# My Project

Generated by Doxygen 1.16.1

# Chapter 1

# Requirements

| ID | |
|---|---|
| The | class satisfies the following concept requirements: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.<br><br>**Note**<br><br>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.∗∗<br><br>• `BidirectionalIterator`: The iterator that can be moved can be moved in both directions (i.e. incremented and decremented).<br><br>**Since**<br><br>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) |

## 1.1 Unsatisfied Requirements

The requirement The does not have a 'satisfies' relation.

## 1.2 Unverified Requirements

The requirement The does not have a 'verifies' relation.

# Chapter 2

# Topic Index

## 2.1 Topics

Here is a list of all topics with brief descriptions:

# Chapter 3

# Namespace Index

## 3.1 Namespace List

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

# Chapter 4

# Hierarchical Index

## 4.1 Class Hierarchy

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

# Chapter 5

# Class Index

## 5.1 Class List

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

# Chapter 6

# File Index

## 6.1 File List

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

# Chapter 7

# Topic Documentation

## 7.1 Optimization Algorithms

Base class interface for all numerical optimization algorithms.

**Files**

- file Blind.h

  *Header file for the Blind (Random Walk) optimization algorithm.*
- file LocalSearch.h

  *Header file for the Local Search optimization algorithm.*
- file OptimizerFactory.h

  *Factory utility for instantiating different optimizer types.*

### 7.1.1 Detailed Description

Base class interface for all numerical optimization algorithms.

## 7.2 Optimization Problems

Base class and group definition for all optimization benchmark functions.

**Files**

- file AckleyOne.h

  *Implementation of the Ackley 1 function.*
- file AckleyTwo.h

  *Implementation of the Ackley 2 function.*
- file EggHolder.h

  *Implementation of the Egg Holder function.*
- file Griewangk.h

  *Implementation of the Griewangk function.*
- file Rastrigin.h

  *Implementation of the Ratrigin function.*
- file Rosenbrock.h

  *Implementation of the Rosenbrock function.*
- file Schwefel.h

  *Implementation of the Schwefel function.*
- file SineEnvelope.h

  *Implementation of the Sine Envelope function.*
- file StretchedV.h

  *Implementation of the Stretched V function.*
- file ProblemFactory.h

  *Factory for instantiating benchmark problems by ID.*

### 7.2.1 Detailed Description

Base class and group definition for all optimization benchmark functions.

## 7.3 Core Engine

The heart of the application that manages configuration and execution.

**Files**

- file Config.h

  *Defines the configuration structure for optimization experiments.*
- file main.cpp

  *Entry point for the Numerical Optimization Benchmarks CLI.*

### 7.3.1 Detailed Description

The heart of the application that manages configuration and execution.

**Author**

Alex Buckley

# Chapter 8

# Namespace Documentation

## 8.1 detail Namespace Reference

detail namespace with internal helper functions

### Namespaces

- namespace dtoa_impl

  *implements the Grisu2 algorithm for binary to decimal floating-point conversion.*

### Classes

- struct make_void
- struct nonesuch
- struct detector
- struct detector< Default, void_t< Op< Args... > >, Op, Args... >
- struct is_detected_lazy
- struct position_t

  *struct to capture the start position of the current token*
- struct integer_sequence
- struct priority_tag
- struct priority_tag< 0 >
- struct static_const
- struct iterator_types
- struct iterator_types< It, void_t< typename It::difference_type, typename It::value_type, typename It::pointer, typename It::reference, typename It::iterator_category > >
- struct iterator_traits
- struct iterator_traits< T, enable_if_t< !std::is_pointer< T >::value > >
- struct iterator_traits< T *, enable_if_t< std::is_object< T >::value > >
- struct is_basic_json
- struct is_basic_json< NLOHMANN_BASIC_JSON_TPL >
- struct is_basic_json_context
- class json_ref
- struct is_json_ref
- struct is_json_ref< json_ref< T > >
- struct has_from_json

- struct is_getable
- struct has_from_json< BasicJsonType, T, enable_if_t< !is_basic_json< T >::value > >
- struct has_non_default_from_json
- struct has_non_default_from_json< BasicJsonType, T, enable_if_t< !is_basic_json< T >::value > >
- struct has_to_json
- struct has_to_json< BasicJsonType, T, enable_if_t< !is_basic_json< T >::value > >
- struct has_key_compare
- struct actual_object_comparator
- struct char_traits
- struct char_traits< unsigned char >
- struct char_traits< signed char >
- struct conjunction
- struct conjunction< B >
- struct conjunction< B, Bn... >
- struct negation
- struct is_default_constructible
- struct is_default_constructible< std::pair< T1, T2 > >
- struct is_default_constructible< const std::pair< T1, T2 > >
- struct is_default_constructible< std::tuple< Ts... > >
- struct is_default_constructible< const std::tuple< Ts... > >
- struct is_constructible
- struct is_constructible< std::pair< T1, T2 > >
- struct is_constructible< const std::pair< T1, T2 > >
- struct is_constructible< std::tuple< Ts... > >
- struct is_constructible< const std::tuple< Ts... > >
- struct is_iterator_traits
- struct is_iterator_traits< iterator_traits< T > >
- struct is_range
- struct is_complete_type
- struct is_complete_type< T, decltype(void(sizeof(T)))>
- struct is_compatible_object_type_impl
- struct is_compatible_object_type_impl< BasicJsonType, CompatibleObjectType, enable_if_t< is_detected< mapped_type_t, CompatibleObjectType >::value &&is_detected< key_type_t, CompatibleObjectType >::value > >
- struct is_compatible_object_type
- struct is_constructible_object_type_impl
- struct is_constructible_object_type_impl< BasicJsonType, ConstructibleObjectType, enable_if_t< is_detected< mapped_type_t, ConstructibleObjectType >::value &&is_detected< key_type_t, ConstructibleObjectType >::value > >
- struct is_constructible_object_type
- struct is_compatible_string_type
- struct is_constructible_string_type
- struct is_compatible_array_type_impl
- struct is_compatible_array_type_impl< BasicJsonType, CompatibleArrayType, enable_if_t< is_detected< iterator_t, CompatibleArrayType >::value &&is_iterator_traits< iterator_traits< detected_t< iterator_t, CompatibleArrayType > > >::value &&!std::is_same< CompatibleArrayType, detected_t< range_value_t, CompatibleArrayType > >::value > >
- struct is_compatible_array_type
- struct is_constructible_array_type_impl
- struct is_constructible_array_type_impl< BasicJsonType, ConstructibleArrayType, enable_if_t< std::is_same< ConstructibleArrayType, typename BasicJsonType::value_type >::value > >

- struct is_constructible_array_type_impl< BasicJsonType, ConstructibleArrayType, enable_if_t< !std::is_↩ same< ConstructibleArrayType, typename BasicJsonType::value_type >::value &&!is_compatible_string↩ _type< BasicJsonType, ConstructibleArrayType >::value &&is_default_constructible< ConstructibleArray↩ Type >::value &&(std::is_move_assignable< ConstructibleArrayType >::value||std::is_copy_assignable< ConstructibleArrayType >::value)&&is_detected< iterator_t, ConstructibleArrayType >::value &&is_iterator↩ _traits< iterator_traits< detected_t< iterator_t, ConstructibleArrayType > > >::value &&is_detected< range_value_t, ConstructibleArrayType >::value &&!std::is_same< ConstructibleArrayType, detected_↩ t< range_value_t, ConstructibleArrayType > >::value &&is_complete_type< detected_t< range_value_t, ConstructibleArrayType > >::value > >
- struct is_constructible_array_type
- struct is_compatible_integer_type_impl
- struct is_compatible_integer_type_impl< RealIntegerType, CompatibleNumberIntegerType, enable_if_t< std::is_integral< RealIntegerType >::value &&std::is_integral< CompatibleNumberIntegerType >::value &&!std::is_same< bool, CompatibleNumberIntegerType >::value > >
- struct is_compatible_integer_type
- struct is_compatible_type_impl
- struct is_compatible_type_impl< BasicJsonType, CompatibleType, enable_if_t< is_complete_type< CompatibleType >::value > >
- struct is_compatible_type
- struct is_constructible_tuple
- struct is_constructible_tuple< T1, std::tuple< Args... > >
- struct is_json_iterator_of
- struct is_json_iterator_of< BasicJsonType, typename BasicJsonType::iterator >
- struct is_json_iterator_of< BasicJsonType, typename BasicJsonType::const_iterator >
- struct is_specialization_of
- struct is_specialization_of< Primary, Primary< Args... > >
- struct is_json_pointer_of
- struct is_json_pointer_of< A, ::nlohmann::json_pointer< A > >
- struct is_json_pointer_of< A, ::nlohmann::json_pointer< A > & >
- struct is_comparable
- struct is_comparable< Compare, A, B, enable_if_t< !is_json_pointer_of< A, B >::value &&std::is↩ _constructible< decltype(std::declval< Compare >()(std::declval< A >(), std::declval< B >()))>::↩ value &&std::is_constructible< decltype(std::declval< Compare >()(std::declval< B >(), std::declval< A >()))>::value > >
- struct is_ordered_map
- struct value_in_range_of_impl2
- struct value_in_range_of_impl2< OfType, T, false, false >
- struct value_in_range_of_impl2< OfType, T, true, false >
- struct value_in_range_of_impl2< OfType, T, false, true >
- struct value_in_range_of_impl2< OfType, T, true, true >
- struct value_in_range_of_impl1
- struct value_in_range_of_impl1< OfType, T, false >
- struct value_in_range_of_impl1< OfType, T, true >
- struct is_c_string
- struct is_transparent
- class exception

  *general exception of the basic_json class*
- class parse_error

  *exception indicating a parse error*
- class invalid_iterator

  *exception indicating errors with iterators*
- class type_error

  *exception indicating executing a member function with a wrong type*
- class out_of_range

  *exception indicating access out of the defined range*

- class other_error
  *exception indicating other library errors*
- struct identity_tag
- struct from_json_fn
- class iteration_proxy_value
- class iteration_proxy
  *proxy class for the items() function*
- struct external_constructor
- struct external_constructor< value_t::boolean >
- struct external_constructor< value_t::string >
- struct external_constructor< value_t::binary >
- struct external_constructor< value_t::number_float >
- struct external_constructor< value_t::number_unsigned >
- struct external_constructor< value_t::number_integer >
- struct external_constructor< value_t::array >
- struct external_constructor< value_t::object >
- struct to_json_fn
- class file_input_adapter
- class input_stream_adapter
- class iterator_input_adapter
- struct wide_string_input_helper
- struct wide_string_input_helper< BaseInputAdapter, 4 >
- struct wide_string_input_helper< BaseInputAdapter, 2 >
- class wide_string_input_adapter
- struct iterator_input_adapter_factory
- struct is_iterator_of_multibyte
- struct iterator_input_adapter_factory< IteratorType, enable_if_t< is_iterator_of_multibyte< IteratorType >::value > >
- class span_input_adapter
- class lexer_base
- class lexer
  *lexical analysis*
- class json_sax_dom_parser
  *SAX implementation to create a JSON value from SAX events.*
- class json_sax_dom_callback_parser
- class json_sax_acceptor
- struct is_sax
- struct is_sax_static_asserts
- class binary_reader
  *deserialization of CBOR, MessagePack, and UBJSON values*
- class parser
  *syntax analysis*
- class primitive_iterator_t
- struct internal_iterator
  *an iterator value*
- class iter_impl
- class json_reverse_iterator
- struct json_default_base
  *Default base class of the basic_json class.*
- struct output_adapter_protocol
- class output_vector_adapter
- class output_stream_adapter
- class output_string_adapter
- class output_adapter
- class binary_writer
  *serialization to CBOR and MessagePack values*
- class serializer

### Typedefs

- template<typename ... Ts>
  using void_t = typename make_void<Ts...>::type
- template<template< class... > class Op, class... Args>
  using is_detected = typename detector<nonesuch, void, Op, Args...>::value_t
- template<template< class... > class Op, class... Args>
  using detected_t = typename detector<nonesuch, void, Op, Args...>::type
- template<class Default, template< class... > class Op, class... Args>
  using detected_or = detector<Default, void, Op, Args...>
- template<class Default, template< class... > class Op, class... Args>
  using detected_or_t = typename detected_or<Default, Op, Args...>::type
- template<class Expected, template< class... > class Op, class... Args>
  using is_detected_exact = std::is_same<Expected, detected_t<Op, Args...>>
- template<class To, template< class... > class Op, class... Args>
  using is_detected_convertible
- template<typename T>
  using uncvref_t = typename std::remove_cv<typename std::remove_reference<T>::type>::type
- template<bool B, typename T = void>
  using enable_if_t = typename std::enable_if<B, T>::type
- template<size_t... Ints>
  using index_sequence = integer_sequence<size_t, Ints...>
- template<typename T, T N>
  using make_integer_sequence = typename utility_internal::Gen<T, N>::type
- template<size_t N>
  using make_index_sequence = make_integer_sequence<size_t, N>
- template<typename... Ts>
  using index_sequence_for = make_index_sequence<sizeof...(Ts)>
- template<typename T>
  using mapped_type_t = typename T::mapped_type
- template<typename T>
  using key_type_t = typename T::key_type
- template<typename T>
  using value_type_t = typename T::value_type
- template<typename T>
  using difference_type_t = typename T::difference_type
- template<typename T>
  using pointer_t = typename T::pointer
- template<typename T>
  using reference_t = typename T::reference
- template<typename T>
  using iterator_category_t = typename T::iterator_category
- template<typename T, typename... Args>
  using to_json_function = decltype(T::to_json(std::declval<Args>()...))
- template<typename T, typename... Args>
  using from_json_function = decltype(T::from_json(std::declval<Args>()...))
- template<typename T, typename U>
  using get_template_function = decltype(std::declval<T>().template get<U>())
- template<typename T>
  using detect_key_compare = typename T::key_compare
- template<typename BasicJsonType>
  using actual_object_comparator_t = typename actual_object_comparator<BasicJsonType>::type
- template<typename R>
  using iterator_t = enable_if_t<is_range<R>::value, result_of_begin<decltype(std::declval<R&>())>>
- template<typename T>
  using range_value_t = value_type_t<iterator_traits<iterator_t<T>>>

- template<typename T>
  using is_json_pointer = is_specialization_of<::nlohmann::json_pointer, uncvref_t<T>>
- template<typename T>
  using detect_is_transparent = typename T::is_transparent
- template<typename Comparator, typename ObjectKeyType, typename KeyTypeCVRef, bool RequireTransparentComparator = true, bool ExcludeObjectKeyType = RequireTransparentComparator, typename KeyType = uncvref_t<KeyTypeCVRef>>
  using is_usable_as_key_type
- template<typename BasicJsonType, typename KeyTypeCVRef, bool RequireTransparentComparator = true, bool ExcludeObjectKeyType = RequireTransparentComparator, typename KeyType = uncvref_t<KeyTypeCVRef>>
  using is_usable_as_basic_json_key_type
- template<typename ObjectType, typename KeyType>
  using detect_erase_with_key_type = decltype(std::declval<ObjectType&>().erase(std::declval<KeyType>()))
- template<typename BasicJsonType, typename KeyType>
  using has_erase_with_key_type
- template<typename... Types>
  using all_integral = conjunction<std::is_integral<Types>...>
- template<typename... Types>
  using all_signed = conjunction<std::is_signed<Types>...>
- template<typename... Types>
  using all_unsigned = conjunction<std::is_unsigned<Types>...>
- template<typename... Types>
  using same_sign
- template<typename OfType, typename T>
  using never_out_of_range
- template<bool Value>
  using bool_constant = std::integral_constant<bool, Value>
- template<typename T>
  using is_c_string_uncvref = is_c_string<uncvref_t<T>>
- template<typename StringType, typename Arg>
  using string_can_append = decltype(std::declval<StringType&>().append(std::declval < Arg && > ()))
- template<typename StringType, typename Arg>
  using detect_string_can_append = is_detected<string_can_append, StringType, Arg>
- template<typename StringType, typename Arg>
  using string_can_append_op = decltype(std::declval<StringType&>() += std::declval < Arg && > ())
- template<typename StringType, typename Arg>
  using detect_string_can_append_op = is_detected<string_can_append_op, StringType, Arg>
- template<typename StringType, typename Arg>
  using string_can_append_iter = decltype(std::declval<StringType&>().append(std::declval<const Arg&>().begin(), std::declval<const Arg&>().end()))
- template<typename StringType, typename Arg>
  using detect_string_can_append_iter = is_detected<string_can_append_iter, StringType, Arg>
- template<typename StringType, typename Arg>
  using string_can_append_data = decltype(std::declval<StringType&>().append(std::declval<const Arg&>().data(), std::declval<const Arg&>().size()))
- template<typename StringType, typename Arg>
  using detect_string_can_append_data = is_detected<string_can_append_data, StringType, Arg>
- using string_input_adapter_type = decltype(input_adapter(std::declval<std::string>()))
- using contiguous_bytes_input_adapter = decltype(input_adapter(std::declval<const char*>(), std::declval<const char*>()))
- template<typename T>
  using null_function_t = decltype(std::declval<T&>().null())
- template<typename T>
  using boolean_function_t
- template<typename T, typename Integer>
  using number_integer_function_t

- template<typename T, typename Unsigned>
  using number_unsigned_function_t
- template<typename T, typename Float, typename String>
  using number_float_function_t
- template<typename T, typename String>
  using string_function_t
- template<typename T, typename Binary>
  using binary_function_t
- template<typename T>
  using start_object_function_t
- template<typename T, typename String>
  using key_function_t
- template<typename T>
  using end_object_function_t = decltype(std::declval<T&>().end_object())
- template<typename T>
  using start_array_function_t
- template<typename T>
  using end_array_function_t = decltype(std::declval<T&>().end_array())
- template<typename T, typename Exception>
  using parse_error_function_t
- template<typename BasicJsonType>
  using parser_callback_t
- template<class T>
  using json_base_class
- template<typename CharType>
  using output_adapter_t = std::shared_ptr<output_adapter_protocol<CharType>>

## Enumerations

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

  *the JSON type enumeration*
- enum class input_format_t {
  **json** , **cbor** , **msgpack** , **ubjson** ,
  **bson** , **bjdata** }

  *the supported input formats*
- enum class cbor_tag_handler_t { error , ignore , store }

  *how to treat CBOR tags*
- enum class **parse_event_t** : std::uint8_t { **read** , **array_end** , **key** , **value** }
- enum class **bjdata_version_t** { **draft2** , **draft3** }
- enum class error_handler_t { strict , replace , ignore }

## Functions

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

  *comparison operator for JSON types*
- template<typename StringType>
  void replace_substring (StringType &s, const StringType &f, const StringType &t)

  *replace all occurrences of a substring by another string*
- template<typename StringType>
  StringType escape (StringType s)

*string escaping as described in RFC 6901 (Sect. 4)*

- template<typename StringType>
  void unescape (StringType &s)

  *string unescaping as described in RFC 6901 (Sect. 4)*
- template<typename T, typename... Args>
  constexpr std::array< T, sizeof...(Args)> make_array (Args &&... args)
- template<typename T, typename U, enable_if_t< !std::is_same< T, U >::value, int > = 0>
  T conditional_static_cast (U value)
- template<typename T, typename U, enable_if_t< std::is_same< T, U >::value, int > = 0>
  T conditional_static_cast (U value)
- template<typename OfType, typename T>
  constexpr bool value_in_range_of (T val)
- std::size_t concat_length ()
- template<typename... Args>
  std::size_t concat_length (const char *cstr, const Args &... rest)
- template<typename StringType, typename... Args>
  std::size_t concat_length (const StringType &str, const Args &... rest)
- template<typename... Args>
  std::size_t concat_length (const char, const Args &... rest)
- template<typename OutStringType>
  void concat_into (OutStringType &)
- template<typename OutStringType, typename Arg, typename... Args, enable_if_t< !detect_string_can_append< OutStringType, Arg >::value &&detect_string_can_append_op< OutStringType, Arg >::value, int > = 0>
  void concat_into (OutStringType &out, Arg &&arg, Args &&... rest)
- template<typename OutStringType, typename Arg, typename... Args, enable_if_t< !detect_string_can_append< OutStringType, Arg >::value &&!detect_string_can_append_op< OutStringType, Arg >::value &&detect_string_can_append_iter< OutStringType, Arg >::value, int > = 0>
  void concat_into (OutStringType &out, const Arg &arg, Args &&... rest)
- template<typename OutStringType, typename Arg, typename... Args, enable_if_t< !detect_string_can_append< OutStringType, Arg >::value &&!detect_string_can_append_op< OutStringType, Arg >::value &&!detect_string_can_append_iter< OutStringType, Arg >::value &&detect_string_can_append_data< OutStringType, Arg >::value, int > = 0>
  void concat_into (OutStringType &out, const Arg &arg, Args &&... rest)
- template<typename OutStringType, typename Arg, typename... Args, enable_if_t< detect_string_can_append< OutStringType, Arg >::value, int > = 0>
  void concat_into (OutStringType &out, Arg &&arg, Args &&... rest)
- template<typename OutStringType = std::string, typename... Args>
  OutStringType concat (Args &&... args)
- template<typename BasicJsonType>
  void from_json (const BasicJsonType &j, typename std::nullptr_t &n)
- template<typename BasicJsonType, typename ArithmeticType, enable_if_t< std::is_arithmetic< ArithmeticType >::value &&!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, typename StringType, enable_if_t< std::is_assignable< StringType &, const typename Basic↩JsonType::string_t >::value &&is_detected_exact< typename BasicJsonType::string_t::value_type, value_type_t, StringType >::value &&!std::is_same< typename BasicJsonType::string_t, StringType >::value &&!is_json_ref< StringType >::value, int > = 0>
  void from_json (const BasicJsonType &j, StringType &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, typename T, typename Allocator, enable_if_t< is_getable< 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< is_getable< BasicJsonType, T >::value, int > = 0>
  void from_json (const BasicJsonType &j, std::valarray< T > &l)
- template<typename BasicJsonType, typename T, std::size_t N>
  auto from_json (const BasicJsonType &j, T(&arr)[N]) -> decltype(j.template get< T >(), void())
- template<typename BasicJsonType, typename T, std::size_t N1, std::size_t N2>
  auto from_json (const BasicJsonType &j, T(&arr)[N1][N2]) -> decltype(j.template get< T >(), void())
- template<typename BasicJsonType, typename T, std::size_t N1, std::size_t N2, std::size_t N3>
  auto from_json (const BasicJsonType &j, T(&arr)[N1][N2][N3]) -> decltype(j.template get< T >(), void())
- template<typename BasicJsonType, typename T, std::size_t N1, std::size_t N2, std::size_t N3, std::size_t N4>
  auto from_json (const BasicJsonType &j, T(&arr)[N1][N2][N3][N4]) -> decltype(j.template get< T >(), void())
- template<typename BasicJsonType>
  void from_json_array_impl (const BasicJsonType &j, typename BasicJsonType::array_t &arr, priority_tag< 3 >)
- template<typename BasicJsonType, typename T, std::size_t N>
  auto from_json_array_impl (const BasicJsonType &j, std::array< T, N > &arr, priority_tag< 2 >) -> decltype(j.template get< T >(), void())
- template<typename BasicJsonType, typename ConstructibleArrayType, enable_if_t< std::is_assignable< ConstructibleArrayType &, ConstructibleArrayType >::value, int > = 0>
  auto from_json_array_impl (const BasicJsonType &j, ConstructibleArrayType &arr, priority_tag< 1 >) -> decltype(arr.reserve(std::declval< typename ConstructibleArrayType::size_type >()), j.template get< typename ConstructibleArrayType::value_type >(), void())
- template<typename BasicJsonType, typename ConstructibleArrayType, enable_if_t< std::is_assignable< ConstructibleArrayType &, ConstructibleArrayType >::value, int > = 0>
  void from_json_array_impl (const BasicJsonType &j, ConstructibleArrayType &arr, priority_tag< 0 >)
- template<typename BasicJsonType, typename ConstructibleArrayType, enable_if_t< is_constructible_array_type< BasicJsonType, ConstructibleArrayType >::value &&!is_constructible_object_type< BasicJsonType, ConstructibleArrayType >::value &&!is_constructible_string_type< BasicJsonType, ConstructibleArrayType >::value &&!std::is_same< ConstructibleArrayType, typename BasicJsonType::binary_t >::value &&!is_basic_json< ConstructibleArrayType >::value, int > = 0>
  auto from_json (const BasicJsonType &j, ConstructibleArrayType &arr) -> decltype(from_json_array_impl(j, arr, priority_tag< 3 > {}), j.template get< typename ConstructibleArrayType::value_type >(), void())
- template<typename BasicJsonType, typename T, std::size_t... Idx>
  std::array< T, sizeof...(Idx)> from_json_inplace_array_impl (BasicJsonType &&j, identity_tag< std::array< T, sizeof...(Idx)> >, index_sequence< Idx... >)
- template<typename BasicJsonType, typename T, std::size_t N>
  auto from_json (BasicJsonType &&j, identity_tag< std::array< T, N > > tag) -> decltype(from_json_inplace_array_impl(std::forward< BasicJsonType >(j), tag, make_index_sequence< N > {}))
- template<typename BasicJsonType>
  void from_json (const BasicJsonType &j, typename BasicJsonType::binary_t &bin)
- template<typename BasicJsonType, typename ConstructibleObjectType, enable_if_t< is_constructible_object_type< BasicJsonType, ConstructibleObjectType >::value, int > = 0>
  void from_json (const BasicJsonType &j, ConstructibleObjectType &obj)
- template<typename BasicJsonType, typename ArithmeticType, enable_if_t< std::is_arithmetic< ArithmeticType >::value &&!std::is_same< ArithmeticType, typename BasicJsonType::number_unsigned_t >::value &&!std::is_same< ArithmeticType, typename BasicJsonType::number_integer_t >::value &&!std::is_same< ArithmeticType, typename BasicJsonType::number_float_t >::value &&!std::is_same< ArithmeticType, typename BasicJsonType::boolean_t >::value, int > = 0>
  void from_json (const BasicJsonType &j, ArithmeticType &val)
- template<typename BasicJsonType, typename... Args, std::size_t... Idx>
  std::tuple< Args... > from_json_tuple_impl_base (BasicJsonType &&j, index_sequence< Idx... >)
- template<typename BasicJsonType>
  std::tuple from_json_tuple_impl_base (BasicJsonType &, index_sequence<>)
- template<typename BasicJsonType, class A1, class A2>
  std::pair< A1, A2 > from_json_tuple_impl (BasicJsonType &&j, identity_tag< std::pair< A1, A2 > >, priority_tag< 0 >)

