const double scalar) noexcept
- SU3 & **operator-=** (const double scalar) noexcept
- SU3 & **operator*=** (const double scalar) noexcept
- void **setSU3Identity** ()
- void **setSU3Zero** ()
- void **setSU3Random** ()
- double **realTrace** ()
- double **imagTrace** ()
- SU3 **exp** ()
- void **printSU3** ()

Public Attributes

- std::array< double, 18 > **mat**

The documentation for this struct was generated from the following files:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/su3.h
- /home/giovanni/Desktop/LatticeYangMills/src/Math/su3.cpp

6.128 subtractable Class Reference

Inheritance diagram for subtractable:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/su3.h

6.129 subtractable Class Reference

Inheritance diagram for subtractable:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.130 subtractable_left Class Reference

Inheritance diagram for subtractable_left:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.131 subtractable_left Class Reference

Inheritance diagram for subtractable_left:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/su3.h

6.132 SuperObs Class Reference

Inheritance diagram for SuperObs:

Collaboration diagram for SuperObs:

Public Member Functions

- void **initObservable** (GluonField *field)
- void **compute** ()

Additional Inherited Members

The documentation for this class was generated from the following files:

- /home/giovanni/Desktop/LatticeYangMills/include/Observables/superobs.h
- /home/giovanni/Desktop/LatticeYangMills/src/Observables/superobs.cpp

6.133 nlohmann::detail::to_json_fn Struct Reference

Public Member Functions

- template<typename BasicJsonType , typename T >
  void **operator()** (BasicJsonType &j, T &&val) const noexcept(noexcept(std::declval< to_json_fn >().call(j, std::forward< T >(val), priority_tag< 1 >{})))

The documentation for this struct was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.134 TopologicalCharge Class Reference

Inheritance diagram for TopologicalCharge:

Collaboration diagram for TopologicalCharge:

Public Member Functions

- void **initObservable** (Lattice *lattice)
- void **compute** ()

Additional Inherited Members

The documentation for this class was generated from the following files:

- /home/giovanni/Desktop/LatticeYangMills/include/Observables/topologicalcharge.h
- /home/giovanni/Desktop/LatticeYangMills/src/Observables/topologicalcharge.cpp

6.135 tao::operators::totally_ordered< T, U > Class Template Reference

Inheritance diagram for tao::operators::totally_ordered< T, U >:

Collaboration diagram for tao::operators::totally_ordered< T, U >:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.136 type Class Reference

Inheritance diagram for type:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.137 nlohmann::detail::type_error Class Reference

exception indicating executing a member function with a wrong type

`#include <json.hpp>`

Inheritance diagram for nlohmann::detail::type_error:

Collaboration diagram for nlohmann::detail::type_error:

Static Public Member Functions

- static type_error **create** (int id_, const std::string &what_arg)

Additional Inherited Members

6.137.1 Detailed Description

exception indicating executing a member function with a wrong type

This exception is thrown in case of a type error; that is, a library function is executed on a JSON value whose type does not match the expected semantics.

Exceptions have ids 3xx.

name / id	example message	description
json.exception.type_error.301	cannot create object from initializer list	To create an object from an initializer list, the initializer list must consist only of a list of pairs whose first element is a string. When this constraint is violated, an array is created instead.
json.exception.type_error.302	type must be object, but is array	During implicit or explicit value conversion, the JSON type must be compatible to the target type. For instance, a JSON string can only be converted into string types, but not into numbers or boolean types.
json.exception.type_error.303	incompatible ReferenceType for get↩ _ref, actual type is object	To retrieve a reference to a value stored in a basic_json object with get_ref, the type of the reference must match the value type. For instance, for a JSON array, the *ReferenceType* must be array_t&.

name / id	example message	description
json.exception.type_error.304	cannot use at() with string	The at() member functions can only be executed for certain JSON types.
json.exception.type_error.305	cannot use operator[] with string	The operator[] member functions can only be executed for certain JSON types.
json.exception.type_error.306	cannot use value() with string	The value() member functions can only be executed for certain JSON types.
json.exception.type_error.307	cannot use erase() with string	The erase() member functions can only be executed for certain JSON types.
json.exception.type_error.308	cannot use push_back() with string	The push_back() and operator+= member functions can only be executed for certain JSON types.
json.exception.type_error.309	cannot use insert() with	The insert() member functions can only be executed for certain JSON types.
json.exception.type_error.310	cannot use swap() with number	The swap() member functions can only be executed for certain JSON types.
json.exception.type_error.311	cannot use emplace_back() with string	The emplace_back() member function can only be executed for certain JS↩ ON types.
json.exception.type_error.312	cannot use update() with string	The update() member functions can only be executed for certain JSON types.
json.exception.type_error.313	invalid value to unflatten	The unflatten function converts an object whose keys are JSON Pointers back into an arbitrary nested JSON value. The JSON Pointers must not overlap, because then the resulting value would not be well defined.
json.exception.type_error.314	only objects can be unflattened	The unflatten function only works for an object whose keys are JSON Pointers.
json.exception.type_error.315	values in object must be primitive	The unflatten function only works for an object whose keys are JSON Pointers and whose values are primitive.

{The following code shows how a `type_error` exception can be caught.,type_error}

See also

exception for the base class of the library exceptions
parse_error for exceptions indicating a parse error
invalid_iterator for exceptions indicating errors with iterators
out_of_range for exceptions indicating access out of the defined range
other_error for exceptions indicating other library errors

Since

version 3.0.0

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/InputOutput/JsonInput/json.hpp

6.138 tao::operators::unit_steppable< T > Class Template Reference

Inheritance diagram for tao::operators::unit_steppable< T >:

Collaboration diagram for tao::operators::unit_steppable< T >:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.139 WilsonFlow Class Reference

Inheritance diagram for WilsonFlow:

Collaboration diagram for WilsonFlow:

Public Member Functions

- **WilsonFlow** (double tauFinal, double epsilon)
- void **flowConfigurations** ()
- void **setAction** (Action *action)
- void **addObservable** (Observable *observable)
- std::array< int, 4 > & **getSize** ()
- std::vector< double > & **getObsValues** ()
- std::vector< Observable * > & **getObs** ()
- void **createLattice** (std::array< int, 4 > latticeSize)
- void **execute** ()
- void **initialize** ()

Additional Inherited Members

The documentation for this class was generated from the following files:

- /home/giovanni/Desktop/LatticeYangMills/include/Apps/wilsonflow.h
- /home/giovanni/Desktop/LatticeYangMills/src/Apps/wilsonflow.cpp

6.140 xorable Class Reference

Inheritance diagram for xorable:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

6.141 xorable_left Class Reference

Inheritance diagram for xorable_left:

The documentation for this class was generated from the following file:

- /home/giovanni/Desktop/LatticeYangMills/include/Math/operators.hpp

Index