- template<typename BasicJsonType, typename A1, typename A2>
  void from_json_tuple_impl (BasicJsonType &&j, std::pair< A1, A2 > &p, priority_tag< 1 >)
- template<typename BasicJsonType, typename... Args>
  std::tuple< Args... > from_json_tuple_impl (BasicJsonType &&j, identity_tag< std::tuple< Args... > >, priority_tag< 2 >)
- template<typename BasicJsonType, typename... Args>
  void from_json_tuple_impl (BasicJsonType &&j, std::tuple< Args... > &t, priority_tag< 3 >)
- template<typename BasicJsonType, typename TupleRelated>
  auto from_json (BasicJsonType &&j, TupleRelated &&t) -> decltype(from_json_tuple_impl(std::forward< BasicJsonType >(j), std::forward< TupleRelated >(t), priority_tag< 3 > {}))
- template<typename BasicJsonType, typename Key, typename Value, typename Compare, typename Allocator, typename = enable_if_t < !std::is_constructible < typename BasicJsonType::string_t, Key >::value >>
  void from_json (const BasicJsonType &j, std::map< Key, Value, Compare, Allocator > &m)
- template<typename BasicJsonType, typename Key, typename Value, typename Hash, typename KeyEqual, typename Allocator, typename = enable_if_t < !std::is_constructible < typename BasicJsonType::string_t, Key >::value >>
  void from_json (const BasicJsonType &j, std::unordered_map< Key, Value, Hash, KeyEqual, Allocator > &m)
- template<typename StringType>
  void int_to_string (StringType &target, std::size_t value)
- template<typename StringType>
  StringType to_string (std::size_t value)
- template<std::size_t N, typename IteratorType, enable_if_t< N==0, int > = 0>
  auto get (const nlohmann::detail::iteration_proxy_value< IteratorType > &i) -> decltype(i.key())
- template<std::size_t N, typename IteratorType, enable_if_t< N==1, int > = 0>
  auto get (const nlohmann::detail::iteration_proxy_value< IteratorType > &i) -> decltype(i.value())
- 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 BoolRef, enable_if_t<((std::is_same< std::vector< bool >::reference, BoolRef >::value &&!std::is_same< std::vector< bool >::reference, typename BasicJsonType::boolean_t & >::value)||(std::is_same< std::vector< bool >::const_reference, BoolRef >::value &&!std::is_same< detail::uncvref_t< std::vector< bool >::const_reference >, typename BasicJsonType::boolean_t >::value)) &&std::is_convertible< const BoolRef &, typename BasicJsonType::boolean_t >::value, int > = 0>
  void to_json (BasicJsonType &j, const BoolRef &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 &&!is_compatible_object_type< BasicJsonType, CompatibleArrayType >::value &&!is_compatible_string_type< BasicJsonType, CompatibleArrayType >::value &&!std::is_same< typename BasicJsonType::binary_t, CompatibleArrayType >::value &&!is_basic_json< CompatibleArrayType >::value, int > = 0>
  void to_json (BasicJsonType &j, const CompatibleArrayType &arr)
- template<typename BasicJsonType>
  void to_json (BasicJsonType &j, const typename BasicJsonType::binary_t &bin)

- template<typename BasicJsonType, typename T, enable_if_t< std::is_convertible< T, BasicJsonType >::value, int > = 0>
  void to_json (BasicJsonType &j, const 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 &&!is_basic_json< 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< !std::is_constructible< typename BasicJsonType::string_t, const T(&)[N]>::value, int > = 0>
  void to_json (BasicJsonType &j, const T(&arr)[N])
- template<typename BasicJsonType, typename T1, typename T2, enable_if_t< std::is_constructible< BasicJsonType, T1 >::value &&std::is_constructible< BasicJsonType, T2 >::value, int > = 0>
  void to_json (BasicJsonType &j, const std::pair< T1, T2 > &p)
- template<typename BasicJsonType, typename T, enable_if_t< std::is_same< T, iteration_proxy_value< typename BasicJsonType::iterator > >::value, int > = 0>
  void to_json (BasicJsonType &j, const T &b)
- 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 Tuple>
  void to_json_tuple_impl (BasicJsonType &j, const Tuple &, index_sequence<>)
- template<typename BasicJsonType, typename T, enable_if_t< is_constructible_tuple< BasicJsonType, T >::value, int > = 0>
  void to_json (BasicJsonType &j, const T &t)
- std::size_t combine (std::size_t seed, std::size_t h) noexcept
- template<typename BasicJsonType>
  std::size_t hash (const BasicJsonType &j)

  *hash a JSON value*
- template<typename IteratorType>
  iterator_input_adapter_factory< IteratorType >::adapter_type input_adapter (IteratorType first, IteratorType last)
- template<typename ContainerType>
  container_input_adapter_factory_impl::container_input_adapter_factory< ContainerType >::adapter_type input_adapter (const ContainerType &container)
- file_input_adapter input_adapter (std::FILE *file)
- input_stream_adapter input_adapter (std::istream &stream)
- input_stream_adapter input_adapter (std::istream &&stream)
- template<typename CharT, typename std::enable_if< std::is_pointer< CharT >::value &&!std::is_array< CharT >::value &&std::is_integral< typename std::remove_pointer< CharT >::type >::value &&sizeof(typename std::remove_pointer< CharT >::type)==1, int >::type = 0>
  contiguous_bytes_input_adapter input_adapter (CharT b)
- template<typename T, std::size_t N>
  auto input_adapter (T(&array)[N]) -> decltype(input_adapter(array, array+N))
- constexpr std::size_t unknown_size ()
- bool little_endianness (int num=1) noexcept

  *determine system byte order*
- template<typename FloatType>
  JSON_HEDLEY_RETURNS_NON_NULL char * to_chars (char *first, const char *last, FloatType value)

  *generates a decimal representation of the floating-point number value in [first, last).*

## Variables

- template<typename T>
  constexpr T static_const< T >::value
- template<typename BasicJsonType, typename InputAdapterType, typename SAX>
  constexpr std::size_t binary_reader< BasicJsonType, InputAdapterType, SAX >::npos

### 8.1.1 Detailed Description

detail namespace with internal helper functions

This namespace collects functions that should not be exposed, implementations of some basic_json methods, and meta-programming helpers.

**Since**

version 2.1.0

### 8.1.2 Typedef Documentation

#### 8.1.2.1 actual_object_comparator_t

```
template<typename BasicJsonType>
using detail::actual_object_comparator_t = typename actual_object_comparator<BasicJsonType>::↩
type
```

Definition at line 3729 of file json.hpp.

#### 8.1.2.2 all_integral

```
template<typename...  Types>
using detail::all_integral = conjunction<std::is_integral<Types>...>
```

Definition at line 4237 of file json.hpp.

#### 8.1.2.3 all_signed

```
template<typename...  Types>
using detail::all_signed = conjunction<std::is_signed<Types>...>
```

Definition at line 4240 of file json.hpp.

#### 8.1.2.4 all_unsigned

```
template<typename...  Types>
using detail::all_unsigned = conjunction<std::is_unsigned<Types>...>
```

Definition at line 4243 of file json.hpp.

#### 8.1.2.5 binary_function_t

```
template<typename T, typename Binary>
using detail::binary_function_t
```

**Initial value:**

```
decltype(std::declval<T&>().binary(std::declval<Binary&>()))
```

Definition at line 9758 of file json.hpp.

#### 8.1.2.6 bool_constant

```
template<bool Value>
using detail::bool_constant = std::integral_constant<bool, Value>
```

Definition at line 4331 of file json.hpp.

#### 8.1.2.7 boolean_function_t

```
template<typename T>
using detail::boolean_function_t
```

**Initial value:**

```
decltype(std::declval<T&>().boolean(std::declval<bool>()))
```

Definition at line 9738 of file json.hpp.

#### 8.1.2.8 contiguous_bytes_input_adapter

```
using detail::contiguous_bytes_input_adapter = decltype(input_adapter(std::declval<const char*>(),
std::declval<const char*>()))
```

Definition at line 7004 of file json.hpp.

#### 8.1.2.9 detect_erase_with_key_type

```
template<typename ObjectType, typename KeyType>
using detail::detect_erase_with_key_type = decltype(std::declval<ObjectType&>().erase(std::↩
declval<KeyType>()))
```

Definition at line 4194 of file json.hpp.

#### 8.1.2.10 detect_is_transparent

```
template<typename T>
using detail::detect_is_transparent = typename T::is_transparent
```

Definition at line 4158 of file json.hpp.

#### 8.1.2.11 detect_key_compare

```
template<typename T>
using detail::detect_key_compare = typename T::key_compare
```

Definition at line 3713 of file json.hpp.

#### 8.1.2.12 detect_string_can_append

```
template<typename StringType, typename Arg>
using detail::detect_string_can_append = is_detected<string_can_append, StringType, Arg>
```

Definition at line 4447 of file json.hpp.

#### 8.1.2.13 detect_string_can_append_data

```
template<typename StringType, typename Arg>
using detail::detect_string_can_append_data = is_detected<string_can_append_data, StringType,
Arg>
```

Definition at line 4465 of file json.hpp.

#### 8.1.2.14 detect_string_can_append_iter

```
template<typename StringType, typename Arg>
using detail::detect_string_can_append_iter = is_detected<string_can_append_iter, StringType,
Arg>
```

Definition at line 4459 of file json.hpp.

#### 8.1.2.15 detect_string_can_append_op

```
template<typename StringType, typename Arg>
using detail::detect_string_can_append_op = is_detected<string_can_append_op, StringType, Arg>
```

Definition at line 4453 of file json.hpp.

#### 8.1.2.16 detected_or

```
template<class Default, template< class...  > class Op, class...  Args>
using detail::detected_or = detector<Default, void, Op, Args...>
```

Definition at line 314 of file json.hpp.

#### 8.1.2.17 detected_or_t

```
template<class Default, template< class...  > class Op, class...  Args>
using detail::detected_or_t = typename detected_or<Default, Op, Args...>::type
```

Definition at line 317 of file json.hpp.

#### 8.1.2.18 detected_t

```
template<template< class...  > class Op, class...  Args>
using detail::detected_t = typename detector<nonesuch, void, Op, Args...>::type
```

Definition at line 311 of file json.hpp.

#### 8.1.2.19 difference_type_t

```
template<typename T>
using detail::difference_type_t = typename T::difference_type
```

Definition at line 3638 of file json.hpp.

#### 8.1.2.20 enable_if_t

```
template<bool B, typename T = void>
using detail::enable_if_t = typename std::enable_if<B, T>::type
```

Definition at line 3226 of file json.hpp.

#### 8.1.2.21 end_array_function_t

```
template<typename T>
using detail::end_array_function_t = decltype(std::declval<T&>().end_array())
```

Definition at line 9777 of file json.hpp.

#### 8.1.2.22 end_object_function_t

```
template<typename T>
using detail::end_object_function_t = decltype(std::declval<T&>().end_object())
```

Definition at line 9770 of file json.hpp.

#### 8.1.2.23 from_json_function

```
template<typename T, typename...  Args>
using detail::from_json_function = decltype(T::from_json(std::declval<Args>()...))
```

Definition at line 3653 of file json.hpp.

#### 8.1.2.24 get_template_function

```
template<typename T, typename U>
using detail::get_template_function = decltype(std::declval<T>().template get<U>())
```

Definition at line 3656 of file json.hpp.

#### 8.1.2.25 has_erase_with_key_type

```
template<typename BasicJsonType, typename KeyType>
using detail::has_erase_with_key_type
```

**Initial value:**

```
typename std::conditional <
                              is_detected <
                              detect_erase_with_key_type,
                              typename BasicJsonType::object_t, KeyType >::value,
                              std::true_type,
                              std::false_type >::type
```

Definition at line 4198 of file json.hpp.

#### 8.1.2.26 index_sequence

```
template<size_t...  Ints>
using detail::index_sequence = integer_sequence<size_t, Ints...>
```

Definition at line 3268 of file json.hpp.

#### 8.1.2.27 index_sequence_for

```
template<typename...  Ts>
using detail::index_sequence_for = make_index_sequence<sizeof...(Ts)>
```

Definition at line 3330 of file json.hpp.

#### 8.1.2.28 is_c_string_uncvref

```
template<typename T>
using detail::is_c_string_uncvref = is_c_string<uncvref_t<T>>
```

Definition at line 4359 of file json.hpp.

#### 8.1.2.29 is_detected

```
template<template< class...  > class Op, class...  Args>
using detail::is_detected = typename detector<nonesuch, void, Op, Args...>::value_t
```

Definition at line 305 of file json.hpp.

#### 8.1.2.30 is_detected_convertible

```
template<class To, template< class...  > class Op, class...  Args>
using detail::is_detected_convertible
```

**Initial value:**

```
std::is_convertible<detected_t<Op, Args...>, To>
```

Definition at line 323 of file json.hpp.

#### 8.1.2.31 is_detected_exact

```
template<class Expected, template< class...  > class Op, class...  Args>
using detail::is_detected_exact = std::is_same<Expected, detected_t<Op, Args...>>
```

Definition at line 320 of file json.hpp.

#### 8.1.2.32 is_json_pointer

```
template<typename T>
using detail::is_json_pointer = is_specialization_of<::nlohmann::json_pointer, uncvref_t<T>>
```

Definition at line 4130 of file json.hpp.

#### 8.1.2.33 is_usable_as_basic_json_key_type

```
template<typename BasicJsonType, typename KeyTypeCVRef, bool RequireTransparentComparator =
true, bool ExcludeObjectKeyType = RequireTransparentComparator, typename KeyType = uncvref_↩
t<KeyTypeCVRef>>
using detail::is_usable_as_basic_json_key_type
```

**Initial value:**

```
typename std::conditional <
   (is_usable_as_key_type<typename BasicJsonType::object_comparator_t,
    typename BasicJsonType::object_t::key_type, KeyTypeCVRef,
    RequireTransparentComparator, ExcludeObjectKeyType>::value
    && !is_json_iterator_of<BasicJsonType, KeyType>::value)


   , std::true_type,
   std::false_type >::type
```

Definition at line 4182 of file json.hpp.

#### 8.1.2.34 is_usable_as_key_type

```
template<typename Comparator, typename ObjectKeyType, typename KeyTypeCVRef, bool Require↩
TransparentComparator = true, bool ExcludeObjectKeyType = RequireTransparentComparator, typename
KeyType = uncvref_t<KeyTypeCVRef>>
using detail::is_usable_as_key_type
```

**Initial value:**

```
typename std::conditional <
                           is_comparable<Comparator, ObjectKeyType, KeyTypeCVRef>::value
                           && !(ExcludeObjectKeyType && std::is_same<KeyType,
                                ObjectKeyType>::value)
                           && (!RequireTransparentComparator
                               || is_detected <detect_is_transparent, Comparator>::value)
                           && !is_json_pointer<KeyType>::value,
                           std::true_type,
                           std::false_type >::type
```

Definition at line 4164 of file json.hpp.

#### 8.1.2.35 iterator_category_t

```
template<typename T>
using detail::iterator_category_t = typename T::iterator_category
```

Definition at line 3647 of file json.hpp.

#### 8.1.2.36 iterator_t

```
template<typename R>
using detail::iterator_t = enable_if_t<is_range<R>::value, result_of_begin<decltype(std::↩
declval<R&>())>>
```

Definition at line 3901 of file json.hpp.

#### 8.1.2.37 json_base_class

```
template<class T>
using detail::json_base_class
```

**Initial value:**

```
typename std::conditional <
                     std::is_same<T, void>::value,
                     json_default_base,
                     T
                     >::type
```

Definition at line 14592 of file json.hpp.

#### 8.1.2.38 key_function_t

```
template<typename T, typename String>
using detail::key_function_t
```

**Initial value:**

```
decltype(std::declval<T&>().key(std::declval<String&>()))
```

Definition at line 9766 of file json.hpp.

#### 8.1.2.39 key_type_t

```
template<typename T>
using detail::key_type_t = typename T::key_type
```

Definition at line 3632 of file json.hpp.

#### 8.1.2.40 make_index_sequence

```
template<size_t N>
using detail::make_index_sequence = make_integer_sequence<size_t, N>
```

Definition at line 3322 of file json.hpp.

#### 8.1.2.41 make_integer_sequence

```
template<typename T, T N>
using detail::make_integer_sequence = typename utility_internal::Gen<T, N>::type
```

Definition at line 3314 of file json.hpp.

#### 8.1.2.42 mapped_type_t

```
template<typename T>
using detail::mapped_type_t = typename T::mapped_type
```

Definition at line 3629 of file json.hpp.

#### 8.1.2.43 never_out_of_range

```
template<typename OfType, typename T>
using detail::never_out_of_range
```

**Initial value:**

```
std::integral_constant < bool,
     (std::is_signed<OfType>::value && (sizeof(T) < sizeof(OfType)))
     || (same_sign<OfType, T>::value && sizeof(OfType) == sizeof(T)) >
```

Definition at line 4251 of file json.hpp.

#### 8.1.2.44 null_function_t

```
template<typename T>
using detail::null_function_t = decltype(std::declval<T&>().null())
```

Definition at line 9735 of file json.hpp.

#### 8.1.2.45 number_float_function_t

```
template<typename T, typename Float, typename String>
using detail::number_float_function_t
```

**Initial value:**

```
decltype(std::declval<T&>().number_float(
                                    std::declval<Float>(), std::declval<const String&>()))
```

Definition at line 9750 of file json.hpp.

#### 8.1.2.46 number_integer_function_t

```
template<typename T, typename Integer>
using detail::number_integer_function_t
```

**Initial value:**

```
decltype(std::declval<T&>().number_integer(std::declval<Integer>()))
```

Definition at line 9742 of file json.hpp.

#### 8.1.2.47 number_unsigned_function_t

```
template<typename T, typename Unsigned>
using detail::number_unsigned_function_t
```

**Initial value:**

```
decltype(std::declval<T&>().number_unsigned(std::declval<Unsigned>()))
```

Definition at line 9746 of file json.hpp.

#### 8.1.2.48 output_adapter_t

```
template<typename CharType>
using detail::output_adapter_t = std::shared_ptr<output_adapter_protocol<CharType>>
```

Definition at line 15762 of file json.hpp.

#### 8.1.2.49 parse_error_function_t

```
template<typename T, typename Exception>
using detail::parse_error_function_t
```

**Initial value:**

```
decltype(std::declval<T&>().parse_error(
       std::declval<std::size_t>(), std::declval<const std::string&>(),
       std::declval<const Exception&>()))
```

Definition at line 9780 of file json.hpp.

#### 8.1.2.50 parser_callback_t

```
template<typename BasicJsonType>
using detail::parser_callback_t
```

**Initial value:**

```
std::function<bool(int , parse_event_t , BasicJsonType& )>
```

Definition at line 12994 of file json.hpp.

#### 8.1.2.51 pointer_t

```
template<typename T>
using detail::pointer_t = typename T::pointer
```

Definition at line 3641 of file json.hpp.

#### 8.1.2.52 range_value_t

```
template<typename T>
using detail::range_value_t = value_type_t<iterator_traits<iterator_t<T>>>
```

Definition at line 3904 of file json.hpp.

#### 8.1.2.53 reference_t

```
template<typename T>
using detail::reference_t = typename T::reference
```

Definition at line 3644 of file json.hpp.

#### 8.1.2.54 same_sign

```
template<typename...  Types>
using detail::same_sign
```

**Initial value:**

```
std::integral_constant < bool,
     all_signed<Types...>::value || all_unsigned<Types...>::value >
```

Definition at line 4247 of file json.hpp.

#### 8.1.2.55 start_array_function_t

```
template<typename T>
using detail::start_array_function_t
```

**Initial value:**

```
decltype(std::declval<T&>().start_array(std::declval<std::size_t>()))
```

Definition at line 9773 of file json.hpp.

#### 8.1.2.56 start_object_function_t

```
template<typename T>
using detail::start_object_function_t
```

**Initial value:**

```
decltype(std::declval<T&>().start_object(std::declval<std::size_t>()))
```

Definition at line 9762 of file json.hpp.

#### 8.1.2.57 string_can_append

```
template<typename StringType, typename Arg>
using detail::string_can_append = decltype(std::declval<StringType&>().append(std::declval <
Arg && > ()))
```

Definition at line 4444 of file json.hpp.

#### 8.1.2.58 string_can_append_data

```
template<typename StringType, typename Arg>
using detail::string_can_append_data = decltype(std::declval<StringType&>().append(std::↩
declval<const Arg&>().data(), std::declval<const Arg&>().size()))
```

Definition at line 4462 of file json.hpp.

#### 8.1.2.59 string_can_append_iter

```
template<typename StringType, typename Arg>
using detail::string_can_append_iter = decltype(std::declval<StringType&>().append(std::↩
declval<const Arg&>().begin(), std::declval<const Arg&>().end()))
```

Definition at line 4456 of file json.hpp.

#### 8.1.2.60 string_can_append_op

```
template<typename StringType, typename Arg>
using detail::string_can_append_op = decltype(std::declval<StringType&>() += std::declval <
Arg && > ())
```

Definition at line 4450 of file json.hpp.

#### 8.1.2.61 string_function_t

```
template<typename T, typename String>
using detail::string_function_t
```

**Initial value:**

```
decltype(std::declval<T&>().string(std::declval<String&>()))
```

Definition at line 9754 of file json.hpp.

#### 8.1.2.62 string_input_adapter_type

```
using detail::string_input_adapter_type = decltype(input_adapter(std::declval<std::string>()))
```

Definition at line 6980 of file json.hpp.

#### 8.1.2.63 to_json_function

```
template<typename T, typename...  Args>
using detail::to_json_function = decltype(T::to_json(std::declval<Args>()...))
```

Definition at line 3650 of file json.hpp.

#### 8.1.2.64 uncvref_t

```
template<typename T>
using detail::uncvref_t = typename std::remove_cv<typename std::remove_reference<T>::type>::↩
type
```

Definition at line 3212 of file json.hpp.

#### 8.1.2.65 value_type_t

```
template<typename T>
using detail::value_type_t = typename T::value_type
```

Definition at line 3635 of file json.hpp.

#### 8.1.2.66 void_t

```
template<typename ...  Ts>
using detail::void_t = typename make_void<Ts...>::type
```

Definition at line 266 of file json.hpp.

### 8.1.3 Enumeration Type Documentation

#### 8.1.3.1 bjdata_version_t

```
enum class detail::bjdata_version_t  [strong]
```

Definition at line 15874 of file json.hpp.

#### 8.1.3.2 cbor_tag_handler_t

```
enum class detail::cbor_tag_handler_t  [strong]
```

how to treat CBOR tags

**Enumerator**

| | |
|---|---|
| error | throw a parse_error exception in case of a tag |

| | |
|---|---|
| ignore | ignore tags |
| store | store tags as binary type |

Definition at line 9884 of file json.hpp.

#### 8.1.3.3 error_handler_t

```
enum class detail::error_handler_t  [strong]
```

**Enumerator**

| | |
|---|---|
| strict | throw a type_error exception in case of invalid UTF-8 |
| replace | replace invalid UTF-8 sequences with U+FFFD |
| ignore | ignore invalid UTF-8 sequences |

Definition at line 18879 of file json.hpp.

#### 8.1.3.4 input_format_t

```
enum class detail::input_format_t  [strong]
```

the supported input formats

Definition at line 6550 of file json.hpp.

#### 8.1.3.5 parse_event_t

```
enum class detail::parse_event_t :  std::uint8_t  [strong]
```

Definition at line 12977 of file json.hpp.

#### 8.1.3.6 value_t

```
enum class detail::value_t :  std::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**

see 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) |
| binary | binary array (ordered collection of bytes) |
| discarded | discarded by the parser callback function |

Definition at line 3003 of file json.hpp.

## 8.1.4 Function Documentation

### 8.1.4.1 combine()

```
std::size_t detail::combine (
            std::size_t seed,
            std::size_t h)  [inline], [noexcept]
```

Definition at line 6372 of file json.hpp.

### 8.1.4.2 concat()

```
template<typename OutStringType = std::string, typename...  Args>
OutStringType detail::concat (
            Args &&...  args)  [inline]
```

Definition at line 4524 of file json.hpp.

#### 8.1.4.3 concat_into() [1/5]

```
template<typename OutStringType>
void detail::concat_into (
            OutStringType & )  [inline]
```

Definition at line 4440 of file json.hpp.

#### 8.1.4.4 concat_into() [2/5]

```
template<typename OutStringType, typename Arg, typename...  Args, enable_if_t< !detect_↩
string_can_append< OutStringType, Arg >::value &&detect_string_can_append_op< OutStringType,
Arg >::value, int > = 0>
void detail::concat_into (
            OutStringType & out,
            Arg && arg,
            Args &&...  rest)  [inline]
```

Definition at line 4496 of file json.hpp.

#### 8.1.4.5 concat_into() [3/5]

```
template<typename OutStringType, typename Arg, typename...  Args, enable_if_t< detect_string↩
_can_append< OutStringType, Arg >::value, int > = 0>
void detail::concat_into (
            OutStringType & out,
            Arg && arg,
            Args &&...  rest)  [inline]
```

Definition at line 4487 of file json.hpp.

#### 8.1.4.6 concat_into() [4/5]

```
template<typename OutStringType, typename Arg, typename...  Args, enable_if_t< !detect_↩
string_can_append< OutStringType, Arg >::value &&!detect_string_can_append_op< OutStringType,
Arg >::value &&!detect_string_can_append_iter< OutStringType, Arg >::value &&detect_string_↩
can_append_data< OutStringType, Arg >::value, int > = 0>
void detail::concat_into (
            OutStringType & out,
            const Arg & arg,
            Args &&...  rest)  [inline]
```

Definition at line 4517 of file json.hpp.

#### 8.1.4.7 concat_into() [5/5]

```
template<typename OutStringType, typename Arg, typename...  Args, enable_if_t< !detect_↩
string_can_append< OutStringType, Arg >::value &&!detect_string_can_append_op< OutStringType,
Arg >::value &&detect_string_can_append_iter< OutStringType, Arg >::value, int > = 0>
void detail::concat_into (
            OutStringType & out,
            const Arg & arg,
            Args &&...  rest)  [inline]
```

Definition at line 4506 of file json.hpp.

#### 8.1.4.8 concat_length() [1/4]

```
std::size_t detail::concat_length ()  [inline]
```

Definition at line 4409 of file json.hpp.

#### 8.1.4.9 concat_length() [2/4]

```
template<typename...  Args>
std::size_t detail::concat_length (
            const char * cstr,
            const Args &...  rest)  [inline]
```

Definition at line 4427 of file json.hpp.

#### 8.1.4.10 concat_length() [3/4]

```
template<typename...  Args>
std::size_t detail::concat_length (
            const char ,
            const Args &...  rest)  [inline]
```

Definition at line 4421 of file json.hpp.

#### 8.1.4.11 concat_length() [4/4]

```
template<typename StringType, typename...  Args>
std::size_t detail::concat_length (
            const StringType & str,
            const Args &...  rest)  [inline]
```

Definition at line 4434 of file json.hpp.

#### 8.1.4.12 conditional_static_cast() [1/2]

```
template<typename T, typename U, enable_if_t< std::is_same< T, U >::value, int > = 0>
T detail::conditional_static_cast (
            U value)
```

Definition at line 4231 of file json.hpp.

#### 8.1.4.13 conditional_static_cast() [2/2]

```
template<typename T, typename U, enable_if_t< !std::is_same< T, U >::value, int > = 0>
T detail::conditional_static_cast (
            U value)
```

Definition at line 4225 of file json.hpp.

#### 8.1.4.14 escape()

```
template<typename StringType>
StringType detail::escape (
            StringType s)  [inline]
```

string escaping as described in RFC 6901 (Sect. 4)

**Parameters**

| | | |
|---|---|---|
| in | *s* | string to escape |

**Returns**

escaped string

Note the order of escaping "~" to "~0" and "/" to "~1" is important.

Definition at line 3121 of file json.hpp.

#### 8.1.4.15 from_json() [1/22]

```
template<typename BasicJsonType, typename T, std::size_t N>
auto detail::from_json (
            BasicJsonType && j,
            identity_tag< std::array< T, N > > tag) -> decltype(from_json_inplace_array_↩
impl(std::forward< BasicJsonType >(j), tag, make_index_sequence< N > {}))
```

Definition at line 5193 of file json.hpp.

#### 8.1.4.16 from_json() [2/22]

```
template<typename BasicJsonType, typename TupleRelated>
auto detail::from_json (
            BasicJsonType && j,
            TupleRelated && t) -> decltype(from_json_tuple_impl(std::forward< BasicJsonType
>(j), std::forward< TupleRelated >(t), priority_tag< 3 > {}))
```

Definition at line 5323 of file json.hpp.

#### 8.1.4.17 from_json() [3/22]

```
template<typename BasicJsonType, typename ArithmeticType, enable_if_t< std::is_arithmetic<
ArithmeticType >::value &&!std::is_same< ArithmeticType, typename BasicJsonType::number_↩
unsigned_t >::value &&!std::is_same< ArithmeticType, typename BasicJsonType::number_integer↩
_t >::value &&!std::is_same< ArithmeticType, typename BasicJsonType::number_float_t >::value
&&!std::is_same< ArithmeticType, typename BasicJsonType::boolean_t >::value, int > = 0>
void detail::from_json (
            const BasicJsonType & j,
            ArithmeticType & val)  [inline]
```

Definition at line 5249 of file json.hpp.

#### 8.1.4.18 from_json() [4/22]

```
template<typename BasicJsonType, typename ConstructibleArrayType, enable_if_t< is_constructible_array_type<
BasicJsonType, ConstructibleArrayType >::value &&!is_constructible_object_type< BasicJsonType,
ConstructibleArrayType >::value &&!is_constructible_string_type< BasicJsonType, Constructible↩
ArrayType >::value &&!std::is_same< ConstructibleArrayType, typename BasicJsonType::binary_t
>::value &&!is_basic_json< ConstructibleArrayType >::value, int > = 0>
auto detail::from_json (
             const BasicJsonType & j,
             ConstructibleArrayType & arr) -> decltype(from_json_array_impl(j, arr, priority_tag<
3 > {}), j.template get< typename ConstructibleArrayType::value_type >(), void())
```

Definition at line 5172 of file json.hpp.

#### 8.1.4.19 from_json() [5/22]

```
template<typename BasicJsonType, typename ConstructibleObjectType, enable_if_t< is_constructible_object_type<
BasicJsonType, ConstructibleObjectType >::value, int > = 0>
void detail::from_json (
             const BasicJsonType & j,
             ConstructibleObjectType & obj)  [inline]
```

Definition at line 5217 of file json.hpp.

#### 8.1.4.20 from_json() [6/22]

```
template<typename BasicJsonType, typename EnumType, enable_if_t< std::is_enum< EnumType >::↩
value, int > = 0>
void detail::from_json (
             const BasicJsonType & j,
             EnumType & e)  [inline]
```

Definition at line 5002 of file json.hpp.

#### 8.1.4.21 from_json() [7/22]

```
template<typename BasicJsonType, typename T, typename Allocator, enable_if_t< is_getable<
BasicJsonType, T >::value, int > = 0>
void detail::from_json (
             const BasicJsonType & j,
             std::forward_list< T, Allocator > & l)  [inline]
```

Definition at line 5013 of file json.hpp.

#### 8.1.4.22 from_json() [8/22]

```
template<typename BasicJsonType, typename Key, typename Value, typename Compare, typename
Allocator, typename = enable_if_t < !std::is_constructible < typename BasicJsonType::string_t,
Key >::value >>
void detail::from_json (
             const BasicJsonType & j,
             std::map< Key, Value, Compare, Allocator > & m)  [inline]
```

Definition at line 5337 of file json.hpp.

#### 8.1.4.23 from_json() [9/22]

```
template<typename BasicJsonType, typename Key, typename Value, typename Hash, typename Key↩
Equal, typename Allocator, typename = enable_if_t < !std::is_constructible < typename Basic↩
JsonType::string_t, Key >::value >>
void detail::from_json (
            const BasicJsonType & j,
            std::unordered_map< Key, Value, Hash, KeyEqual, Allocator > & m)  [inline]
```

Definition at line 5357 of file json.hpp.

#### 8.1.4.24 from_json() [10/22]

```
template<typename BasicJsonType, typename T, enable_if_t< is_getable< BasicJsonType, T >::↩
value, int > = 0>
void detail::from_json (
            const BasicJsonType & j,
            std::valarray< T > & l)  [inline]
```

Definition at line 5030 of file json.hpp.

#### 8.1.4.25 from_json() [11/22]

```
template<typename BasicJsonType, typename StringType, enable_if_t< std::is_assignable< String↩
Type &, const typename BasicJsonType::string_t >::value &&is_detected_exact< typename Basic↩
JsonType::string_t::value_type, value_type_t, StringType >::value &&!std::is_same< typename
BasicJsonType::string_t, StringType >::value &&!is_json_ref< StringType >::value, int > = 0>
void detail::from_json (
            const BasicJsonType & j,
            StringType & s)  [inline]
```

Definition at line 4971 of file json.hpp.

#### 8.1.4.26 from_json() [12/22]

```
template<typename BasicJsonType, typename T, std::size_t N1, std::size_t N2>
auto detail::from_json (
            const BasicJsonType & j,
            T(&) arr[N1][N2]) -> decltype(j.template get< T >(), void())
```

Definition at line 5055 of file json.hpp.

#### 8.1.4.27 from_json() [13/22]

```
template<typename BasicJsonType, typename T, std::size_t N1, std::size_t N2, std::size_t N3>
auto detail::from_json (
            const BasicJsonType & j,
            T(&) arr[N1][N2][N3]) -> decltype(j.template get< T >(), void())
```

Definition at line 5068 of file json.hpp.

#### 8.1.4.28 from_json() [14/22]

```
template<typename BasicJsonType, typename T, std::size_t N1, std::size_t N2, std::size_t N3,
std::size_t N4>
auto detail::from_json (
            const BasicJsonType & j,
            T(&) arr[N1][N2][N3][N4]) -> decltype(j.template get< T >(), void())
```

Definition at line 5084 of file json.hpp.

#### 8.1.4.29 from_json() [15/22]

```
template<typename BasicJsonType, typename T, std::size_t N>
auto detail::from_json (
            const BasicJsonType & j,
            T(&) arr[N]) -> decltype(j.template get< T >(), void())
```

Definition at line 5045 of file json.hpp.

#### 8.1.4.30 from_json() [16/22]

```
template<typename BasicJsonType>
void detail::from_json (
            const BasicJsonType & j,
            typename BasicJsonType::binary_t & bin)  [inline]
```

Definition at line 5205 of file json.hpp.

#### 8.1.4.31 from_json() [17/22]

```
template<typename BasicJsonType>
void detail::from_json (
            const BasicJsonType & j,
            typename BasicJsonType::boolean_t & b)  [inline]
```

Definition at line 4945 of file json.hpp.

#### 8.1.4.32 from_json() [18/22]

```
template<typename BasicJsonType>
void detail::from_json (
            const BasicJsonType & j,
            typename BasicJsonType::number_float_t & val)  [inline]
```

Definition at line 4982 of file json.hpp.

#### 8.1.4.33 from_json() [19/22]

```
template<typename BasicJsonType>
void detail::from_json (
            const BasicJsonType & j,
            typename BasicJsonType::number_integer_t & val)  [inline]
```

Definition at line 4994 of file json.hpp.

#### 8.1.4.34 from_json() [20/22]

```
template<typename BasicJsonType>
void detail::from_json (
            const BasicJsonType & j,
            typename BasicJsonType::number_unsigned_t & val)  [inline]
```

Definition at line 4988 of file json.hpp.

#### 8.1.4.35 from_json() [21/22]

```
template<typename BasicJsonType>
void detail::from_json (
            const BasicJsonType & j,
            typename BasicJsonType::string_t & s)  [inline]
```

Definition at line 4955 of file json.hpp.

#### 8.1.4.36 from_json() [22/22]

```
template<typename BasicJsonType>
void detail::from_json (
            const BasicJsonType & j,
            typename std::nullptr_t & n)  [inline]
```

Definition at line 4882 of file json.hpp.

#### 8.1.4.37 from_json_array_impl() [1/4]

```
template<typename BasicJsonType, typename ConstructibleArrayType, enable_if_t< std::is_↩
assignable< ConstructibleArrayType &, ConstructibleArrayType >::value, int > = 0>
void detail::from_json_array_impl (
            const BasicJsonType & j,
            ConstructibleArrayType & arr,
            priority_tag< 0 > )  [inline]
```

Definition at line 5147 of file json.hpp.

#### 8.1.4.38 from_json_array_impl() [2/4]

```
template<typename BasicJsonType, typename ConstructibleArrayType, enable_if_t< std::is_↩
assignable< ConstructibleArrayType &, ConstructibleArrayType >::value, int > = 0>
auto detail::from_json_array_impl (
             const BasicJsonType & j,
             ConstructibleArrayType & arr,
             priority_tag< 1 > ) -> decltype( arr.reserve(std::declval< typename Constructible↩
ArrayType::size_type >()), j.template get< typename ConstructibleArrayType::value_type >(),
void())
```

Definition at line 5123 of file json.hpp.

#### 8.1.4.39 from_json_array_impl() [3/4]

```
template<typename BasicJsonType, typename T, std::size_t N>
auto detail::from_json_array_impl (
             const BasicJsonType & j,
             std::array< T, N > & arr,
             priority_tag< 2 > ) -> decltype(j.template get< T >(), void())
```

Definition at line 5109 of file json.hpp.

#### 8.1.4.40 from_json_array_impl() [4/4]

```
template<typename BasicJsonType>
void detail::from_json_array_impl (
             const BasicJsonType & j,
             typename BasicJsonType::array_t & arr,
             priority_tag< 3 > )  [inline]
```

Definition at line 5103 of file json.hpp.

#### 8.1.4.41 from_json_inplace_array_impl()

```
template<typename BasicJsonType, typename T, std::size_t...  Idx>
std::array< T, sizeof...(Idx)> detail::from_json_inplace_array_impl (
             BasicJsonType && j,
             identity_tag< std::array< T, sizeof...(Idx)> > ,
             index_sequence< Idx...  > )
```

Definition at line 5186 of file json.hpp.

#### 8.1.4.42 from_json_tuple_impl() [1/4]

```
template<typename BasicJsonType, class A1, class A2>
std::pair< A1, A2 > detail::from_json_tuple_impl (
             BasicJsonType && j,
             identity_tag< std::pair< A1, A2 > > ,
             priority_tag< 0 > )
```

Definition at line 5298 of file json.hpp.

#### 8.1.4.43 from_json_tuple_impl() [2/4]

```
template<typename BasicJsonType, typename... Args>
std::tuple< Args... > detail::from_json_tuple_impl (
             BasicJsonType && j,
             identity_tag< std::tuple< Args... > > ,
             priority_tag< 2 > )
```

Definition at line 5311 of file json.hpp.

#### 8.1.4.44 from_json_tuple_impl() [3/4]

```
template<typename BasicJsonType, typename A1, typename A2>
void detail::from_json_tuple_impl (
             BasicJsonType && j,
             std::pair< A1, A2 > & p,
             priority_tag< 1 > ) [inline]
```

Definition at line 5305 of file json.hpp.

#### 8.1.4.45 from_json_tuple_impl() [4/4]

```
template<typename BasicJsonType, typename... Args>
void detail::from_json_tuple_impl (
             BasicJsonType && j,
             std::tuple< Args... > & t,
             priority_tag< 3 > ) [inline]
```

Definition at line 5317 of file json.hpp.

#### 8.1.4.46 from_json_tuple_impl_base() [1/2]

```
template<typename BasicJsonType, typename... Args, std::size_t... Idx>
std::tuple< Args... > detail::from_json_tuple_impl_base (
             BasicJsonType && j,
             index_sequence< Idx... > )
```

Definition at line 5286 of file json.hpp.

#### 8.1.4.47 from_json_tuple_impl_base() [2/2]

```
template<typename BasicJsonType>
std::tuple detail::from_json_tuple_impl_base (
             BasicJsonType & ,
             index_sequence<> )
```

Definition at line 5292 of file json.hpp.

#### 8.1.4.48 get() [1/2]

```
template<std::size_t N, typename IteratorType, enable_if_t< N==0, int > = 0>
auto detail::get (
            const nlohmann::detail::iteration_proxy_value< IteratorType > & i) -> decltype(i.↩
key())
```

Definition at line 5677 of file json.hpp.

#### 8.1.4.49 get() [2/2]

```
template<std::size_t N, typename IteratorType, enable_if_t< N==1, int > = 0>
auto detail::get (
            const nlohmann::detail::iteration_proxy_value< IteratorType > & i) -> decltype(i.↩
value())
```

Definition at line 5685 of file json.hpp.

#### 8.1.4.50 get_arithmetic_value()

```
template<typename BasicJsonType, typename ArithmeticType, enable_if_t< std::is_arithmetic<
ArithmeticType >::value &&!std::is_same< ArithmeticType, typename BasicJsonType::boolean_↩
t >::value, int > = 0>
void detail::get_arithmetic_value (
            const BasicJsonType & j,
            ArithmeticType & val)
```

Definition at line 4912 of file json.hpp.

#### 8.1.4.51 hash()

```
template<typename BasicJsonType>
std::size_t detail::hash (
            const BasicJsonType & j)
```

hash a JSON value

The hash function tries to rely on std::hash where possible. Furthermore, the type of the JSON value is taken into account to have different hash values for null, 0, 0U, and false, etc.

**Template Parameters**

| | |
|---|---|
| *BasicJsonType* | basic_json specialization |

**Parameters**

| | |
|---|---|
| *j* | JSON value to hash |

**Returns**

hash value of j

Definition at line 6390 of file json.hpp.

#### 8.1.4.52 input_adapter() [1/7]

```
template<typename CharT, typename std::enable_if< std::is_pointer< CharT >::value &&!std::↩
is_array< CharT >::value &&std::is_integral< typename std::remove_pointer< CharT >::type
>::value &&sizeof(typename std::remove_pointer< CharT >::type)==1, int >::type = 0>
contiguous_bytes_input_adapter detail::input_adapter (
            CharT b)
```

Definition at line 7014 of file json.hpp.

#### 8.1.4.53 input_adapter() [2/7]

```
template<typename ContainerType>
container_input_adapter_factory_impl::container_input_adapter_factory< ContainerType >::↩
adapter_type detail::input_adapter (
            const ContainerType & container)
```

Definition at line 6974 of file json.hpp.

#### 8.1.4.54 input_adapter() [3/7]

```
template<typename IteratorType>
iterator_input_adapter_factory< IteratorType >::adapter_type detail::input_adapter (
            IteratorType first,
            IteratorType last)
```

Definition at line 6940 of file json.hpp.

#### 8.1.4.55 input_adapter() [4/7]

```
file_input_adapter detail::input_adapter (
            std::FILE * file)  [inline]
```

Definition at line 6984 of file json.hpp.

#### 8.1.4.56 input_adapter() [5/7]

```
input_stream_adapter detail::input_adapter (
            std::istream && stream)  [inline]
```

Definition at line 6998 of file json.hpp.

#### 8.1.4.57 input_adapter() [6/7]

```
input_stream_adapter detail::input_adapter (
            std::istream & stream)  [inline]
```

Definition at line 6993 of file json.hpp.

#### 8.1.4.58 input_adapter() [7/7]

```
template<typename T, std::size_t N>
auto detail::input_adapter (
             T(&) array[N]) -> decltype(input_adapter(array, array+N))
```

Definition at line 7026 of file json.hpp.

#### 8.1.4.59 int_to_string()

```
template<typename StringType>
void detail::int_to_string (
             StringType & target,
             std::size_t value)
```

Definition at line 5495 of file json.hpp.

#### 8.1.4.60 little_endianness()

```
bool detail::little_endianness (
             int num = 1)  [inline], [noexcept]
```

determine system byte order

**Returns**

true if and only if system's byte order is little endian

**Note**

from https://stackoverflow.com/a/1001328/266378

Definition at line 9898 of file json.hpp.

#### 8.1.4.61 make_array()

```
template<typename T, typename...  Args>
std::array< T, sizeof...(Args)> detail::make_array (
             Args &&...  args)  [constexpr]
```

Definition at line 3353 of file json.hpp.

#### 8.1.4.62 operator<()

```
bool 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 < binary
- furthermore, each type is not smaller than itself
- discarded values are not comparable
- binary is represented as a b"" string in python and directly comparable to a string; however, making a binary array directly comparable with a string would be surprising behavior in a JSON file.

**Since**

version 1.0.0

Definition at line 3033 of file json.hpp.

#### 8.1.4.63 replace_substring()

```
template<typename StringType>
void detail::replace_substring (
            StringType & s,
            const StringType & f,
            const StringType & t)  [inline]
```

replace all occurrences of a substring by another string

**Parameters**

| | | |
|---|---|---|
| in,out | *s* | the string to manipulate; changed so that all occurrences of *f* are replaced with *t* |
| in | *f* | the substring to replace with *t* |
| in | *t* | the string to replace *f* |

**Precondition**

The search string *f* must not be empty. **This precondition is enforced with an assertion.**

**Since**

version 2.0.0

Definition at line 3102 of file json.hpp.

#### 8.1.4.64 to_chars()

```
template<typename FloatType>
JSON_HEDLEY_RETURNS_NON_NULL char * detail::to_chars (
            char * first,
            const char * last,
            FloatType value)
```

generates a decimal representation of the floating-point number value in [first, last).

The format of the resulting decimal representation is similar to printf's g format. Returns an iterator pointing past-the-end of the decimal representation.

**Note**

The input number must be finite, i.e. NaN's and Inf's are not supported.

The buffer must be large enough.

The result is NOT null-terminated.

Definition at line 18800 of file json.hpp.

#### 8.1.4.65 to_json() [1/19]

```
template<typename BasicJsonType, typename CompatibleNumberIntegerType, enable_if_t< is_compatible_integer_typ
typename BasicJsonType::number_integer_t, CompatibleNumberIntegerType >::value, int > = 0>
void detail::to_json (
            BasicJsonType & j,
            CompatibleNumberIntegerType val)  [inline], [noexcept]
```

Definition at line 6038 of file json.hpp.

#### 8.1.4.66 to_json() [2/19]

```
template<typename BasicJsonType, typename CompatibleNumberUnsignedType, enable_if_t< is_compatible_integer_ty
typename BasicJsonType::number_unsigned_t, CompatibleNumberUnsignedType >::value, int > = 0>
void detail::to_json (
            BasicJsonType & j,
            CompatibleNumberUnsignedType val)  [inline], [noexcept]
```

Definition at line 6031 of file json.hpp.

#### 8.1.4.67 to_json() [3/19]

```
template<typename BasicJsonType, typename BoolRef, enable_if_t<((std::is_same< std::vector<
bool >::reference, BoolRef >::value &&!std::is_same< std::vector< bool >::reference, typename
BasicJsonType::boolean_t & >::value)||(std::is_same< std::vector< bool >::const_reference,
BoolRef >::value &&!std::is_same< detail::uncvref_t< std::vector< bool >::const_reference >,
typename BasicJsonType::boolean_t >::value)) &&std::is_convertible< const BoolRef &, typename
BasicJsonType::boolean_t >::value, int > = 0>
void detail::to_json (
            BasicJsonType & j,
            const BoolRef & b)  [inline], [noexcept]
```

Definition at line 6004 of file json.hpp.

#### 8.1.4.68 to_json() [4/19]

```
template<typename BasicJsonType, typename CompatibleArrayType, enable_if_t< is_compatible_array_type<
BasicJsonType, CompatibleArrayType >::value &&!is_compatible_object_type< BasicJsonType,
CompatibleArrayType >::value &&!is_compatible_string_type< BasicJsonType, CompatibleArray↩
Type >::value &&!std::is_same< typename BasicJsonType::binary_t, CompatibleArrayType >::value
&&!is_basic_json< CompatibleArrayType >::value, int > = 0>
void detail::to_json (
            BasicJsonType & j,
            const CompatibleArrayType & arr)  [inline]
```

Definition at line 6068 of file json.hpp.

#### 8.1.4.69 to_json() [5/19]

```
template<typename BasicJsonType, typename CompatibleObjectType, enable_if_t< is_compatible_object_type<
BasicJsonType, CompatibleObjectType >::value &&!is_basic_json< CompatibleObjectType >::value,
int > = 0>
void detail::to_json (
            BasicJsonType & j,
            const CompatibleObjectType & obj)  [inline]
```

Definition at line 6094 of file json.hpp.

#### 8.1.4.70 to_json() [6/19]

```
template<typename BasicJsonType, typename CompatibleString, enable_if_t< std::is_constructible<
typename BasicJsonType::string_t, CompatibleString >::value, int > = 0>
void detail::to_json (
            BasicJsonType & j,
            const CompatibleString & s)  [inline]
```

Definition at line 6011 of file json.hpp.

#### 8.1.4.71 to_json() [7/19]

```
template<typename BasicJsonType, typename T1, typename T2, enable_if_t< std::is_constructible<
BasicJsonType, T1 >::value &&std::is_constructible< BasicJsonType, T2 >::value, int > = 0>
void detail::to_json (
            BasicJsonType & j,
            const std::pair< T1, T2 > & p)  [inline]
```

Definition at line 6116 of file json.hpp.

#### 8.1.4.72 to_json() [8/19]

```
template<typename BasicJsonType, typename T, enable_if_t< std::is_convertible< T, BasicJson↩
Type >::value, int > = 0>
void detail::to_json (
            BasicJsonType & j,
            const std::valarray< T > & arr)  [inline]
```

Definition at line 6081 of file json.hpp.

#### 8.1.4.73 to_json() [9/19]

```
template<typename BasicJsonType>
void detail::to_json (
            BasicJsonType & j,
            const std::vector< bool > & e)  [inline]
```

Definition at line 6055 of file json.hpp.

#### 8.1.4.74 to_json() [10/19]

```
template<typename BasicJsonType, typename T, enable_if_t< std::is_same< T, iteration_proxy_value<
typename BasicJsonType::iterator > >::value, int > = 0>
void detail::to_json (
            BasicJsonType & j,
            const T & b)  [inline]
```

Definition at line 6124 of file json.hpp.

#### 8.1.4.75 to_json() [11/19]

```
template<typename BasicJsonType, typename T, enable_if_t< is_constructible_tuple< BasicJson↩
Type, T >::value, int > = 0>
void detail::to_json (
            BasicJsonType & j,
            const T & t)  [inline]
```

Definition at line 6143 of file json.hpp.

#### 8.1.4.76 to_json() [12/19]

```
template<typename BasicJsonType, typename T, std::size_t N, enable_if_t< !std::is_constructible<
typename BasicJsonType::string_t, const T(&)[N]>::value, int > = 0>
void detail::to_json (
            BasicJsonType & j,
            const T(&) arr[N])  [inline]
```

Definition at line 6110 of file json.hpp.

#### 8.1.4.77 to_json() [13/19]

```
template<typename BasicJsonType>
void detail::to_json (
            BasicJsonType & j,
            const typename BasicJsonType::binary_t & bin)  [inline]
```

Definition at line 6074 of file json.hpp.

#### 8.1.4.78 to_json() [14/19]

```
template<typename BasicJsonType, typename EnumType, enable_if_t< std::is_enum< EnumType >::↩
value, int > = 0>
void detail::to_json (
            BasicJsonType & j,
            EnumType e)  [inline], [noexcept]
```

Definition at line 6046 of file json.hpp.

#### 8.1.4.79 to_json() [15/19]

```
template<typename BasicJsonType, typename FloatType, enable_if_t< std::is_floating_point<
FloatType >::value, int > = 0>
void detail::to_json (
            BasicJsonType & j,
            FloatType val)  [inline], [noexcept]
```

Definition at line 6024 of file json.hpp.

#### 8.1.4.80 to_json() [16/19]

```
template<typename BasicJsonType, typename T, enable_if_t< std::is_same< T, typename Basic↩
JsonType::boolean_t >::value, int > = 0>
void detail::to_json (
            BasicJsonType & j,
            T b)  [inline], [noexcept]
```

Definition at line 5991 of file json.hpp.

#### 8.1.4.81 to_json() [17/19]

```
template<typename BasicJsonType>
void detail::to_json (
            BasicJsonType & j,
            typename BasicJsonType::array_t && arr)  [inline]
```

Definition at line 6087 of file json.hpp.

#### 8.1.4.82 to_json() [18/19]

```
template<typename BasicJsonType>
void detail::to_json (
            BasicJsonType & j,
            typename BasicJsonType::object_t && obj)  [inline]
```

Definition at line 6100 of file json.hpp.

#### 8.1.4.83 to_json() [19/19]

```
template<typename BasicJsonType>
void detail::to_json (
            BasicJsonType & j,
            typename BasicJsonType::string_t && s)  [inline]
```

Definition at line 6017 of file json.hpp.

#### 8.1.4.84 to_json_tuple_impl() [1/2]

```
template<typename BasicJsonType, typename Tuple>
void detail::to_json_tuple_impl (
            BasicJsonType & j,
            const Tuple & ,
            index_sequence<> )  [inline]
```

Definition at line 6136 of file json.hpp.

#### 8.1.4.85 to_json_tuple_impl() [2/2]

```
template<typename BasicJsonType, typename Tuple, std::size_t...  Idx>
void detail::to_json_tuple_impl (
            BasicJsonType & j,
            const Tuple & t,
            index_sequence< Idx...  > )  [inline]
```

Definition at line 6130 of file json.hpp.

#### 8.1.4.86 to_string()

```
template<typename StringType>
StringType detail::to_string (
            std::size_t value)
```

Definition at line 5503 of file json.hpp.

#### 8.1.4.87 unescape()

```
template<typename StringType>
void detail::unescape (
            StringType & s)  [inline]
```

string unescaping as described in RFC 6901 (Sect. 4)

**Parameters**

| | | |
|---|---|---|
| in | *s* | string to unescape |

**Returns**

unescaped string

Note the order of escaping "~1" to "/" and "~0" to "~" is important.

Definition at line 3136 of file json.hpp.

#### 8.1.4.88 unknown_size()

```
std::size_t detail::unknown_size ()  [constexpr]
```

Definition at line 8864 of file json.hpp.

#### 8.1.4.89 value_in_range_of()

```
template<typename OfType, typename T>
bool detail::value_in_range_of (
            T val)  [constexpr]
```

Definition at line 4325 of file json.hpp.

### 8.1.5 Variable Documentation

#### 8.1.5.1 binary_reader< BasicJsonType, InputAdapterType, SAX >::npos

```
template<typename BasicJsonType, typename InputAdapterType, typename SAX>
std::size_t detail::binary_reader< BasicJsonType, InputAdapterType, SAX >::npos  [constexpr]
```

Definition at line 12925 of file json.hpp.

#### 8.1.5.2 static_const< T >::value

```
template<typename T>
T detail::static_const< T >::value  [constexpr]
```

Definition at line 3349 of file json.hpp.

## 8.2 detail::dtoa_impl Namespace Reference

implements the Grisu2 algorithm for binary to decimal floating-point conversion.

#### Classes

- struct diyfp
- struct boundaries
- struct cached_power

### Functions

- template<typename Target, typename Source>
  Target reinterpret_bits (const Source source)
- template<typename FloatType>
  boundaries compute_boundaries (FloatType value)
- cached_power get_cached_power_for_binary_exponent (int e)
- int find_largest_pow10 (const std::uint32_t n, std::uint32_t &pow10)
- void grisu2_round (char *buf, int len, std::uint64_t dist, std::uint64_t delta, std::uint64_t rest, std::uint64_t ten_k)
- void grisu2_digit_gen (char *buffer, int &length, int &decimal_exponent, diyfp M_minus, diyfp w, diyfp M_plus)
- void grisu2 (char *buf, int &len, int &decimal_exponent, diyfp m_minus, diyfp v, diyfp m_plus)
- template<typename FloatType>
  void grisu2 (char *buf, int &len, int &decimal_exponent, FloatType value)
- JSON_HEDLEY_RETURNS_NON_NULL char * append_exponent (char *buf, int e)

  *appends a decimal representation of e to buf*
- JSON_HEDLEY_RETURNS_NON_NULL char * format_buffer (char *buf, int len, int decimal_exponent, int min_exp, int max_exp)

  *prettify v = buf * 10^decimal_exponent*

### Variables

- constexpr int kAlpha = -60
- constexpr int kGamma = -32

## 8.2.1 Detailed Description

implements the Grisu2 algorithm for binary to decimal floating-point conversion.

This implementation is a slightly modified version of the reference implementation which may be obtained from http://florian.loitsch.com/publications (bench.tar.gz).

The code is distributed under the MIT license, Copyright (c) 2009 Florian Loitsch.

For a detailed description of the algorithm see:

[1] Loitsch, "Printing Floating-Point Numbers Quickly and Accurately with Integers", Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation, PLDI 2010 [2] Burger, Dybvig, "Printing Floating-Point Numbers Quickly and Accurately", Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation, PLDI 1996

## 8.2.2 Function Documentation

### 8.2.2.1 append_exponent()

```
JSON_HEDLEY_RETURNS_NON_NULL char * detail::dtoa_impl::append_exponent (
            char * buf,
            int e)  [inline]
```

appends a decimal representation of e to buf

**Returns**

a pointer to the element following the exponent.

**Precondition**

-1000 < e < 1000

Definition at line 18663 of file json.hpp.

#### 8.2.2.2 compute_boundaries()

```
template<typename FloatType>
boundaries detail::dtoa_impl::compute_boundaries (
            FloatType value)
```

Compute the (normalized) diyfp representing the input number 'value' and its boundaries.

**Precondition**

value must be finite and positive

Definition at line 17924 of file json.hpp.

#### 8.2.2.3 find_largest_pow10()

```
int detail::dtoa_impl::find_largest_pow10 (
            const std::uint32_t n,
            std::uint32_t & pow10)  [inline]
```

For n != 0, returns k, such that pow10 := 10^(k-1) <= n < 10^k. For n == 0, returns 1 and sets pow10 := 1.

Definition at line 18227 of file json.hpp.

#### 8.2.2.4 format_buffer()

```
JSON_HEDLEY_RETURNS_NON_NULL char * detail::dtoa_impl::format_buffer (
            char * buf,
            int len,
            int decimal_exponent,
            int min_exp,
            int max_exp)  [inline]
```

prettify v = buf * 10^decimal_exponent

If v is in the range [10^min_exp, 10^max_exp) it will be printed in fixed-point notation. Otherwise it will be printed in exponential notation.

**Precondition**

min_exp < 0

max_exp > 0

Definition at line 18715 of file json.hpp.

#### 8.2.2.5 get_cached_power_for_binary_exponent()

```
cached_power detail::dtoa_impl::get_cached_power_for_binary_exponent (
            int e)  [inline]
```

For a normalized diyfp w = f * 2^e, this function returns a (normalized) cached power-of-ten c = f_c * 2^e_c, such that the exponent of the product w * c satisfies (Definition 3.2 from [1])

```
 alpha <= e_c + e + q <= gamma.
```

Definition at line 18063 of file json.hpp.

#### 8.2.2.6 grisu2() [1/2]

```
void detail::dtoa_impl::grisu2 (
            char * buf,
            int & len,
            int & decimal_exponent,
            diyfp m_minus,
            diyfp v,
            diyfp m_plus)  [inline]
```

v = buf * 10^decimal_exponent len is the length of the buffer (number of decimal digits) The buffer must be large enough, i.e. >= max_digits10.

Definition at line 18563 of file json.hpp.

#### 8.2.2.7 grisu2() [2/2]

```
template<typename FloatType>
void detail::dtoa_impl::grisu2 (
            char * buf,
            int & len,
            int & decimal_exponent,
            FloatType value)
```

v = buf * 10^decimal_exponent len is the length of the buffer (number of decimal digits) The buffer must be large enough, i.e. >= max_digits10.

Definition at line 18623 of file json.hpp.

#### 8.2.2.8 grisu2_digit_gen()

```
void detail::dtoa_impl::grisu2_digit_gen (
            char * buffer,
            int & length,
            int & decimal_exponent,
            diyfp M_minus,
            diyfp w,
            diyfp M_plus)  [inline]
```

Generates V = buffer * 10^decimal_exponent, such that M- <= V <= M+. M- and M+ must be normalized and share the same exponent -60 <= e <= -32.

Definition at line 18322 of file json.hpp.

#### 8.2.2.9 grisu2_round()

```
void detail::dtoa_impl::grisu2_round (
            char * buf,
            int len,
            std::uint64_t dist,
            std::uint64_t delta,
            std::uint64_t rest,
            std::uint64_t ten_k)  [inline]
```

Definition at line 18281 of file json.hpp.

#### 8.2.2.10 reinterpret_bits()

```
template<typename Target, typename Source>
Target detail::dtoa_impl::reinterpret_bits (
            const Source source)
```

Definition at line 17783 of file json.hpp.

### 8.2.3 Variable Documentation

#### 8.2.3.1 kAlpha

```
int detail::dtoa_impl::kAlpha = -60  [constexpr]
```

Definition at line 18046 of file json.hpp.

#### 8.2.3.2 kGamma

```
int detail::dtoa_impl::kGamma = -32  [constexpr]
```

Definition at line 18047 of file json.hpp.

# Chapter 9

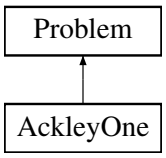
# Class Documentation

## 9.1 AckleyOne Class Reference

Implements the Ackley 1 benchmark function.

`#include <AckleyOne.h>`

Inheritance diagram for AckleyOne:




### Public Member Functions

- double evaluate (const std::vector< double > &x) const override

  *Evaluates the fitness of a candidate solution.*

### Public Member Functions inherited from Problem

- Problem (double lb, double ub, const std::string_view n)

  *Constructs a Problem instance.*
- virtual ~**Problem** ()=default

  *Virtual destructor for safe polymorphic cleanup.*

- double getLowerBound () const
- double getUpperBound () const
- const std::string getName () const

**Additional Inherited Members**

## Protected Attributes inherited from Problem

- const double lowerBound

  *Lower bound of the search space.*
- const double upperBound

  *Upper bound of the search space.*
- const std::string name

  *Name of the benchmark function.*

### 9.1.1 Detailed Description

Implements the Ackley 1 benchmark function.

Definition at line 21 of file AckleyOne.h.

### 9.1.2 Constructor & Destructor Documentation

#### 9.1.2.1 AckleyOne()

```
AckleyOne::AckleyOne () [inline]
```

Definition at line 28 of file AckleyOne.h.

### 9.1.3 Member Function Documentation

#### 9.1.3.1 evaluate()

```
double AckleyOne::evaluate (
            const std::vector< double > & x) const [inline], [override], [virtual]
```

Evaluates the fitness of a candidate solution.

- This is a pure virtual function that must be implemented by specific benchmark functions (e.g., Ackley, Rosenbrock).
- **Parameters**

| | |
|---|---|
| *x* | The solution vector to evaluate. |

**Returns**

The scalar fitness value (cost).

Implements Problem.

Definition at line 30 of file AckleyOne.h.

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

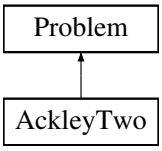
- include/Problem/AckleyOne.h

## 9.2 AckleyTwo Class Reference

Implements the Ackley 2 benchmark function.

`#include <AckleyTwo.h>`

Inheritance diagram for AckleyTwo:




### Public Member Functions

- double evaluate (const std::vector< double > &x) const override

  *Evaluates the fitness of a candidate solution.*

### Public Member Functions inherited from Problem

- Problem (double lb, double ub, const std::string_view n)

  *Constructs a Problem instance.*
- virtual ~**Problem** ()=default

  *Virtual destructor for safe polymorphic cleanup.*

- double getLowerBound () const
- double getUpperBound () const
- const std::string getName () const

### Additional Inherited Members

### Protected Attributes inherited from Problem

- const double lowerBound

  *Lower bound of the search space.*
- const double upperBound

  *Upper bound of the search space.*
- const std::string name

  *Name of the benchmark function.*

### 9.2.1 Detailed Description

Implements the Ackley 2 benchmark function.

Definition at line 22 of file AckleyTwo.h.

### 9.2.2 Constructor & Destructor Documentation

#### 9.2.2.1 AckleyTwo()

```
AckleyTwo::AckleyTwo ()  [inline]
```

Definition at line 29 of file AckleyTwo.h.

### 9.2.3 Member Function Documentation

#### 9.2.3.1 evaluate()

```
double AckleyTwo::evaluate (
            const std::vector< double > & x) const  [inline], [override], [virtual]
```

Evaluates the fitness of a candidate solution.

- This is a pure virtual function that must be implemented by specific benchmark functions (e.g., Ackley, Rosenbrock).
- **Parameters**

| | |
|---|---|
| *x* | The solution vector to evaluate. |

**Returns**

The scalar fitness value (cost).

Implements Problem.

Definition at line 31 of file AckleyTwo.h.

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

- include/Problem/AckleyTwo.h

## 9.3 detail::actual_object_comparator< BasicJsonType > Struct Template Reference

**Public Types**

- using object_t = typename BasicJsonType::object_t
- using object_comparator_t = typename BasicJsonType::default_object_comparator_t
- using type

### 9.3.1 Detailed Description

**template<typename BasicJsonType>**
**struct detail::actual_object_comparator< BasicJsonType >**

Definition at line 3720 of file json.hpp.

### 9.3.2 Member Typedef Documentation

#### 9.3.2.1 object_comparator_t

```
template<typename BasicJsonType>
using detail::actual_object_comparator< BasicJsonType >::object_comparator_t = typename Basic↩
JsonType::default_object_comparator_t
```

Definition at line 3723 of file json.hpp.

#### 9.3.2.2 object_t

```
template<typename BasicJsonType>
using detail::actual_object_comparator< BasicJsonType >::object_t = typename BasicJsonType::↩
object_t
```

Definition at line 3722 of file json.hpp.

#### 9.3.2.3 type

```
template<typename BasicJsonType>
using detail::actual_object_comparator< BasicJsonType >::type
```

**Initial value:**

```
typename std::conditional < has_key_compare<object_t>::value,
        typename object_t::key_compare, object_comparator_t>::type
```

Definition at line 3724 of file json.hpp.

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

- include/External/json.hpp

## 9.4 adl_serializer< ValueType, typename > Struct Template Reference

namespace for Niels Lohmann

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

**Static Public Member Functions**

- template<typename BasicJsonType, typename TargetType = ValueType>
  static auto from_json (BasicJsonType &&j, TargetType &val) noexcept(noexcept(::nlohmann::from_↩ json(std::forward< BasicJsonType >(j), val))) -> decltype(::nlohmann::from_json(std::forward< Basic↩ JsonType >(j), val), void())

  *convert a JSON value to any value type*
- template<typename BasicJsonType, typename TargetType = ValueType>
  static auto from_json (BasicJsonType &&j) noexcept(noexcept(::nlohmann::from_json(std::forward< Basic↩ JsonType >(j), detail::identity_tag< TargetType > {}))) -> decltype(::nlohmann::from_json(std::forward< BasicJsonType >(j), detail::identity_tag< TargetType > {}))

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

  *convert any value type to a JSON value*

## 9.4.1 Detailed Description

**template<typename ValueType, typename>**
**struct adl_serializer< ValueType, typename >**

namespace for Niels Lohmann

**See also**

https://github.com/nlohmann

**Since**

version 1.0.0

default JSONSerializer template argument

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

**See also**

https://json.nlohmann.me/api/adl_serializer/

Definition at line 6200 of file json.hpp.

## 9.4.2 Member Function Documentation

### 9.4.2.1 from_json() [1/2]

```
template<typename ValueType, typename>
template<typename BasicJsonType, typename TargetType = ValueType>
auto adl_serializer< ValueType, typename >::from_json (
            BasicJsonType && j) -> decltype(::nlohmann::from_json(std::forward< BasicJsonType
>(j), detail::identity_tag< TargetType > {})) [inline], [static], [noexcept]
```

convert a JSON value to any value type

**See also**

https://json.nlohmann.me/api/adl_serializer/from_json/

Definition at line 6215 of file json.hpp.

#### 9.4.2.2 from_json() [2/2]

```
template<typename ValueType, typename>
template<typename BasicJsonType, typename TargetType = ValueType>
auto adl_serializer< ValueType, typename >::from_json (
            BasicJsonType && j,
            TargetType & val) -> decltype(::nlohmann::from_json(std::forward< BasicJsonType
>(j), val), void())  [inline], [static], [noexcept]
```

convert a JSON value to any value type

**See also**

https://json.nlohmann.me/api/adl_serializer/from_json/

Definition at line 6205 of file json.hpp.

#### 9.4.2.3 to_json()

```
template<typename ValueType, typename>
template<typename BasicJsonType, typename TargetType = ValueType>
auto adl_serializer< ValueType, typename >::to_json (
            BasicJsonType & j,
            TargetType && val) -> decltype(::nlohmann::to_json(j, std::forward< TargetType
>(val)), void())  [inline], [static], [noexcept]
```

convert any value type to a JSON value

**See also**

https://json.nlohmann.me/api/adl_serializer/to_json/

Definition at line 6225 of file json.hpp.

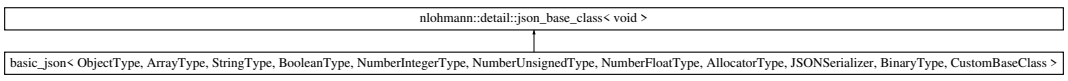
The documentation for this struct was generated from the following file:

- include/External/json.hpp

## 9.5 basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass > Class Template Reference

namespace for Niels Lohmann

Inheritance diagram for basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass >:



### Public Types

- using value_t = detail::value_t
- using json_pointer = ::nlohmann::json_pointer<StringType>
- template<typename T, typename SFINAE>
  using json_serializer = JSONSerializer<T, SFINAE>
- using error_handler_t = detail::error_handler_t
- using cbor_tag_handler_t = detail::cbor_tag_handler_t
- using bjdata_version_t = detail::bjdata_version_t
- using initializer_list_t = std::initializer_list<detail::json_ref<basic_json>>
- using input_format_t = detail::input_format_t
- using json_sax_t = json_sax<basic_json>
- using parse_error
- using invalid_iterator = detail::invalid_iterator
- using type_error = detail::type_error
- using out_of_range = detail::out_of_range
- using other_error = detail::other_error
- using reference
- using const_reference = const value_type&
- using difference_type = std::ptrdiff_t
- using size_type = std::size_t
- using allocator_type = AllocatorType<basic_json>
- using pointer = typename std::allocator_traits<allocator_type>::pointer
- using const_pointer = typename std::allocator_traits<allocator_type>::const_pointer
- using iterator = iter_impl<basic_json>
- using const_iterator = iter_impl<const basic_json>
- using reverse_iterator = json_reverse_iterator<typename basic_json::iterator>
- using const_reverse_iterator = json_reverse_iterator<typename basic_json::const_iterator>

### Public Attributes

- ∗JSON Pointer
- ∗SAX interface type
- ∗∗brief returns the allocator associated with the container ∗sa https: static allocator_type get_allocator() { return allocator_type()
- ∗brief returns version information on the library ∗sa https: JSON_HEDLEY_WARN_UNUSED_RESULT static basic_json meta() { basic_json result
- result ["copyright"] = "(C) 2013-2026 Niels Lohmann"
- return result
- ∗∗∗name JSON value data types ∗The data types to store a JSON value These types are derived from ∗the template arguments passed to class ref basic_json ∗∗brief a type for an object ∗sa https: using object_t = ObjectType<StringType
- ∗∗∗name JSON value data types ∗The data types to store a JSON value These types are derived from ∗the template arguments passed to class ref basic_json ∗∗brief a type for an object ∗sa default_object_comparator_t
- ∗∗∗name JSON value data types ∗The data types to store a JSON value These types are derived from ∗the template arguments passed to class ref basic_json ∗∗brief a type for an object ∗sa AllocatorType< std::pair< const StringType, basic_json > >
- ∗brief a type for an array ∗sa https: using array_t = ArrayType<basic_json
- ∗brief a type for an array ∗sa AllocatorType< basic_json >
- ∗brief a type for a string ∗sa https: using string_t = StringType
- ∗brief a type for a boolean ∗sa https: using boolean_t = BooleanType
- ∗brief a type for a number ∗(integer) ∗@sa https brief a type for a number ∗(unsigned) ∗@sa https brief a type for a number ∗(floating-point) ∗@sa https brief a type for a packed binary type ∗sa https: using binary_t = nlohmann::byte_container_with_subtype<BinaryType>

- ∗brief object key comparator type ∗sa https: using object_comparator_t = detail::actual_object_comparator_t<basic_json>
- ∗∗∗brief parser event types ∗sa https: using parse_event_t = detail::parse_event_t
- ∗brief per element parser callback type ∗sa https: using parser_callback_t = detail::parser_callback_t<basic_json>
- ∗∗∗name constructors and destructors ∗Constructors of class ref basic_json copy move constructor copy ∗ assignment
- ∗∗∗name constructors and destructors ∗Constructors of class ref basic_json copy move constructor copy static functions creating and the destructor ∗data m_data = {}

### Static Public Attributes

- ∗∗∗name constructors and destructors ∗Constructors of class ref basic_json copy move constructor copy static functions creating objects

### Friends

- template<detail::value_t>
  struct detail::external_constructor
- template<typename>
  class ::nlohmann::json_pointer
- template<typename BasicJsonType, typename InputType>
  class ::nlohmann::detail::parser
- template<typename BasicJsonType>
  class ::nlohmann::detail::iter_impl
- template<typename BasicJsonType, typename CharType>
  class ::nlohmann::detail::binary_writer
- template<typename BasicJsonType, typename InputType, typename SAX>
  class ::nlohmann::detail::binary_reader
- template<typename BasicJsonType, typename InputAdapterType>
  class ::nlohmann::detail::json_sax_dom_parser
- template<typename BasicJsonType, typename InputAdapterType>
  class ::nlohmann::detail::json_sax_dom_callback_parser
- class ::nlohmann::detail::exception

## 9.5.1 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 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, class BinaryType = std::vector<std::uint8_t>, class CustomBaseClass = void>**
**class basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass >**

namespace for Niels Lohmann

a class to store JSON values

**See also**

https://json.nlohmann.me/api/basic_json/
https://github.com/nlohmann

**Since**

version 1.0.0

a class to store JSON values

**Since**

version 1.0.0

Definition at line 20227 of file json.hpp.

### 9.5.2 Member Typedef Documentation

#### 9.5.2.1 allocator_type

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
using basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::allocator_type = AllocatorType<basic_json>
```

Definition at line 20353 of file json.hpp.

#### 9.5.2.2 bjdata_version_t

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
using basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::bjdata_version_t = detail::bjdata_version_t
```

Definition at line 20305 of file json.hpp.

#### 9.5.2.3 cbor_tag_handler_t

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
using basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::cbor_tag_handler_t = detail::cbor_tag_handler_t
```

Definition at line 20303 of file json.hpp.

#### 9.5.2.4 const_iterator

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
using basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::const_iterator = iter_impl<const basic_json>
```

Definition at line 20363 of file json.hpp.

#### 9.5.2.5 const_pointer

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
using basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::const_pointer = typename std::allocator_traits<allocator_type>::const_pointer
```

Definition at line 20358 of file json.hpp.

#### 9.5.2.6 const_reference

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
using basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::const_reference = const value_type&
```

Definition at line 20345 of file json.hpp.

#### 9.5.2.7 const_reverse_iterator

```
template<template< typename U, typename V, typename...  Args > class ObjectType = std::↩
map, template< typename U, typename...  Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
using basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::const_reverse_iterator = json_reverse_iterator<typename basic_json::const_iterator>
```

Definition at line 20367 of file json.hpp.

#### 9.5.2.8 difference_type

```
template<template< typename U, typename V, typename...  Args > class ObjectType = std::↩
map, template< typename U, typename...  Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
using basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::difference_type = std::ptrdiff_t
```

Definition at line 20348 of file json.hpp.

#### 9.5.2.9 error_handler_t

```
template<template< typename U, typename V, typename...  Args > class ObjectType = std::↩
map, template< typename U, typename...  Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
using basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::error_handler_t = detail::error_handler_t
```

Definition at line 20301 of file json.hpp.

#### 9.5.2.10 initializer_list_t

```
template<template< typename U, typename V, typename...  Args > class ObjectType = std::↩
map, template< typename U, typename...  Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
using basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::initializer_list_t = std::initializer_list<detail::json_ref<basic_json>>
```

Definition at line 20307 of file json.hpp.

#### 9.5.2.11 input_format_t

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
using basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::input_format_t = detail::input_format_t
```

Definition at line 20309 of file json.hpp.

#### 9.5.2.12 invalid_iterator

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
using basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::invalid_iterator = detail::invalid_iterator
```

Definition at line 20323 of file json.hpp.

#### 9.5.2.13 iterator

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
using basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::iterator = iter_impl<basic_json>
```

Definition at line 20361 of file json.hpp.

#### 9.5.2.14 json_pointer

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
using basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::json_pointer = ::nlohmann::json_pointer<StringType>
```

Definition at line 20297 of file json.hpp.

#### 9.5.2.15 json_sax_t

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
using basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::json_sax_t = json_sax<basic_json>
```

Definition at line 20311 of file json.hpp.

#### 9.5.2.16 json_serializer

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
template<typename T, typename SFINAE>
using basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::json_serializer = JSONSerializer<T, SFINAE>
```

Definition at line 20299 of file json.hpp.

#### 9.5.2.17 other_error

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
using basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::other_error = detail::other_error
```

Definition at line 20326 of file json.hpp.

#### 9.5.2.18 out_of_range

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
using basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::out_of_range = detail::out_of_range
```

Definition at line 20325 of file json.hpp.

#### 9.5.2.19 parse_error

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
using basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::parse_error
```

**Initial value:**

```
{
    using exception = detail::exception detail::parse_error
```

Definition at line 20322 of file json.hpp.

#### 9.5.2.20 pointer

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
using basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::pointer = typename std::allocator_traits<allocator_type>::pointer
```

Definition at line 20356 of file json.hpp.

#### 9.5.2.21 reference

```
template<template< typename U, typename V, typename...  Args > class ObjectType = std::↩
map, template< typename U, typename...  Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
using basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::reference
```

**Initial value:**

```
{

     *  the type of elements in a basic_json container
    using value_type = basic_json value_type&
```

Definition at line 20343 of file json.hpp.

#### 9.5.2.22 reverse_iterator

```
template<template< typename U, typename V, typename...  Args > class ObjectType = std::↩
map, template< typename U, typename...  Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
using basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::reverse_iterator = json_reverse_iterator<typename basic_json::iterator>
```

Definition at line 20365 of file json.hpp.

#### 9.5.2.23 size_type

```
template<template< typename U, typename V, typename...  Args > class ObjectType = std::↩
map, template< typename U, typename...  Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
using basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::size_type = std::size_t
```

Definition at line 20350 of file json.hpp.

#### 9.5.2.24 type_error

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
using basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::type_error = detail::type_error
```

Definition at line 20324 of file json.hpp.

#### 9.5.2.25 value_t

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
using basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::value_t = detail::value_t
```

Definition at line 20295 of file json.hpp.

### 9.5.3 Friends And Related Symbol Documentation

#### 9.5.3.1 ::nlohmann::detail::binary_reader

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
template<typename BasicJsonType, typename InputType, typename SAX>
friend class ::nlohmann::detail::binary_reader [friend]
```

Definition at line 20246 of file json.hpp.

#### 9.5.3.2 ::nlohmann::detail::binary_writer

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
template<typename BasicJsonType, typename CharType>
friend class ::nlohmann::detail::binary_writer [friend]
```

Definition at line 20244 of file json.hpp.

#### 9.5.3.3 ::nlohmann::detail::exception

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
friend class ::nlohmann::detail::exception [friend]
```

Definition at line 20251 of file json.hpp.

#### 9.5.3.4 ::nlohmann::detail::iter_impl

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
template<typename BasicJsonType>
friend class ::nlohmann::detail::iter_impl [friend]
```

Definition at line 20242 of file json.hpp.

#### 9.5.3.5 ::nlohmann::detail::json_sax_dom_callback_parser

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
template<typename BasicJsonType, typename InputAdapterType>
friend class ::nlohmann::detail::json_sax_dom_callback_parser [friend]
```

Definition at line 20250 of file json.hpp.

#### 9.5.3.6 ::nlohmann::detail::json_sax_dom_parser

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
template<typename BasicJsonType, typename InputAdapterType>
friend class ::nlohmann::detail::json_sax_dom_parser [friend]
```

Definition at line 20248 of file json.hpp.

#### 9.5.3.7 ::nlohmann::detail::parser

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
template<typename BasicJsonType, typename InputType>
friend class ::nlohmann::detail::parser [friend]
```

Definition at line 20239 of file json.hpp.

#### 9.5.3.8 ::nlohmann::json_pointer

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
template<typename>
friend class ::nlohmann::json_pointer [friend]
```

Definition at line 20234 of file json.hpp.

#### 9.5.3.9 detail::external_constructor

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
template<detail::value_t>
friend struct detail::external_constructor [friend]
```

Definition at line 20231 of file json.hpp.

### 9.5.4 Member Data Documentation

#### 9.5.4.1 AllocatorType< basic_json >

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
* brief a type for an array* sa basic_json< ObjectType, ArrayType, StringType, BooleanType,
NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer, Binary↩
Type, CustomBaseClass >::AllocatorType< basic_json >
```

Definition at line 20473 of file json.hpp.

#### 9.5.4.2 AllocatorType< std::pair< const StringType, basic_json > >

```
template<template< typename U, typename V, typename...  Args > class ObjectType = std::↩
map, template< typename U, typename...  Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
* * * name JSON value data types* The data types to store a JSON value These types are derived
from* the template arguments passed to class ref basic_json* * brief a type for an object*
sa basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::AllocatorType< std::pair< const StringType, basic_json > >
```

Definition at line 20468 of file json.hpp.

#### 9.5.4.3 assignment

```
template<template< typename U, typename V, typename...  Args > class ObjectType = std::↩
map, template< typename U, typename...  Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
* * * name constructors and destructors* Constructors of class ref basic_json copy move constructor
copy* basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::assignment
```

Definition at line 20943 of file json.hpp.

#### 9.5.4.4 basic_json

```
template<template< typename U, typename V, typename...  Args > class ObjectType = std::↩
map, template< typename U, typename...  Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
* * * name JSON value data types* The data types to store a JSON value These types are derived
from* the template arguments passed to class ref basic_json* * brief a type for an object*
sa basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::basic_json
```

Definition at line 20467 of file json.hpp.

#### 9.5.4.5 default_object_comparator_t

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
* * * name JSON value data types* The data types to store a JSON value These types are derived
from* the template arguments passed to class ref basic_json* * brief a type for an object*
sa basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::default_object_comparator_t
```

Definition at line 20468 of file json.hpp.

#### 9.5.4.6 https [1/10]

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
* brief per element parser callback type* sa basic_json< ObjectType, ArrayType, StringType,
BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer,
BinaryType, CustomBaseClass >::https
```

Definition at line 20934 of file json.hpp.

#### 9.5.4.7 https [2/10]

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
* * * brief parser event types* sa basic_json< ObjectType, ArrayType, StringType, Boolean↩
Type, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer,
BinaryType, CustomBaseClass >::https
```

Definition at line 20930 of file json.hpp.

#### 9.5.4.8 https [3/10]

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
* brief object key comparator type* sa basic_json< ObjectType, ArrayType, StringType, Boolean↩
Type, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer,
BinaryType, CustomBaseClass >::https
```

Definition at line 20501 of file json.hpp.

#### 9.5.4.9 https [4/10]

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
* brief a type for a number* (integer) * @sa https brief a type for a number* (unsigned) *
@sa https brief a type for a number* (floating-point) * @sa https brief a type for a packed
binary type* sa basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType,
NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBase↩
Class >::https
```

Definition at line 20497 of file json.hpp.

#### 9.5.4.10 https [5/10]

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
* brief a type for a boolean* sa basic_json< ObjectType, ArrayType, StringType, Boolean↩
Type, NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer,
BinaryType, CustomBaseClass >::https
```

Definition at line 20481 of file json.hpp.

#### 9.5.4.11 https [6/10]

```
template<template< typename U, typename V, typename...  Args > class ObjectType = std::↩
map, template< typename U, typename...  Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
* brief a type for a string* sa basic_json< ObjectType, ArrayType, StringType, BooleanType,
NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer, Binary↩
Type, CustomBaseClass >::https
```

Definition at line 20477 of file json.hpp.

#### 9.5.4.12 https [7/10]

```
template<template< typename U, typename V, typename...  Args > class ObjectType = std::↩
map, template< typename U, typename...  Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
* brief a type for an array* sa basic_json< ObjectType, ArrayType, StringType, BooleanType,
NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer, Binary↩
Type, CustomBaseClass >::https
```

Definition at line 20473 of file json.hpp.

#### 9.5.4.13 https [8/10]

```
template<template< typename U, typename V, typename...  Args > class ObjectType = std::↩
map, template< typename U, typename...  Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
* * * name JSON value data types* The data types to store a JSON value These types are derived
from* the template arguments passed to class ref basic_json* * brief a type for an object*
sa basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::https
```

**Initial value:**

```
{

    *  @brief default object key comparator type
    *  The actual object key comparator type (@ref object_comparator_t) may be
    *  different.
    *  @sa https:



    using default_object_comparator_t = std::less<StringType>
```

Definition at line 20465 of file json.hpp.

#### 9.5.4.14 https [9/10]

```
template<template< typename U, typename V, typename...  Args > class ObjectType = std::↩
map, template< typename U, typename...  Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
* brief returns version information on the library* sa basic_json< ObjectType, ArrayType,
StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, Allocator↩
Type, JSONSerializer, BinaryType, CustomBaseClass >::https
```

Definition at line 20379 of file json.hpp.

#### 9.5.4.15 https [10/10]

```
template<template< typename U, typename V, typename...  Args > class ObjectType = std::↩
map, template< typename U, typename...  Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
* * brief returns the allocator associated with the container* sa basic_json< ObjectType,
ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType,
AllocatorType, JSONSerializer, BinaryType, CustomBaseClass >::https
```

Definition at line 20372 of file json.hpp.

#### 9.5.4.16 m_data

```
template<template< typename U, typename V, typename...  Args > class ObjectType = std::↩
map, template< typename U, typename...  Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
* * * name constructors and destructors* Constructors of class ref basic_json copy move constructor
copy static functions creating and the destructor* data basic_json< ObjectType, ArrayType,
StringType, BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType, Allocator↩
Type, JSONSerializer, BinaryType, CustomBaseClass >::m_data = {}
```

Definition at line 24438 of file json.hpp.

#### 9.5.4.17 objects

```
template<template< typename U, typename V, typename...  Args > class ObjectType = std::↩
map, template< typename U, typename...  Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
* * * name constructors and destructors* Constructors of class ref basic_json copy move constructor
copy static functions creating basic_json< ObjectType, ArrayType, StringType, BooleanType,
NumberIntegerType, NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer, Binary↩
Type, CustomBaseClass >::objects  [static]
```

Definition at line 20943 of file json.hpp.

#### 9.5.4.18 Pointer

```
template<template< typename U, typename V, typename...  Args > class ObjectType = std::↩
map, template< typename U, typename...  Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
* JSON basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::Pointer
```

Definition at line 20296 of file json.hpp.

#### 9.5.4.19 result [1/2]

```
template<template< typename U, typename V, typename...  Args > class ObjectType = std::↩
map, template< typename U, typename...  Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
return basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, Number↩
UnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass
>::result
```

Definition at line 20440 of file json.hpp.

#### 9.5.4.20 result [2/2]

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsigned↩
Type, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBaseClass >::result↩
["compiler"]["c++"] = "(C) 2013-2026 Niels Lohmann"
```

Definition at line 20385 of file json.hpp.

#### 9.5.4.21 type

```
template<template< typename U, typename V, typename... Args > class ObjectType = std::↩
map, template< typename U, typename... Args > class ArrayType = std::vector, class String↩
Type = 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, class BinaryType = std::vector<std::uint8_t>, class Custom↩
BaseClass = void>
* SAX interface basic_json< ObjectType, ArrayType, StringType, BooleanType, NumberIntegerType,
NumberUnsignedType, NumberFloatType, AllocatorType, JSONSerializer, BinaryType, CustomBase↩
Class >::type
```

Definition at line 20310 of file json.hpp.

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

- include/External/json.hpp

## 9.6 BenchmarkRunner Class Reference

### Public Member Functions

- void runBenchmarks (const std::string &inputFile, const std::string &benchmarkName)

### 9.6.1 Detailed Description

Definition at line 14 of file BenchmarkRunner.h.

### 9.6.2 Member Function Documentation

#### 9.6.2.1 runBenchmarks()

```
void BenchmarkRunner::runBenchmarks (
            const std::string & inputFile,
            const std::string & benchmarkName)
```

Definition at line 125 of file BenchmarkRunner.cpp.

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

- include/BenchmarkRunner.h
- src/BenchmarkRunner.cpp

## 9.7 detail::binary_reader< BasicJsonType, InputAdapterType, SAX > Class Template Reference

deserialization of CBOR, MessagePack, and UBJSON values

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

#### Public Member Functions

- binary_reader (InputAdapterType &&adapter, const input_format_t format=input_format_t::json) noexcept
  *create a binary reader*
- **binary_reader** (const binary_reader &)=delete
- **binary_reader** (binary_reader &&)=default
- binary_reader & **operator=** (const binary_reader &)=delete
- binary_reader & **operator=** (binary_reader &&)=default
- bool sax_parse (const input_format_t format, json_sax_t *sax_, const bool strict=true, const cbor_tag_handler_t tag_handler=cbor_tag_handler_t::error)

### 9.7.1 Detailed Description

**template<typename BasicJsonType, typename InputAdapterType, typename SAX = json_sax_dom_↩ parser<BasicJsonType, InputAdapterType>>**
**class detail::binary_reader< BasicJsonType, InputAdapterType, SAX >**

deserialization of CBOR, MessagePack, and UBJSON values

Definition at line 9911 of file json.hpp.

## 9.7.2 Constructor & Destructor Documentation

### 9.7.2.1 binary_reader()

```
template<typename BasicJsonType, typename InputAdapterType, typename SAX = json_sax_dom_↩
parser<BasicJsonType, InputAdapterType>>
detail::binary_reader< BasicJsonType, InputAdapterType, SAX >::binary_reader (
            InputAdapterType && adapter,
            const input_format_t format = input_format_t::json)  [inline], [explicit], [noexcept]
```

create a binary reader

**Parameters**

| | | |
|---|---|---|
| in | *adapter* | input adapter to read from |

Definition at line 9928 of file json.hpp.

## 9.7.3 Member Function Documentation

### 9.7.3.1 sax_parse()

```
template<typename BasicJsonType, typename InputAdapterType, typename SAX = json_sax_dom_↩
parser<BasicJsonType, InputAdapterType>>
bool detail::binary_reader< BasicJsonType, InputAdapterType, SAX >::sax_parse (
            const input_format_t format,
            json_sax_t * sax_,
            const bool strict = true,
            const cbor_tag_handler_t tag_handler = cbor_tag_handler_t::error)  [inline]
```

**Parameters**

| | | |
|---|---|---|
| in | *format* | the binary format to parse |
| in | *sax_* | a SAX event processor |
| in | *strict* | whether to expect the input to be consumed completed |
| in | *tag_handler* | how to treat CBOR tags |

**Returns**

whether parsing was successful

Definition at line 9949 of file json.hpp.

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

- include/External/json.hpp

# 9.8 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_bson (const BasicJsonType &j)
- void write_cbor (const BasicJsonType &j)
- void write_msgpack (const BasicJsonType &j)
- void write_ubjson (const BasicJsonType &j, const bool use_count, const bool use_type, const bool add_↩ prefix=true, const bool use_bjdata=false, const bjdata_version_t bjdata_version=bjdata_version_t::draft2)

### Static Public Member Functions

- template<typename C = CharType, enable_if_t< std::is_signed< C >::value &&std::is_signed< char >::value > * = nullptr>
  static constexpr CharType to_char_type (std::uint8_t x) noexcept
- template<typename C = CharType, enable_if_t< std::is_signed< C >::value &&std::is_unsigned< char >::value > * = nullptr>
  static CharType to_char_type (std::uint8_t x) noexcept
- template<typename C = CharType, enable_if_t< std::is_unsigned< C >::value > * = nullptr>
  static constexpr CharType to_char_type (std::uint8_t x) noexcept
- template<typename InputCharType, typename C = CharType, enable_if_t< std::is_signed< C >::value &&std::is_signed< char >::↩ value &&std::is_same< char, typename std::remove_cv< InputCharType >::type >::value > * = nullptr>
  static constexpr CharType to_char_type (InputCharType x) noexcept

## 9.8.1 Detailed Description

**template<typename BasicJsonType, typename CharType>**
**class detail::binary_writer< BasicJsonType, CharType >**

serialization to CBOR and MessagePack values

Definition at line 15888 of file json.hpp.

## 9.8.2 Constructor & Destructor Documentation

### 9.8.2.1 binary_writer()

```
template<typename BasicJsonType, typename CharType>
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 |

Definition at line 15900 of file json.hpp.

### 9.8.3 Member Function Documentation

#### 9.8.3.1 to_char_type() [1/4]

```
template<typename BasicJsonType, typename CharType>
template<typename InputCharType, typename C = CharType, enable_if_t< std::is_signed< C >::↩
value &&std::is_signed< char >::value &&std::is_same< char, typename std::remove_cv< Input↩
CharType >::type >::value > * = nullptr>
constexpr CharType detail::binary_writer< BasicJsonType, CharType >::to_char_type (
            InputCharType x)  [inline], [static], [constexpr], [noexcept]
```

Definition at line 17691 of file json.hpp.

#### 9.8.3.2 to_char_type() [2/4]

```
template<typename BasicJsonType, typename CharType>
template<typename C = CharType, enable_if_t< std::is_unsigned< C >::value > * = nullptr>
constexpr CharType detail::binary_writer< BasicJsonType, CharType >::to_char_type (
            std::uint8_t x)  [inline], [static], [constexpr], [noexcept]
```

Definition at line 17680 of file json.hpp.

#### 9.8.3.3 to_char_type() [3/4]

```
template<typename BasicJsonType, typename CharType>
template<typename C = CharType, enable_if_t< std::is_signed< C >::value &&std::is_unsigned<
char >::value > * = nullptr>
CharType detail::binary_writer< BasicJsonType, CharType >::to_char_type (
            std::uint8_t x)  [inline], [static], [noexcept]
```

Definition at line 17656 of file json.hpp.

#### 9.8.3.4 to_char_type() [4/4]

```
template<typename BasicJsonType, typename CharType>
template<typename C = CharType, enable_if_t< std::is_signed< C >::value &&std::is_signed<
char >::value > * = nullptr>
constexpr CharType detail::binary_writer< BasicJsonType, CharType >::to_char_type (
            std::uint8_t x)  [inline], [static], [constexpr], [noexcept]
```

Definition at line 17649 of file json.hpp.

#### 9.8.3.5 write_bson()

```
template<typename BasicJsonType, typename CharType>
void detail::binary_writer< BasicJsonType, CharType >::write_bson (
            const BasicJsonType & j)  [inline]
```

**Parameters**

| | | |
|---|---|---|
| in | *j* | JSON value to serialize |

**Precondition**

j.type() == value_t::object

Definition at line 15909 of file json.hpp.

#### 9.8.3.6 write_cbor()

```
template<typename BasicJsonType, typename CharType>
void detail::binary_writer< BasicJsonType, CharType >::write_cbor (
            const BasicJsonType & j)  [inline]
```

**Parameters**

| | | |
|---|---|---|
| in | *j* | JSON value to serialize |

Definition at line 15938 of file json.hpp.

#### 9.8.3.7 write_msgpack()

```
template<typename BasicJsonType, typename CharType>
void detail::binary_writer< BasicJsonType, CharType >::write_msgpack (
            const BasicJsonType & j)  [inline]
```

**Parameters**

| | | |
|---|---|---|
| in | *j* | JSON value to serialize |

Definition at line 16262 of file json.hpp.

#### 9.8.3.8 write_ubjson()

```
template<typename BasicJsonType, typename CharType>
void detail::binary_writer< BasicJsonType, CharType >::write_ubjson (
            const BasicJsonType & j,
            const bool use_count,
            const bool use_type,
            const bool add_prefix = true,
            const bool use_bjdata = false,
            const bjdata_version_t bjdata_version = bjdata_version_t::draft2)  [inline]
```

**Parameters**

| | | |
|---|---|---|
| in | *j* | JSON value to serialize |
| in | *use_count* | whether to use '#' prefixes (optimized format) |
| in | *use_type* | whether to use '$' prefixes (optimized format) |
| in | *add_prefix* | whether prefixes need to be used for this value |
| in | *use_bjdata* | whether write in BJData format, default is false |
| in | *bjdata_version* | which BJData version to use, default is draft2 |

Definition at line 16589 of file json.hpp.

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

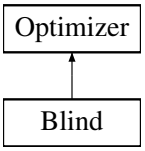
- include/External/json.hpp

## 9.9 Blind Class Reference

Implements a blind (random walk) optimization algorithm.

`#include <Blind.h>`

Inheritance diagram for Blind:




#### Public Member Functions

- Blind (SolutionBuilder &solutionBuilder, Problem &problem, int maxIterations)

  *Constructs a Blind optimizer.*
- double optimize () override

  *Executes the blind optimization process.*

### Public Member Functions inherited from Optimizer

- Optimizer (SolutionBuilder &solutionBuilder, Problem &problem, int maxIterations)

  *Constructs an Optimizer.*
- virtual ~**Optimizer** ()=default

  *Virtual destructor for safe polymorphic deletion.*

- double getBestFitness ()
- std::vector< double > & getBestSolution ()
- std::vector< double > & getBestFitnesses ()
- std::vector< std::vector< double > > & getSolutions ()
- int getMaxIterations ()
- Problem & getProblem ()
- SolutionBuilder & getSolutionBuilder ()

#### Additional Inherited Members

### Protected Attributes inherited from Optimizer

- Problem & problem

  *Optimization problem definition.*
- SolutionBuilder & solutionBuilder

  *Solution generation utility.*
- int maxIterations

  *Maximum number of iterations.*
- std::vector< double > bestSolution

  *Best solution found.*
- std::vector< double > bestFitnesses

  *Best fitness value so far per iteration.*
- std::vector< std::vector< double > > solutions

  *All generated solutions.*

### 9.9.1 Detailed Description

Implements a blind (random walk) optimization algorithm.

The Blind optimizer repeatedly samples random solutions from the solution space without using neighborhood information or gradient guidance. It keeps track of the best solution found across all iterations.

This algorithm serves as a baseline for comparison against more informed local search techniques.

Definition at line 27 of file Blind.h.

### 9.9.2 Constructor & Destructor Documentation

#### 9.9.2.1 Blind()

```
Blind::Blind (
            SolutionBuilder & solutionBuilder,
            Problem & problem,
            int maxIterations)  [inline]
```

Constructs a Blind optimizer.

**Parameters**

| | |
|---|---|
| *solutionBuilder* | Reference to the solution generator. |
| *problem* | Reference to the optimization problem. |
| *maxIterations* | Maximum number of iterations to perform. |

Definition at line 37 of file Blind.h.

### 9.9.3 Member Function Documentation

#### 9.9.3.1 optimize()

```
double Blind::optimize ()  [override], [virtual]
```

Executes the blind optimization process.

**Returns**

Execution time of the algorithm.

Implements Optimizer.

Definition at line 6 of file Blind.cpp.

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

- include/Optimizer/Blind.h
- src/Optimizer/Blind.cpp

## 9.10 detail::dtoa_impl::boundaries Struct Reference

**Public Attributes**

- diyfp w
- diyfp minus
- diyfp plus

### 9.10.1 Detailed Description

Definition at line 17910 of file json.hpp.

### 9.10.2 Member Data Documentation

#### 9.10.2.1 minus

```
diyfp detail::dtoa_impl::boundaries::minus
```

Definition at line 17913 of file json.hpp.

#### 9.10.2.2 plus

```
diyfp detail::dtoa_impl::boundaries::plus
```

Definition at line 17914 of file json.hpp.

#### 9.10.2.3 w

```
diyfp detail::dtoa_impl::boundaries::w
```

Definition at line 17912 of file json.hpp.

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

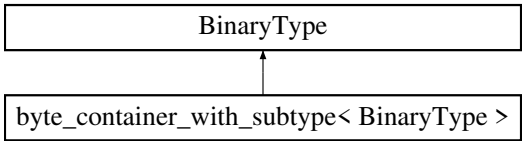
- include/External/json.hpp

## 9.11 byte_container_with_subtype< BinaryType > Class Template Reference

an internal type for a backed binary type

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

Inheritance diagram for byte_container_with_subtype< BinaryType >:



**Public Types**

- using container_type = BinaryType
- using subtype_type = std::uint64_t

**Public Member Functions**

- byte_container_with_subtype () noexcept(noexcept(container_type()))
- byte_container_with_subtype (const container_type &b) noexcept(noexcept(container_type(b)))
- byte_container_with_subtype (container_type &&b) noexcept(noexcept(container_type(std::move(b))))
- byte_container_with_subtype (const container_type &b, subtype_type subtype_) noexcept(noexcept(container↩ _type(b)))
- byte_container_with_subtype (container_type &&b, subtype_type subtype_) noexcept(noexcept(container↩ _type(std::move(b))))
- bool operator== (const byte_container_with_subtype &rhs) const
- bool operator!= (const byte_container_with_subtype &rhs) const
- void set_subtype (subtype_type subtype_) noexcept

  *sets the binary subtype*
- constexpr subtype_type subtype () const noexcept

  *return the binary subtype*
- constexpr bool has_subtype () const noexcept

  *return whether the value has a subtype*
- void clear_subtype () noexcept

  *clears the binary subtype*

## 9.11.1 Detailed Description

**template<typename BinaryType>**
**class byte_container_with_subtype< BinaryType >**

an internal type for a backed binary type

**See also**

https://json.nlohmann.me/api/byte_container_with_subtype/

Definition at line 6258 of file json.hpp.

## 9.11.2 Member Typedef Documentation

### 9.11.2.1 container_type

```
template<typename BinaryType>
using byte_container_with_subtype< BinaryType >::container_type = BinaryType
```

Definition at line 6261 of file json.hpp.

#### 9.11.2.2 subtype_type

```
template<typename BinaryType>
using byte_container_with_subtype< BinaryType >::subtype_type = std::uint64_t
```

Definition at line 6262 of file json.hpp.

### 9.11.3 Constructor & Destructor Documentation

#### 9.11.3.1 byte_container_with_subtype() [1/5]

```
template<typename BinaryType>
byte_container_with_subtype< BinaryType >::byte_container_with_subtype ()  [inline], [noexcept]
```

**See also**

https://json.nlohmann.me/api/byte_container_with_subtype/byte_container_with_subtype/

Definition at line 6265 of file json.hpp.

#### 9.11.3.2 byte_container_with_subtype() [2/5]

```
template<typename BinaryType>
byte_container_with_subtype< BinaryType >::byte_container_with_subtype (
            const container_type & b)  [inline], [noexcept]
```

**See also**

https://json.nlohmann.me/api/byte_container_with_subtype/byte_container_with_subtype/

Definition at line 6270 of file json.hpp.

#### 9.11.3.3 byte_container_with_subtype() [3/5]

```
template<typename BinaryType>
byte_container_with_subtype< BinaryType >::byte_container_with_subtype (
            container_type && b)  [inline], [noexcept]
```

**See also**

https://json.nlohmann.me/api/byte_container_with_subtype/byte_container_with_subtype/

Definition at line 6275 of file json.hpp.

#### 9.11.3.4 byte_container_with_subtype() [4/5]

```
template<typename BinaryType>
byte_container_with_subtype< BinaryType >::byte_container_with_subtype (
            const container_type & b,
            subtype_type subtype_)  [inline], [noexcept]
```

**See also**

https://json.nlohmann.me/api/byte_container_with_subtype/byte_container_with_subtype/

Definition at line 6280 of file json.hpp.

#### 9.11.3.5 byte_container_with_subtype() [5/5]

```
template<typename BinaryType>
byte_container_with_subtype< BinaryType >::byte_container_with_subtype (
            container_type && b,
            subtype_type subtype_)  [inline], [noexcept]
```

**See also**

https://json.nlohmann.me/api/byte_container_with_subtype/byte_container_with_subtype/

Definition at line 6287 of file json.hpp.

### 9.11.4 Member Function Documentation

#### 9.11.4.1 clear_subtype()

```
template<typename BinaryType>
void byte_container_with_subtype< BinaryType >::clear_subtype ()  [inline], [noexcept]
```

clears the binary subtype

**See also**

https://json.nlohmann.me/api/byte_container_with_subtype/clear_subtype/

Definition at line 6328 of file json.hpp.

#### 9.11.4.2 has_subtype()

```
template<typename BinaryType>
bool byte_container_with_subtype< BinaryType >::has_subtype () const  [inline], [constexpr],
[noexcept]
```

return whether the value has a subtype

**See also**

https://json.nlohmann.me/api/byte_container_with_subtype/has_subtype/

Definition at line 6321 of file json.hpp.

#### 9.11.4.3 operator"!=()

```
template<typename BinaryType>
bool byte_container_with_subtype< BinaryType >::operator!= (
            const byte_container_with_subtype< BinaryType > & rhs) const  [inline]
```

Definition at line 6299 of file json.hpp.

#### 9.11.4.4 operator==()

```
template<typename BinaryType>
bool byte_container_with_subtype< BinaryType >::operator== (
            const byte_container_with_subtype< BinaryType > & rhs) const  [inline]
```

Definition at line 6293 of file json.hpp.

#### 9.11.4.5 set_subtype()

```
template<typename BinaryType>
void byte_container_with_subtype< BinaryType >::set_subtype (
            subtype_type subtype_)  [inline], [noexcept]
```

sets the binary subtype

**See also**

https://json.nlohmann.me/api/byte_container_with_subtype/set_subtype/

Definition at line 6306 of file json.hpp.

#### 9.11.4.6 subtype()

```
template<typename BinaryType>
subtype_type byte_container_with_subtype< BinaryType >::subtype () const  [inline], [constexpr],
[noexcept]
```

return the binary subtype

**See also**

https://json.nlohmann.me/api/byte_container_with_subtype/subtype/

Definition at line 6314 of file json.hpp.

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

- include/External/json.hpp

## 9.12 detail::dtoa_impl::cached_power Struct Reference

**Public Attributes**

- std::uint64_t f
- int e
- int k

### 9.12.1 Detailed Description

Definition at line 18049 of file json.hpp.

### 9.12.2 Member Data Documentation

#### 9.12.2.1 e

```
int detail::dtoa_impl::cached_power::e
```

Definition at line 18052 of file json.hpp.

#### 9.12.2.2 f

```
std::uint64_t detail::dtoa_impl::cached_power::f
```

Definition at line 18051 of file json.hpp.

#### 9.12.2.3 k

```
int detail::dtoa_impl::cached_power::k
```

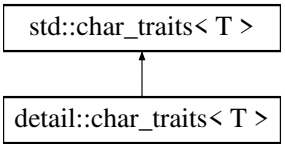
Definition at line 18053 of file json.hpp.

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

- include/External/json.hpp

## 9.13 detail::char_traits< T > Struct Template Reference

Inheritance diagram for detail::char_traits< T >:




### 9.13.1 Detailed Description

**template<typename T>**
**struct detail::char_traits< T >**

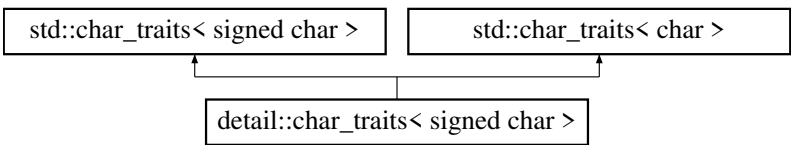
Definition at line 3737 of file json.hpp.

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

- include/External/json.hpp

## 9.14 detail::char_traits< signed char > Struct Reference

Inheritance diagram for detail::char_traits< signed char >:




#### Public Types

- using char_type = signed char
- using int_type = uint64_t

**Static Public Member Functions**

- static int_type to_int_type (char_type c) noexcept
- static char_type to_char_type (int_type i) noexcept
- static constexpr int_type eof () noexcept

## 9.14.1 Detailed Description

Definition at line 3766 of file json.hpp.

## 9.14.2 Member Typedef Documentation

### 9.14.2.1 char_type

```
using detail::char_traits< signed char >::char_type = signed char
```

Definition at line 3768 of file json.hpp.

### 9.14.2.2 int_type

```
using detail::char_traits< signed char >::int_type = uint64_t
```

Definition at line 3769 of file json.hpp.

## 9.14.3 Member Function Documentation

### 9.14.3.1 eof()

```
constexpr int_type detail::char_traits< signed char >::eof ()  [inline], [static], [constexpr],
[noexcept]
```

Definition at line 3782 of file json.hpp.

### 9.14.3.2 to_char_type()

```
char_type detail::char_traits< signed char >::to_char_type (
            int_type i)  [inline], [static], [noexcept]
```

Definition at line 3777 of file json.hpp.

### 9.14.3.3 to_int_type()

```
int_type detail::char_traits< signed char >::to_int_type (
            char_type c)  [inline], [static], [noexcept]
```

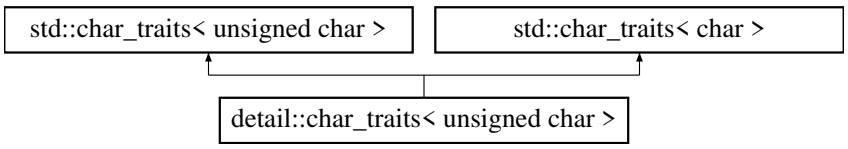
Definition at line 3772 of file json.hpp.

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

- include/External/json.hpp

# 9.15 detail::char_traits< unsigned char > Struct Reference

Inheritance diagram for detail::char_traits< unsigned char >:




### Public Types

- using char_type = unsigned char
- using int_type = uint64_t

### Static Public Member Functions

- static int_type to_int_type (char_type c) noexcept
- static char_type to_char_type (int_type i) noexcept
- static constexpr int_type eof () noexcept

## 9.15.1 Detailed Description

Definition at line 3742 of file json.hpp.

## 9.15.2 Member Typedef Documentation

### 9.15.2.1 char_type

```
using detail::char_traits< unsigned char >::char_type = unsigned char
```

Definition at line 3744 of file json.hpp.

### 9.15.2.2 int_type

```
using detail::char_traits< unsigned char >::int_type = uint64_t
```

Definition at line 3745 of file json.hpp.

## 9.15.3 Member Function Documentation

### 9.15.3.1 eof()

```
constexpr int_type detail::char_traits< unsigned char >::eof ()  [inline], [static], [constexpr],
[noexcept]
```

Definition at line 3758 of file json.hpp.

#### 9.15.3.2 to_char_type()

```
char_type detail::char_traits< unsigned char >::to_char_type (
            int_type i)  [inline], [static], [noexcept]
```

Definition at line 3753 of file json.hpp.

#### 9.15.3.3 to_int_type()

```
int_type detail::char_traits< unsigned char >::to_int_type (
            char_type c)  [inline], [static], [noexcept]
```

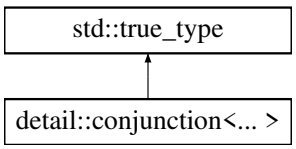
Definition at line 3748 of file json.hpp.

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

- include/External/json.hpp

## 9.16 detail::conjunction<... > Struct Template Reference

Inheritance diagram for detail::conjunction<... >:




### 9.16.1 Detailed Description

**template<class...>**
**struct detail::conjunction<... >**

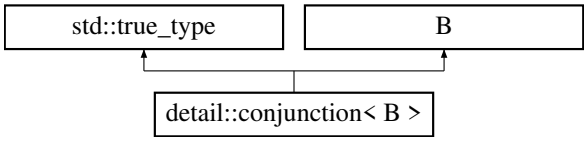
Definition at line 3817 of file json.hpp.

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

- include/External/json.hpp

## 9.17 detail::conjunction< B > Struct Template Reference

Inheritance diagram for detail::conjunction< B >:



### 9.17.1 Detailed Description

**template<class B>**
**struct detail::conjunction< B >**

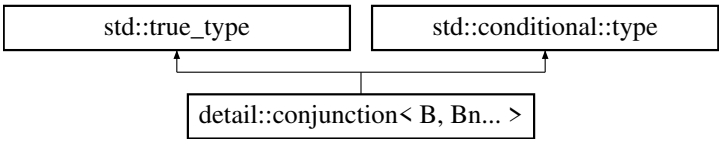
Definition at line 3818 of file json.hpp.

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

- include/External/json.hpp

## 9.18 detail::conjunction< B, Bn... > Struct Template Reference

Inheritance diagram for detail::conjunction< B, Bn... >:




### 9.18.1 Detailed Description

**template<class B, class... Bn>**
**struct detail::conjunction< B, Bn... >**

Definition at line 3820 of file json.hpp.

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

- include/External/json.hpp

## 9.19 detail::container_input_adapter_factory_impl::container_input_adapter_factory< ContainerType, Enable > Struct Template Reference

### 9.19.1 Detailed Description

**template<typename ContainerType, typename Enable = void>**
**struct detail::container_input_adapter_factory_impl::container_input_adapter_factory< ContainerType, Enable >**

Definition at line 6957 of file json.hpp.

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

- include/External/json.hpp

# 9.20 detail::container_input_adapter_factory_impl::container_input_adapter_factory< ContainerType, void_t< decltype(begin(std::declval< ContainerType >()), end(std::declval< ContainerType >()))> > Struct Template Reference

**Public Types**

- using adapter_type = decltype(input_adapter(begin(std::declval<ContainerType>()), end(std::declval<ContainerType>())))

**Static Public Member Functions**

- static adapter_type create (const ContainerType &container)

## 9.20.1 Detailed Description

**template<typename ContainerType>**
**struct detail::container_input_adapter_factory_impl::container_input_adapter_factory< ContainerType, void_t< decltype(begin(std::declval< ContainerType >()), end(std::declval< ContainerType >()))> >**

Definition at line 6960 of file json.hpp.

## 9.20.2 Member Typedef Documentation

### 9.20.2.1 adapter_type

```
template<typename ContainerType>
using detail::container_input_adapter_factory_impl::container_input_adapter_factory< ContainerType, void_t< decltype(begin(std::declval< ContainerType >()), end(std::declval< ContainerType >()))> >::adapter_type = decltype(input_adapter(begin(std::declval<ContainerType>()), end(std::declval<ContainerType>())))
```

Definition at line 6963 of file json.hpp.

## 9.20.3 Member Function Documentation

### 9.20.3.1 create()

```
template<typename ContainerType>
adapter_type detail::container_input_adapter_factory_impl::container_input_adapter_factory<
ContainerType, void_t< decltype(begin(std::declval< ContainerType >()), end(std::declval<
ContainerType >()))> >::create (
            const ContainerType & container)  [inline], [static]
```

Definition at line 6965 of file json.hpp.

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

- include/External/json.hpp

## 9.21 DeJong Class Reference

Implements the DeJong 1 benchmark function.

`#include <DeJongOne.h>`

### 9.21.1 Detailed Description

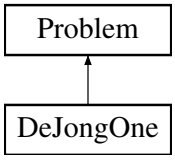
Implements the DeJong 1 benchmark function.

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

- include/Problem/DeJongOne.h

## 9.22 DeJongOne Class Reference

Inheritance diagram for DeJongOne:




#### Public Member Functions

- double evaluate (const std::vector< double > &x) const override

  *Evaluates the fitness of a candidate solution.*

### Public Member Functions inherited from Problem

- Problem (double lb, double ub, const std::string_view n)

  *Constructs a Problem instance.*
- virtual ~**Problem** ()=default

  *Virtual destructor for safe polymorphic cleanup.*

- double getLowerBound () const
- double getUpperBound () const
- const std::string getName () const

#### Additional Inherited Members

### Protected Attributes inherited from Problem

- const double lowerBound

  *Lower bound of the search space.*
- const double upperBound

  *Upper bound of the search space.*
- const std::string name

  *Name of the benchmark function.*

### 9.22.1 Detailed Description

Definition at line 19 of file DeJongOne.h.

### 9.22.2 Constructor & Destructor Documentation

#### 9.22.2.1 DeJongOne()

```
DeJongOne::DeJongOne () [inline]
```

Definition at line 27 of file DeJongOne.h.

### 9.22.3 Member Function Documentation

#### 9.22.3.1 evaluate()

```
double DeJongOne::evaluate (
            const std::vector< double > & x) const [inline], [override], [virtual]
```

Evaluates the fitness of a candidate solution.

- This is a pure virtual function that must be implemented by specific benchmark functions (e.g., Ackley, Rosenbrock).
- **Parameters**

| | |
|---|---|
| *x* | The solution vector to evaluate. |

**Returns**

The scalar fitness value (cost).

Implements Problem.

Definition at line 30 of file DeJongOne.h.

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

- include/Problem/DeJongOne.h

## 9.23 detail::detector< Default, AlwaysVoid, Op, Args > Struct Template Reference

**Public Types**

- using value_t = std::false_type
- using type = Default

### 9.23.1 Detailed Description

**template<class Default, class AlwaysVoid, template< class... > class Op, class... Args>**
**struct detail::detector< Default, AlwaysVoid, Op, Args >**

Definition at line 291 of file json.hpp.

### 9.23.2 Member Typedef Documentation

#### 9.23.2.1 type

```
template<class Default, class AlwaysVoid, template< class...  > class Op, class...  Args>
using detail::detector< Default, AlwaysVoid, Op, Args >::type = Default
```

Definition at line 294 of file json.hpp.

#### 9.23.2.2 value_t

```
template<class Default, class AlwaysVoid, template< class...  > class Op, class...  Args>
using detail::detector< Default, AlwaysVoid, Op, Args >::value_t = std::false_type
```

Definition at line 293 of file json.hpp.

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

- include/External/json.hpp

## 9.24 detail::detector< Default, void_t< Op< Args... > >, Op, Args... > Struct Template Reference

**Public Types**

- using value_t = std::true_type
- using type = Op<Args...>

### 9.24.1 Detailed Description

**template<class Default, template< class... > class Op, class... Args>**
**struct detail::detector< Default, void_t< Op< Args... > >, Op, Args... >**

Definition at line 298 of file json.hpp.

### 9.24.2 Member Typedef Documentation

#### 9.24.2.1 type

```
template<class Default, template< class...  > class Op, class...  Args>
using detail::detector< Default, void_t< Op< Args...  > >, Op, Args...  >::type = Op<Args...>
```

Definition at line 301 of file json.hpp.

#### 9.24.2.2 value_t

```
template<class Default, template< class...  > class Op, class...  Args>
using detail::detector< Default, void_t< Op< Args...  > >, Op, Args...  >::value_t = std::↩
true_type
```

Definition at line 300 of file json.hpp.

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

- include/External/json.hpp

## 9.25 detail::dtoa_impl::diyfp Struct Reference

#### Public Member Functions

- constexpr diyfp (std::uint64_t f_, int e_) noexcept

#### Static Public Member Functions

- static diyfp sub (const diyfp &x, const diyfp &y) noexcept
  *returns x - y*
- static diyfp mul (const diyfp &x, const diyfp &y) noexcept
  *returns x ∗ y*
- static diyfp normalize (diyfp x) noexcept
  *normalize x such that the significand is >= 2^(q-1)*
- static diyfp normalize_to (const diyfp &x, const int target_exponent) noexcept
  *normalize x such that the result has the exponent E*

#### Public Attributes

- std::uint64_t f = 0
- int e = 0

#### Static Public Attributes

- static constexpr int kPrecision = 64

### 9.25.1 Detailed Description

Definition at line 17792 of file json.hpp.

### 9.25.2 Constructor & Destructor Documentation

#### 9.25.2.1 diyfp()

```
detail::dtoa_impl::diyfp::diyfp (
            std::uint64_t f_,
            int e_)  [inline], [constexpr], [noexcept]
```

Definition at line 17799 of file json.hpp.

### 9.25.3 Member Function Documentation

#### 9.25.3.1 mul()

```
diyfp detail::dtoa_impl::diyfp::mul (
            const diyfp & x,
            const diyfp & y)  [inline], [static], [noexcept]
```

returns x ∗ y

**Note**

The result is rounded. (Only the upper q bits are returned.)

Definition at line 17817 of file json.hpp.

#### 9.25.3.2 normalize()

```
diyfp detail::dtoa_impl::diyfp::normalize (
            diyfp x)  [inline], [static], [noexcept]
```

normalize x such that the significand is >= 2^(q-1)

**Precondition**

x.f != 0

Definition at line 17882 of file json.hpp.

#### 9.25.3.3 normalize_to()

```
diyfp detail::dtoa_impl::diyfp::normalize_to (
            const diyfp & x,
            const int target_exponent)  [inline], [static], [noexcept]
```

normalize x such that the result has the exponent E

**Precondition**

e >= x.e and the upper e - x.e bits of x.f must be zero.

Definition at line 17899 of file json.hpp.

#### 9.25.3.4 sub()

```
diyfp detail::dtoa_impl::diyfp::sub (
            const diyfp & x,
            const diyfp & y)  [inline], [static], [noexcept]
```

returns x - y

**Precondition**

x.e == y.e and x.f >= y.f

Definition at line 17805 of file json.hpp.

### 9.25.4 Member Data Documentation

#### 9.25.4.1 e

```
int detail::dtoa_impl::diyfp::e = 0
```

Definition at line 17797 of file json.hpp.

#### 9.25.4.2 f

```
std::uint64_t detail::dtoa_impl::diyfp::f = 0
```

Definition at line 17796 of file json.hpp.

#### 9.25.4.3 kPrecision

```
int detail::dtoa_impl::diyfp::kPrecision = 64  [static], [constexpr]
```

Definition at line 17794 of file json.hpp.

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

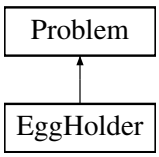
- include/External/json.hpp

## 9.26 EggHolder Class Reference

Implements the Egg Holder benchmark function.

`#include <EggHolder.h>`

Inheritance diagram for EggHolder:




### Public Member Functions

- double evaluate (const std::vector< double > &x) const override

  *Evaluates the fitness of a candidate solution.*

### Public Member Functions inherited from Problem

- Problem (double lb, double ub, const std::string_view n)

  *Constructs a Problem instance.*
- virtual ~**Problem** ()=default

  *Virtual destructor for safe polymorphic cleanup.*

- double getLowerBound () const
- double getUpperBound () const
- const std::string getName () const

### Additional Inherited Members

### Protected Attributes inherited from Problem

- const double lowerBound

  *Lower bound of the search space.*
- const double upperBound

  *Upper bound of the search space.*
- const std::string name

  *Name of the benchmark function.*

### 9.26.1 Detailed Description

Implements the Egg Holder benchmark function.

Definition at line 21 of file EggHolder.h.

### 9.26.2 Constructor & Destructor Documentation

#### 9.26.2.1 EggHolder()

```
EggHolder::EggHolder ()  [inline]
```

Definition at line 32 of file EggHolder.h.

### 9.26.3 Member Function Documentation

#### 9.26.3.1 evaluate()

```
double EggHolder::evaluate (
            const std::vector< double > & x) const  [inline], [override], [virtual]
```

Evaluates the fitness of a candidate solution.

- This is a pure virtual function that must be implemented by specific benchmark functions (e.g., Ackley, Rosenbrock).

- **Parameters**

| | |
|---|---|
| *x* | The solution vector to evaluate. |

**Returns**

The scalar fitness value (cost).

Implements Problem.

Definition at line 34 of file EggHolder.h.

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

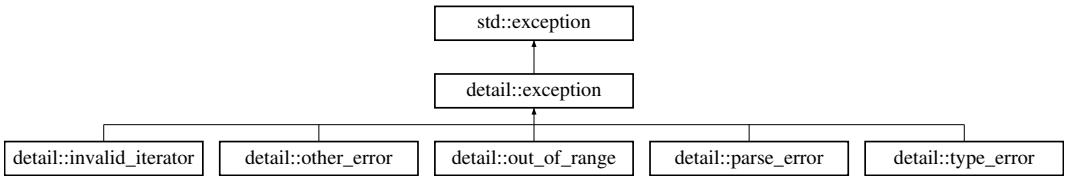
- include/Problem/EggHolder.h

## 9.27 detail::exception Class Reference

general exception of the basic_json class

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

Inheritance diagram for detail::exception:



### Public Member Functions

- const char * what () const noexcept override
  *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_)
- static std::string diagnostics (std::nullptr_t)
- template<typename BasicJsonType>
  static std::string diagnostics (const BasicJsonType *leaf_element)

## 9.27.1 Detailed Description

general exception of the basic_json class

**See also**

https://json.nlohmann.me/api/basic_json/exception/

Definition at line 4558 of file json.hpp.

## 9.27.2 Constructor & Destructor Documentation

### 9.27.2.1 exception()

```
detail::exception::exception (
            int id_,
            const char * what_arg)  [inline], [protected]
```

Definition at line 4572 of file json.hpp.

## 9.27.3 Member Function Documentation

### 9.27.3.1 diagnostics() [1/2]

```
template<typename BasicJsonType>
std::string detail::exception::diagnostics (
            const BasicJsonType * leaf_element)  [inline], [static], [protected]
```

Definition at line 4585 of file json.hpp.

#### 9.27.3.2 diagnostics() [2/2]

```
std::string detail::exception::diagnostics (
            std::nullptr_t )  [inline], [static], [protected]
```

Definition at line 4579 of file json.hpp.

#### 9.27.3.3 name()

```
std::string detail::exception::name (
            const std::string & ename,
            int id_)  [inline], [static], [protected]
```

Definition at line 4574 of file json.hpp.

#### 9.27.3.4 what()

```
const char * detail::exception::what () const  [inline], [override], [noexcept]
```

returns the explanatory string

Definition at line 4562 of file json.hpp.

### 9.27.4 Member Data Documentation

#### 9.27.4.1 id

```
const int detail::exception::id
```

the id of the exception

Definition at line 4568 of file json.hpp.

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

- include/External/json.hpp

## 9.28 Experiment Class Reference

#### Public Member Functions

- Experiment (std::string name, int problemType, int popSize, int dims, unsigned int seed, int lower, int upper)
- void runExperiment ()
- const std::string & getName () const
- const std::vector< double > & getFitness () const
- double getWallTime () const

### 9.28.1 Detailed Description

Definition at line 11 of file Experiment.h.

### 9.28.2 Constructor & Destructor Documentation

#### 9.28.2.1 Experiment()

```
Experiment::Experiment (
            std::string name,
            int problemType,
            int popSize,
            int dims,
            unsigned int seed,
            int lower,
            int upper)  [inline]
```

Definition at line 23 of file Experiment.h.

### 9.28.3 Member Function Documentation

#### 9.28.3.1 getFitness()

```
const std::vector< double > & Experiment::getFitness () const  [inline]
```

Definition at line 44 of file Experiment.h.

#### 9.28.3.2 getName()

```
const std::string & Experiment::getName () const  [inline]
```

Definition at line 43 of file Experiment.h.

#### 9.28.3.3 getWallTime()

```
double Experiment::getWallTime () const  [inline]
```

Definition at line 45 of file Experiment.h.

#### 9.28.3.4 runExperiment()

```
void Experiment::runExperiment ()
```

Definition at line 19 of file Experiment.cpp.

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

- include/Experiment.h
- src/Experiment.cpp

## 9.29 ExperimentConfig Struct Reference

Container for all parameters required to execute a benchmark run.

`#include <Config.h>`

### Public Attributes

- std::string experimentName

  *Identifier for the specific experiment run.*
- int problemType

  *ID of the benchmark problem to instantiate.*
- int dimensions

  *Number of dimensions for the problem space.*
- double lower

  *Global lower bound override.*
- double upper

  *Global upper bound override.*
- int seed

  *Random number generator seed for reproducibility.*
- std::string optimizer

  *Name/Type of the optimizer algorithm to use.*
- int maxIterations

  *Termination criteria: maximum evaluation cycles.*
- double neighborDelta

  *Step size for neighborhood exploration (Local Search only).*
- int numNeighbors

  *Number of neighbors to sample per iteration (Local Search only).*

### 9.29.1 Detailed Description

Container for all parameters required to execute a benchmark run.

- This structure is typically populated by ExperimentRunner from JSON and passed to factories and runners to initialize the experiment state.

Definition at line 21 of file Config.h.

### 9.29.2 Member Data Documentation

#### 9.29.2.1 dimensions

`int ExperimentConfig::dimensions`

Number of dimensions for the problem space.

Definition at line 24 of file Config.h.

#### 9.29.2.2 experimentName

```
std::string ExperimentConfig::experimentName
```

Identifier for the specific experiment run.

Definition at line 22 of file Config.h.

#### 9.29.2.3 lower

```
double ExperimentConfig::lower
```

Global lower bound override.

Definition at line 25 of file Config.h.

#### 9.29.2.4 maxIterations

```
int ExperimentConfig::maxIterations
```

Termination criteria: maximum evaluation cycles.

Definition at line 29 of file Config.h.

#### 9.29.2.5 neighborDelta

```
double ExperimentConfig::neighborDelta
```

Step size for neighborhood exploration (Local Search only).

Definition at line 30 of file Config.h.

#### 9.29.2.6 numNeighbors

```
int ExperimentConfig::numNeighbors
```

Number of neighbors to sample per iteration (Local Search only).

Definition at line 31 of file Config.h.

#### 9.29.2.7 optimizer

```
std::string ExperimentConfig::optimizer
```

Name/Type of the optimizer algorithm to use.

Definition at line 28 of file Config.h.

#### 9.29.2.8 problemType

```
int ExperimentConfig::problemType
```

ID of the benchmark problem to instantiate.

Definition at line 23 of file Config.h.

#### 9.29.2.9 seed

```
int ExperimentConfig::seed
```

Random number generator seed for reproducibility.

Definition at line 27 of file Config.h.

#### 9.29.2.10 upper

```
double ExperimentConfig::upper
```

Global upper bound override.

Definition at line 26 of file Config.h.

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

- include/Config.h

## 9.30 detail::utility_internal::Extend< Seq, SeqSize, Rem > Struct Template Reference

### 9.30.1 Detailed Description

**template<typename Seq, size_t SeqSize, size_t Rem>**
**struct detail::utility_internal::Extend< Seq, SeqSize, Rem >**

Definition at line 3274 of file json.hpp.

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

- include/External/json.hpp

## 9.31 detail::utility_internal::Extend< integer_sequence< T, Ints... >, SeqSize, 0 > Struct Template Reference

**Public Types**

- using type = integer_sequence < T, Ints..., (Ints + SeqSize)... >

### 9.31.1 Detailed Description

**template<typename T, T... Ints, size_t SeqSize>**
**struct detail::utility_internal::Extend< integer_sequence< T, Ints... >, SeqSize, 0 >**

Definition at line 3278 of file json.hpp.

### 9.31.2 Member Typedef Documentation

#### 9.31.2.1 type

```
template<typename T, T...  Ints, size_t SeqSize>
using detail::utility_internal::Extend< integer_sequence< T, Ints...  >, SeqSize, 0 >::type =
integer_sequence < T, Ints..., (Ints + SeqSize)...  >
```

Definition at line 3280 of file json.hpp.

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

- include/External/json.hpp

## 9.32 detail::utility_internal::Extend< integer_sequence< T, Ints... >, SeqSize, 1 > Struct Template Reference

**Public Types**

- using type = integer_sequence < T, Ints..., (Ints + SeqSize)..., 2 * SeqSize >

### 9.32.1 Detailed Description

**template<typename T, T... Ints, size_t SeqSize>**
**struct detail::utility_internal::Extend< integer_sequence< T, Ints... >, SeqSize, 1 >**

Definition at line 3284 of file json.hpp.

### 9.32.2 Member Typedef Documentation

#### 9.32.2.1 type

```
template<typename T, T...  Ints, size_t SeqSize>
using detail::utility_internal::Extend< integer_sequence< T, Ints...  >, SeqSize, 1 >::type =
integer_sequence < T, Ints..., (Ints + SeqSize)..., 2 * SeqSize >
```

Definition at line 3286 of file json.hpp.

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

- include/External/json.hpp

# 9.33 detail::external_constructor< value_t > Struct Template Reference

## 9.33.1 Detailed Description

**template<value_t>**
**struct detail::external_constructor< value_t >**

Definition at line 5752 of file json.hpp.

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

- include/External/json.hpp

# 9.34 detail::external_constructor< value_t::array > Struct 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< !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)

## 9.34.1 Detailed Description

Definition at line 5862 of file json.hpp.

## 9.34.2 Member Function Documentation

### 9.34.2.1 construct() [1/5]

```
template<typename BasicJsonType, typename CompatibleArrayType, enable_if_t< !std::is_same<
CompatibleArrayType, typename BasicJsonType::array_t >::value, int > = 0>
void detail::external_constructor< value_t::array >::construct (
            BasicJsonType & j,
            const CompatibleArrayType & arr)  [inline], [static]
```

Definition at line 5887 of file json.hpp.

#### 9.34.2.2 construct() [2/5]

```
template<typename BasicJsonType, typename T, enable_if_t< std::is_convertible< T, BasicJson↩
Type >::value, int > = 0>
void detail::external_constructor< value_t::array >::construct (
            BasicJsonType & j,
            const std::valarray< T > & arr)  [inline], [static]
```

Definition at line 5916 of file json.hpp.

#### 9.34.2.3 construct() [3/5]

```
template<typename BasicJsonType>
void detail::external_constructor< value_t::array >::construct (
            BasicJsonType & j,
            const std::vector< bool > & arr)  [inline], [static]
```

Definition at line 5900 of file json.hpp.

#### 9.34.2.4 construct() [4/5]

```
template<typename BasicJsonType>
void detail::external_constructor< value_t::array >::construct (
            BasicJsonType & j,
            const typename BasicJsonType::array_t & arr)  [inline], [static]
```

Definition at line 5865 of file json.hpp.

#### 9.34.2.5 construct() [5/5]

```
template<typename BasicJsonType>
void detail::external_constructor< value_t::array >::construct (
            BasicJsonType & j,
            typename BasicJsonType::array_t && arr)  [inline], [static]
```

Definition at line 5875 of file json.hpp.

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

- include/External/json.hpp

## 9.35 detail::external_constructor< value_t::binary > Struct Reference

### Static Public Member Functions

- template<typename BasicJsonType>
  static void construct (BasicJsonType &j, const typename BasicJsonType::binary_t &b)
- template<typename BasicJsonType>
  static void construct (BasicJsonType &j, typename BasicJsonType::binary_t &&b)

### 9.35.1 Detailed Description

Definition at line 5801 of file json.hpp.

### 9.35.2 Member Function Documentation

#### 9.35.2.1 construct() [1/2]

```
template<typename BasicJsonType>
void detail::external_constructor< value_t::binary >::construct (
            BasicJsonType & j,
            const typename BasicJsonType::binary_t & b)  [inline], [static]
```

Definition at line 5804 of file json.hpp.

#### 9.35.2.2 construct() [2/2]

```
template<typename BasicJsonType>
void detail::external_constructor< value_t::binary >::construct (
            BasicJsonType & j,
            typename BasicJsonType::binary_t && b)  [inline], [static]
```

Definition at line 5813 of file json.hpp.

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

- include/External/json.hpp

## 9.36 detail::external_constructor< value_t::boolean > Struct Reference

**Static Public Member Functions**

- template<typename BasicJsonType>
  static void construct (BasicJsonType &j, typename BasicJsonType::boolean_t b) noexcept

### 9.36.1 Detailed Description

Definition at line 5755 of file json.hpp.

### 9.36.2 Member Function Documentation

#### 9.36.2.1 construct()

```
template<typename BasicJsonType>
void detail::external_constructor< value_t::boolean >::construct (
            BasicJsonType & j,
            typename BasicJsonType::boolean_t b)  [inline], [static], [noexcept]
```

Definition at line 5758 of file json.hpp.

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

- include/External/json.hpp

# 9.37 detail::external_constructor< value_t::number_float > Struct Reference

### Static Public Member Functions

- template<typename BasicJsonType>
  static void construct (BasicJsonType &j, typename BasicJsonType::number_float_t val) noexcept

## 9.37.1 Detailed Description

Definition at line 5823 of file json.hpp.

## 9.37.2 Member Function Documentation

### 9.37.2.1 construct()

```
template<typename BasicJsonType>
void detail::external_constructor< value_t::number_float >::construct (
            BasicJsonType & j,
            typename BasicJsonType::number_float_t val)  [inline], [static], [noexcept]
```

Definition at line 5826 of file json.hpp.

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

- include/External/json.hpp

# 9.38 detail::external_constructor< value_t::number_integer > Struct Reference

### Static Public Member Functions

- template<typename BasicJsonType>
  static void construct (BasicJsonType &j, typename BasicJsonType::number_integer_t val) noexcept

## 9.38.1 Detailed Description

Definition at line 5849 of file json.hpp.

### 9.38.2 Member Function Documentation

#### 9.38.2.1 construct()

```
template<typename BasicJsonType>
void detail::external_constructor< value_t::number_integer >::construct (
            BasicJsonType & j,
            typename BasicJsonType::number_integer_t val)  [inline], [static], [noexcept]
```

Definition at line 5852 of file json.hpp.

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

- include/External/json.hpp

## 9.39 detail::external_constructor< value_t::number_unsigned > Struct Reference

#### Static Public Member Functions

- template<typename BasicJsonType>
  static void construct (BasicJsonType &j, typename BasicJsonType::number_unsigned_t val) noexcept

### 9.39.1 Detailed Description

Definition at line 5836 of file json.hpp.

### 9.39.2 Member Function Documentation

#### 9.39.2.1 construct()

```
template<typename BasicJsonType>
void detail::external_constructor< value_t::number_unsigned >::construct (
            BasicJsonType & j,
            typename BasicJsonType::number_unsigned_t val)  [inline], [static], [noexcept]
```

Definition at line 5839 of file json.hpp.

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

- include/External/json.hpp

# 9.40 detail::external_constructor< value_t::object > Struct 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< !std::is_same< CompatibleObjectType, typename BasicJsonType::object_t >::value, int > = 0>
  static void construct (BasicJsonType &j, const CompatibleObjectType &obj)

## 9.40.1 Detailed Description

Definition at line 5932 of file json.hpp.

## 9.40.2 Member Function Documentation

### 9.40.2.1 construct() [1/3]

```
template<typename BasicJsonType, typename CompatibleObjectType, enable_if_t< !std::is_same<
CompatibleObjectType, typename BasicJsonType::object_t >::value, int > = 0>
void detail::external_constructor< value_t::object >::construct (
            BasicJsonType & j,
            const CompatibleObjectType & obj)  [inline], [static]
```

Definition at line 5956 of file json.hpp.

### 9.40.2.2 construct() [2/3]

```
template<typename BasicJsonType>
void detail::external_constructor< value_t::object >::construct (
            BasicJsonType & j,
            const typename BasicJsonType::object_t & obj)  [inline], [static]
```

Definition at line 5935 of file json.hpp.

### 9.40.2.3 construct() [3/3]

```
template<typename BasicJsonType>
void detail::external_constructor< value_t::object >::construct (
            BasicJsonType & j,
            typename BasicJsonType::object_t && obj)  [inline], [static]
```

Definition at line 5945 of file json.hpp.

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

- include/External/json.hpp

# 9.41 detail::external_constructor< value_t::string > Struct 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)
- template<typename BasicJsonType, typename CompatibleStringType, enable_if_t< !std::is_same< CompatibleStringType, typename BasicJsonType::string_t >::value, int > = 0>
  static void construct (BasicJsonType &j, const CompatibleStringType &str)

## 9.41.1 Detailed Description

Definition at line 5768 of file json.hpp.

## 9.41.2 Member Function Documentation

### 9.41.2.1 construct() [1/3]

```
template<typename BasicJsonType, typename CompatibleStringType, enable_if_t< !std::is_same<
CompatibleStringType, typename BasicJsonType::string_t >::value, int > = 0>
void detail::external_constructor< value_t::string >::construct (
            BasicJsonType & j,
            const CompatibleStringType & str)  [inline], [static]
```

Definition at line 5791 of file json.hpp.

### 9.41.2.2 construct() [2/3]

```
template<typename BasicJsonType>
void detail::external_constructor< value_t::string >::construct (
            BasicJsonType & j,
            const typename BasicJsonType::string_t & s)  [inline], [static]
```

Definition at line 5771 of file json.hpp.

### 9.41.2.3 construct() [3/3]

```
template<typename BasicJsonType>
void detail::external_constructor< value_t::string >::construct (
            BasicJsonType & j,
            typename BasicJsonType::string_t && s)  [inline], [static]
```

Definition at line 5780 of file json.hpp.

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

- include/External/json.hpp

## 9.42 detail::file_input_adapter Class Reference

`#include <json.hpp>`

### Public Types

- using char_type = char

### Public Member Functions

- file_input_adapter (std::FILE *f) noexcept
- **file_input_adapter** (const file_input_adapter &)=delete
- **file_input_adapter** (file_input_adapter &&) noexcept=default
- file_input_adapter & **operator=** (const file_input_adapter &)=delete
- file_input_adapter & **operator=** (file_input_adapter &&)=delete
- std::char_traits< char >::int_type get_character () noexcept
- template<class T>
  std::size_t get_elements (T *dest, std::size_t count=1)

### 9.42.1 Detailed Description

Input adapter for stdio file access. This adapter read only 1 byte and do not use any buffer. This adapter is a very low level adapter.

Definition at line 6561 of file json.hpp.

### 9.42.2 Member Typedef Documentation

#### 9.42.2.1 char_type

```
using detail::file_input_adapter::char_type = char
```

Definition at line 6564 of file json.hpp.

### 9.42.3 Constructor & Destructor Documentation

#### 9.42.3.1 file_input_adapter()

```
detail::file_input_adapter::file_input_adapter (
            std::FILE * f)  [inline], [explicit], [noexcept]
```

Definition at line 6567 of file json.hpp.

### 9.42.4 Member Function Documentation

#### 9.42.4.1 get_character()

```
std::char_traits< char >::int_type detail::file_input_adapter::get_character ()  [inline],
[noexcept]
```

Definition at line 6580 of file json.hpp.

#### 9.42.4.2 get_elements()

```
template<class T>
std::size_t detail::file_input_adapter::get_elements (
            T * dest,
            std::size_t count = 1)  [inline]
```

Definition at line 6587 of file json.hpp.

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

- include/External/json.hpp

## 9.43 detail::from_json_fn Struct Reference

### Public Member Functions

- template<typename BasicJsonType, typename T>
  auto operator() (const BasicJsonType &j, T &&val) const noexcept(noexcept(from_json(j, std::forward< T >(val)))) -> decltype(from_json(j, std::forward< T >(val)))

### 9.43.1 Detailed Description

Definition at line 5394 of file json.hpp.

### 9.43.2 Member Function Documentation

#### 9.43.2.1 operator()()

```
template<typename BasicJsonType, typename T>
auto detail::from_json_fn::operator() (
            const BasicJsonType & j,
            T && val) const -> decltype(from_json(j, std::forward< T >(val)))  [inline],
[noexcept]
```

Definition at line 5397 of file json.hpp.

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

- include/External/json.hpp

## 9.44 detail::utility_internal::Gen< T, N > Struct Template Reference

**Public Types**

- using type

### 9.44.1 Detailed Description

**template<typename T, size_t N>**
**struct detail::utility_internal::Gen< T, N >**

Definition at line 3292 of file json.hpp.

### 9.44.2 Member Typedef Documentation

#### 9.44.2.1 type

```
template<typename T, size_t N>
using detail::utility_internal::Gen< T, N >::type
```

**Initial value:**

```
typename Extend < typename Gen < T, N / 2 >::type, N / 2, N % 2 >::type
```

Definition at line 3294 of file json.hpp.

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

- include/External/json.hpp

## 9.45 detail::utility_internal::Gen< T, 0 > Struct Template Reference

**Public Types**

- using type = integer_sequence<T>

### 9.45.1 Detailed Description

**template<typename T>**
**struct detail::utility_internal::Gen< T, 0 >**

Definition at line 3299 of file json.hpp.

### 9.45.2 Member Typedef Documentation

#### 9.45.2.1 type

```
template<typename T>
using detail::utility_internal::Gen< T, 0 >::type = integer_sequence<T>
```

Definition at line 3301 of file json.hpp.

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

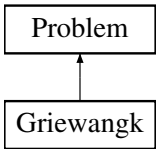
- include/External/json.hpp

## 9.46 Griewangk Class Reference

Implements the Griewangk benchmark function.

```
#include <Griewangk.h>
```

Inheritance diagram for Griewangk:




#### Public Member Functions

- double evaluate (const std::vector< double > &x) const override

  *Evaluates the fitness of a candidate solution.*

### Public Member Functions inherited from Problem

- Problem (double lb, double ub, const std::string_view n)

  *Constructs a Problem instance.*
- virtual ~**Problem** ()=default

  *Virtual destructor for safe polymorphic cleanup.*

- double getLowerBound () const
- double getUpperBound () const
- const std::string getName () const

**Additional Inherited Members**

## Protected Attributes inherited from Problem

- const double lowerBound

  *Lower bound of the search space.*
- const double upperBound

  *Upper bound of the search space.*
- const std::string name

  *Name of the benchmark function.*

### 9.46.1 Detailed Description

Implements the Griewangk benchmark function.

Definition at line 20 of file Griewangk.h.

### 9.46.2 Constructor & Destructor Documentation

#### 9.46.2.1 Griewangk()

```
Griewangk::Griewangk () [inline]
```

Definition at line 27 of file Griewangk.h.

### 9.46.3 Member Function Documentation

#### 9.46.3.1 evaluate()

```
double Griewangk::evaluate (
            const std::vector< double > & x) const [inline], [override], [virtual]
```

Evaluates the fitness of a candidate solution.

- This is a pure virtual function that must be implemented by specific benchmark functions (e.g., Ackley, Rosenbrock).
- **Parameters**

| | |
|---|---|
| *x* | The solution vector to evaluate. |

**Returns**

The scalar fitness value (cost).

Implements Problem.

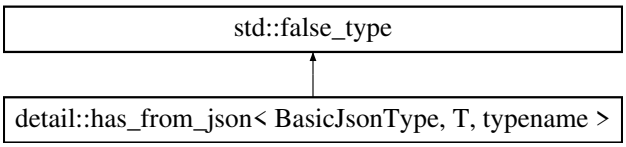
Definition at line 29 of file Griewangk.h.

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

- include/Problem/Griewangk.h

## 9.47 detail::has_from_json< BasicJsonType, T, typename > Struct Template Reference

Inheritance diagram for detail::has_from_json< BasicJsonType, T, typename >:




### 9.47.1 Detailed Description

**template<typename BasicJsonType, typename T, typename = void>**
**struct detail::has_from_json< BasicJsonType, T, typename >**

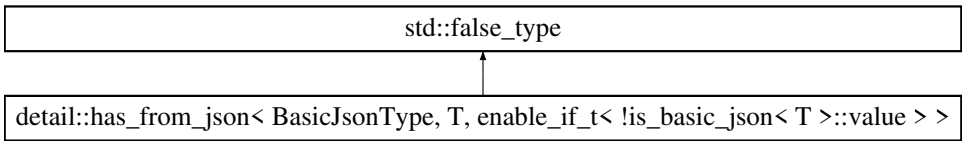
Definition at line 3660 of file json.hpp.

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

- include/External/json.hpp

## 9.48 detail::has_from_json< BasicJsonType, T, enable_if_t< !is_basic_json< T >::value > > Struct Template Reference

Inheritance diagram for detail::has_from_json< BasicJsonType, T, enable_if_t< !is_basic_json< T >::value > >:




#### Public Types

- using serializer = typename BasicJsonType::template json_serializer<T, void>

#### Static Public Attributes

- static constexpr bool value

### 9.48.1 Detailed Description

**template<typename BasicJsonType, typename T>**
**struct detail::has_from_json< BasicJsonType, T, enable_if_t< !is_basic_json< T >::value > >**

Definition at line 3673 of file json.hpp.

### 9.48.2 Member Typedef Documentation

#### 9.48.2.1 serializer

```
template<typename BasicJsonType, typename T>
using detail::has_from_json< BasicJsonType, T, enable_if_t< !is_basic_json< T >::value >
>::serializer = typename BasicJsonType::template json_serializer<T, void>
```

Definition at line 3675 of file json.hpp.

### 9.48.3 Member Data Documentation

#### 9.48.3.1 value

```
template<typename BasicJsonType, typename T>
bool detail::has_from_json< BasicJsonType, T, enable_if_t< !is_basic_json< T >::value >
>::value  [static], [constexpr]
```

**Initial value:**

```
                        =
is_detected_exact<void, from_json_function, serializer,
const BasicJsonType&, T&>::value
```

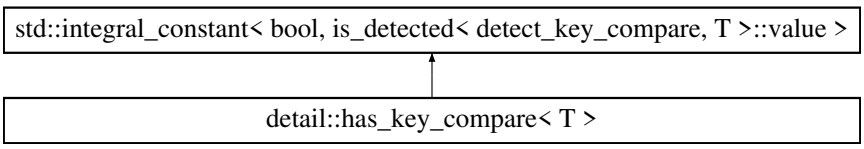
Definition at line 3677 of file json.hpp.

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

- include/External/json.hpp

## 9.49 detail::has_key_compare< T > Struct Template Reference

Inheritance diagram for detail::has_key_compare< T >:




### 9.49.1 Detailed Description

**template<typename T>**
**struct detail::has_key_compare< T >**

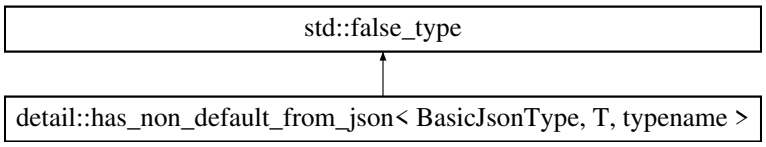
Definition at line 3716 of file json.hpp.

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

- include/External/json.hpp

# 9.50 detail::has_non_default_from_json< BasicJsonType, T, typename > Struct Template Reference

Inheritance diagram for detail::has_non_default_from_json< BasicJsonType, T, typename >:




## 9.50.1 Detailed Description

**template<typename BasicJsonType, typename T, typename = void>**
**struct detail::has_non_default_from_json< BasicJsonType, T, typename >**

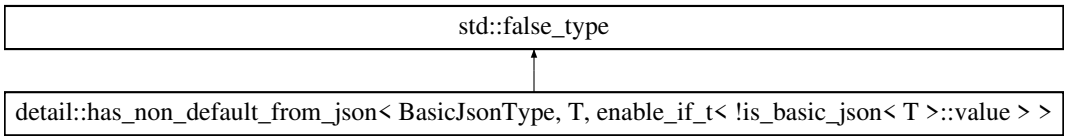
Definition at line 3685 of file json.hpp.

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

- include/External/json.hpp

# 9.51 detail::has_non_default_from_json< BasicJsonType, T, enable_if_t< !is_basic_json< T >::value > > Struct Template Reference

Inheritance diagram for detail::has_non_default_from_json< BasicJsonType, T, enable_if_t< !is_basic_json< T >::value > >:




### Public Types

- using serializer = typename BasicJsonType::template json_serializer<T, void>

### Static Public Attributes

- static constexpr bool value

### 9.51.1 Detailed Description

**template<typename BasicJsonType, typename T>**
**struct detail::has_non_default_from_json< BasicJsonType, T, enable_if_t< !is_basic_json< T >::value > >**

Definition at line 3688 of file json.hpp.

### 9.51.2 Member Typedef Documentation

#### 9.51.2.1 serializer

```
template<typename BasicJsonType, typename T>
using detail::has_non_default_from_json< BasicJsonType, T, enable_if_t< !is_basic_json< T
>::value > >::serializer = typename BasicJsonType::template json_serializer<T, void>
```

Definition at line 3690 of file json.hpp.

### 9.51.3 Member Data Documentation

#### 9.51.3.1 value

```
template<typename BasicJsonType, typename T>
bool detail::has_non_default_from_json< BasicJsonType, T, enable_if_t< !is_basic_json< T
>::value > >::value  [static], [constexpr]
```

**Initial value:**

```
                    =
is_detected_exact<T, from_json_function, serializer,
const BasicJsonType&>::value
```

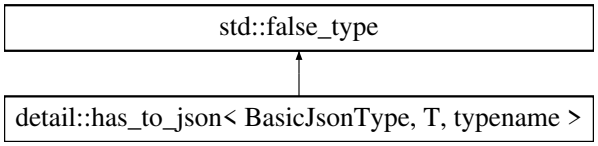
Definition at line 3692 of file json.hpp.

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

- include/External/json.hpp

## 9.52 detail::has_to_json< BasicJsonType, T, typename > Struct Template Reference

Inheritance diagram for detail::has_to_json< BasicJsonType, T, typename >:



### 9.52.1 Detailed Description

**template<typename BasicJsonType, typename T, typename = void>**
**struct detail::has_to_json< BasicJsonType, T, typename >**

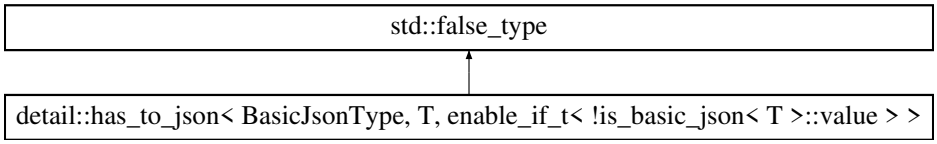
Definition at line 3700 of file json.hpp.

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

- include/External/json.hpp

## 9.53 detail::has_to_json< BasicJsonType, T, enable_if_t< !is_basic_json< T >::value > > Struct Template Reference

Inheritance diagram for detail::has_to_json< BasicJsonType, T, enable_if_t< !is_basic_json< T >::value > >:




#### Public Types

- using serializer = typename BasicJsonType::template json_serializer<T, void>

#### Static Public Attributes

- static constexpr bool value

### 9.53.1 Detailed Description

**template<typename BasicJsonType, typename T>**
**struct detail::has_to_json< BasicJsonType, T, enable_if_t< !is_basic_json< T >::value > >**

Definition at line 3703 of file json.hpp.

### 9.53.2 Member Typedef Documentation

#### 9.53.2.1 serializer

```
template<typename BasicJsonType, typename T>
using detail::has_to_json< BasicJsonType, T, enable_if_t< !is_basic_json< T >::value > >::↩
serializer = typename BasicJsonType::template json_serializer<T, void>
```

Definition at line 3705 of file json.hpp.

### 9.53.3 Member Data Documentation

#### 9.53.3.1 value

```
template<typename BasicJsonType, typename T>
bool detail::has_to_json< BasicJsonType, T, enable_if_t< !is_basic_json< T >::value > >::↩
value  [static], [constexpr]
```

**Initial value:**

```
                        =
is_detected_exact<void, to_json_function, serializer, BasicJsonType&,
T>::value
```

Definition at line 3707 of file json.hpp.

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

- include/External/json.hpp

## 9.54 detail::identity_tag< T > Struct Template Reference

### 9.54.1 Detailed Description

**template<class T>**
**struct detail::identity_tag< T >**

Definition at line 4824 of file json.hpp.

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

- include/External/json.hpp

## 9.55 detail::input_stream_adapter Class Reference

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

#### Public Types

- using char_type = char

#### 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
- input_stream_adapter & **operator=** (input_stream_adapter &&)=delete
- input_stream_adapter (input_stream_adapter &&rhs) noexcept
- std::char_traits< char >::int_type get_character ()
- template<class T>
  std::size_t get_elements (T *dest, std::size_t count=1)

### 9.55.1 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.

Definition at line 6606 of file json.hpp.

### 9.55.2 Member Typedef Documentation

#### 9.55.2.1 char_type

```
using detail::input_stream_adapter::char_type = char
```

Definition at line 6609 of file json.hpp.

### 9.55.3 Constructor & Destructor Documentation

#### 9.55.3.1 ~input_stream_adapter()

```
detail::input_stream_adapter::~input_stream_adapter ()  [inline]
```

Definition at line 6611 of file json.hpp.

#### 9.55.3.2 input_stream_adapter() [1/2]

```
detail::input_stream_adapter::input_stream_adapter (
            std::istream & i)  [inline], [explicit]
```

Definition at line 6621 of file json.hpp.

#### 9.55.3.3 input_stream_adapter() [2/2]

```
detail::input_stream_adapter::input_stream_adapter (
            input_stream_adapter && rhs)  [inline], [noexcept]
```

Definition at line 6630 of file json.hpp.

### 9.55.4 Member Function Documentation

#### 9.55.4.1 get_character()

```
std::char_traits< char >::int_type detail::input_stream_adapter::get_character ()  [inline]
```

Definition at line 6640 of file json.hpp.

#### 9.55.4.2 get_elements()

```
template<class T>
std::size_t detail::input_stream_adapter::get_elements (
            T * dest,
            std::size_t count = 1)  [inline]
```

Definition at line 6652 of file json.hpp.

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

- include/External/json.hpp

## 9.56 detail::integer_sequence< T, Ints > Struct Template Reference

### Public Types

- using value_type = T

### Static Public Member Functions

- static constexpr std::size_t size () noexcept

### 9.56.1 Detailed Description

**template<typename T, T... Ints>**
**struct detail::integer_sequence< T, Ints >**

Definition at line 3253 of file json.hpp.

### 9.56.2 Member Typedef Documentation

#### 9.56.2.1 value_type

```
template<typename T, T...  Ints>
using detail::integer_sequence< T, Ints >::value_type = T
```

Definition at line 3255 of file json.hpp.

### 9.56.3 Member Function Documentation

#### 9.56.3.1 size()

```
template<typename T, T...  Ints>
constexpr std::size_t detail::integer_sequence< T, Ints >::size ()  [inline], [static], [constexpr],
[noexcept]
```

Definition at line 3256 of file json.hpp.

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

- include/External/json.hpp

# 9.57 detail::internal_iterator< BasicJsonType > Struct Template Reference

an iterator value

`#include <json.hpp>`

### Public Attributes

- ∗iterator for JSON objects BasicJsonType::object_t::iterator object_iterator {}
- ∗iterator for JSON arrays BasicJsonType::array_t::iterator array_iterator {}
- ∗generic iterator for all other types primitive_iterator_t primitive_iterator {}

## 9.57.1 Detailed Description

**template<typename BasicJsonType>**
**struct 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.

Definition at line 13640 of file json.hpp.

## 9.57.2 Member Data Documentation

### 9.57.2.1 array_iterator

```
template<typename BasicJsonType>
* iterator for JSON arrays BasicJsonType::array_t::iterator detail::internal_iterator< Basic↩
JsonType >::array_iterator {}
```

Definition at line 13645 of file json.hpp.

### 9.57.2.2 object_iterator

```
template<typename BasicJsonType>
* iterator for JSON objects BasicJsonType::object_t::iterator detail::internal_iterator< Basic↩
JsonType >::object_iterator {}
```

Definition at line 13643 of file json.hpp.

#### 9.57.2.3 primitive_iterator

```
template<typename BasicJsonType>
* generic iterator for all other types primitive_iterator_t detail::internal_iterator< Basic↩
JsonType >::primitive_iterator {}
```

Definition at line 13647 of file json.hpp.

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

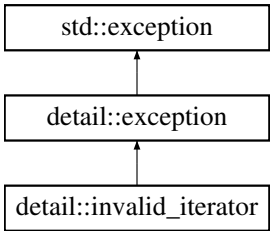
- include/External/json.hpp

## 9.58 detail::invalid_iterator Class Reference

exception indicating errors with iterators

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

Inheritance diagram for detail::invalid_iterator:




### Static Public Member Functions

- template<typename BasicJsonContext, enable_if_t< is_basic_json_context< BasicJsonContext >::value, int > = 0>
  static invalid_iterator create (int id_, const std::string &what_arg, BasicJsonContext context)

### Additional Inherited Members

### Public Member Functions inherited from detail::exception

- const char * what () const noexcept override
  *returns the explanatory string*

### Public Attributes inherited from detail::exception

- const int id
  *the id of the exception*

### Protected Member Functions inherited from detail::exception

- exception (int id_, const char *what_arg)

### Static Protected Member Functions inherited from detail::exception

- static std::string name (const std::string &ename, int id_)
- static std::string diagnostics (std::nullptr_t)
- template<typename BasicJsonType>
  static std::string diagnostics (const BasicJsonType *leaf_element)

## 9.58.1 Detailed Description

exception indicating errors with iterators

**See also**

https://json.nlohmann.me/api/basic_json/invalid_iterator/

Definition at line 4727 of file json.hpp.

## 9.58.2 Member Function Documentation

### 9.58.2.1 create()

```
template<typename BasicJsonContext, enable_if_t< is_basic_json_context< BasicJsonContext
>::value, int > = 0>
invalid_iterator detail::invalid_iterator::create (
            int id_,
            const std::string & what_arg,
            BasicJsonContext context)  [inline], [static]
```

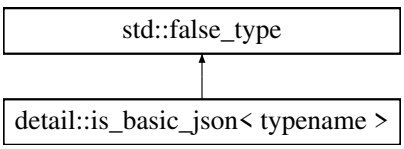
Definition at line 4731 of file json.hpp.

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

- include/External/json.hpp

# 9.59 detail::is_basic_json< typename > Struct Template Reference

Inheritance diagram for detail::is_basic_json< typename >:



### 9.59.1 Detailed Description

**template**<**typename**>
**struct detail::is_basic_json**< **typename** >

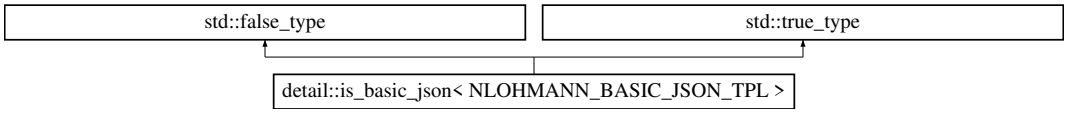
Definition at line 3596 of file json.hpp.

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

- include/External/json.hpp

## 9.60 detail::is_basic_json< NLOHMANN_BASIC_JSON_TPL > Struct Reference

Inheritance diagram for detail::is_basic_json< NLOHMANN_BASIC_JSON_TPL >:




### 9.60.1 Detailed Description

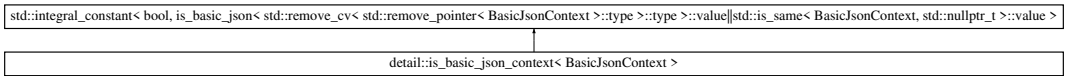
Definition at line 3599 of file json.hpp.

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

- include/External/json.hpp

## 9.61 detail::is_basic_json_context< BasicJsonContext > Struct Template Reference

Inheritance diagram for detail::is_basic_json_context< BasicJsonContext >:




### 9.61.1 Detailed Description

**template**<**typename BasicJsonContext**>
**struct detail::is_basic_json_context**< **BasicJsonContext** >

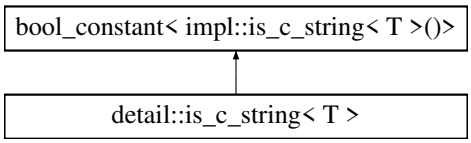
Definition at line 3605 of file json.hpp.

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

- include/External/json.hpp

## 9.62 detail::is_c_string< T > Struct Template Reference

Inheritance diagram for detail::is_c_string< T >:




### 9.62.1 Detailed Description

**template<typename T>**
**struct detail::is_c_string< T >**

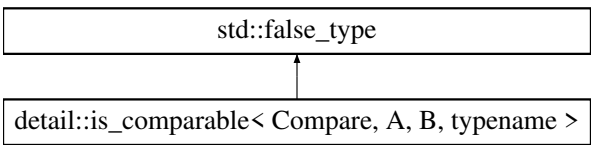
Definition at line 4356 of file json.hpp.

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

- include/External/json.hpp

## 9.63 detail::is_comparable< Compare, A, B, typename > Struct Template Reference

Inheritance diagram for detail::is_comparable< Compare, A, B, typename >:




### 9.63.1 Detailed Description

**template<typename Compare, typename A, typename B, typename = void>**
**struct detail::is_comparable< Compare, A, B, typename >**

Definition at line 4144 of file json.hpp.

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

- include/External/json.hpp

## 9.64 detail::is_comparable< Compare, A, B, enable_if_t< !is_json_pointer_of< A, B >::value &&std::is_constructible< decltype(std::declval< Compare >()(std::declval< A >(), std::declval< B >()))>::value &&std::is_constructible< decltype(std::declval< Compare >()(std::declval< B >(), std::declval< A >()))>::value > > Struct Template Reference

Inheritance diagram for detail::is_comparable< Compare, A, B, enable_if_t< !is_json_pointer_of< A, B >::value &&std::is_constructible< decltype(std::declval< Compare >()(std::declval< A >(), std::declval< B >()))>::value &&std::is_constructible< decltype(std::declval< Compare >()(std::declval< B >(), std::declval< A >()))>::value > >:

### 9.64.1 Detailed Description

**template<typename Compare, typename A, typename B>**
**struct detail::is_comparable< Compare, A, B, enable_if_t< !is_json_pointer_of< A, B >::value &&std::is_constructible< decltype(std::declval< Compare >()(std::declval< A >(), std::declval< B >()))>::value &&std::is_constructible< decltype(std::declval< Compare >()(std::declval< B >(), std::declval< A >()))>::value > >**

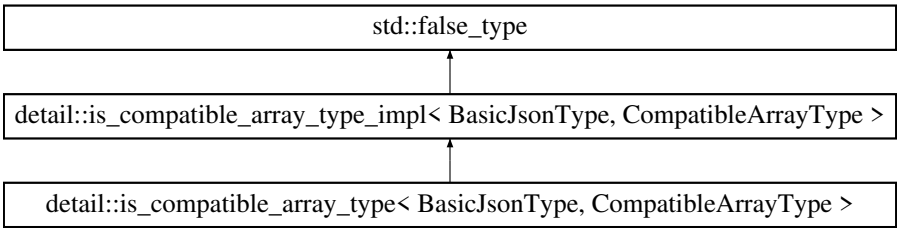
Definition at line 4152 of file json.hpp.

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

- include/External/json.hpp

## 9.65 detail::is_compatible_array_type< BasicJsonType, CompatibleArrayType > Struct Template Reference

Inheritance diagram for detail::is_compatible_array_type< BasicJsonType, CompatibleArrayType >:




### 9.65.1 Detailed Description

**template<typename BasicJsonType, typename CompatibleArrayType>**
**struct detail::is_compatible_array_type< BasicJsonType, CompatibleArrayType >**

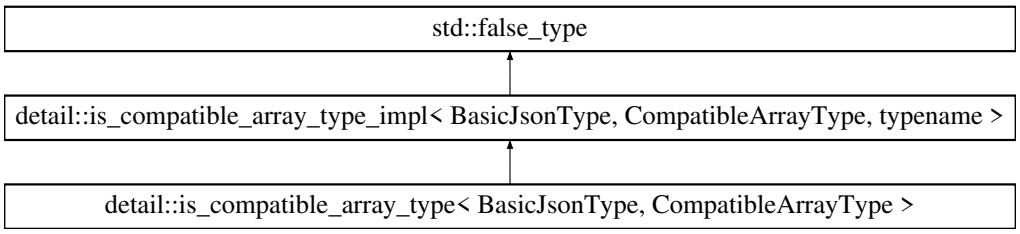
Definition at line 4016 of file json.hpp.

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

- include/External/json.hpp

# 9.66 detail::is_compatible_array_type_impl< BasicJsonType, CompatibleArrayType, typename > Struct Template Reference

Inheritance diagram for detail::is_compatible_array_type_impl< BasicJsonType, CompatibleArrayType, typename >:




## 9.66.1 Detailed Description

**template<typename BasicJsonType, typename CompatibleArrayType, typename = void>**
**struct detail::is_compatible_array_type_impl< BasicJsonType, CompatibleArrayType, typename >**

Definition at line 3998 of file json.hpp.

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

- include/External/json.hpp

# 9.67 detail::is_compatible_array_type_impl< BasicJsonType, CompatibleArrayType, enable_if_t< is_detected< iterator_t, CompatibleArrayType >::value &&is_iterator_traits< iterator_traits< detected_t< iterator_t, CompatibleArrayType > > >::value &&!std::is_same< CompatibleArrayType, detected_t< range_value_t, CompatibleArrayType > >::value > > Struct Template Reference

Inheritance diagram for detail::is_compatible_array_type_impl< BasicJsonType, CompatibleArrayType, enable_if_t< is_detected< iterator_t, CompatibleArrayType >::value &&is_iterator_traits< iterator_traits< detected_t< iterator_t, CompatibleArrayType > > >::value &&!std::is_same< CompatibleArrayType, detected_t< range_value_t, CompatibleArrayType > >::value > >:


### Static Public Attributes

- static constexpr bool value

### 9.67.1 Detailed Description

**template<typename BasicJsonType, typename CompatibleArrayType>**
**struct detail::is_compatible_array_type_impl< BasicJsonType, CompatibleArrayType, enable_if_t< is_detected< iterator_t, CompatibleArrayType >::value &&is_iterator_traits< iterator_traits< detected_t< iterator_t, CompatibleArrayType > > >::value &&!std::is_same< CompatibleArrayType, detected_t< range_value_t, CompatibleArrayType > >::value > >**

Definition at line 4001 of file json.hpp.

### 9.67.2 Member Data Documentation

#### 9.67.2.1 value

```
template<typename BasicJsonType, typename CompatibleArrayType>
bool detail::is_compatible_array_type_impl< BasicJsonType, CompatibleArrayType, enable_if_t<
is_detected< iterator_t, CompatibleArrayType >::value &&is_iterator_traits< iterator_traits<
detected_t< iterator_t, CompatibleArrayType > > >::value &&!std::is_same< CompatibleArray↩
Type, detected_t< range_value_t, CompatibleArrayType > >::value > >::value  [static], [constexpr]
```

**Initial value:**

```
=
is_constructible<BasicJsonType,
range_value_t<CompatibleArrayType»::value
```

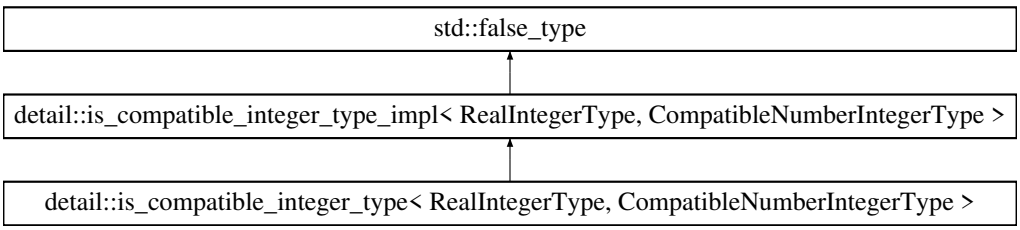
Definition at line 4010 of file json.hpp.

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

- include/External/json.hpp

## 9.68 detail::is_compatible_integer_type< RealIntegerType, CompatibleNumberIntegerType > Struct Template Reference

Inheritance diagram for detail::is_compatible_integer_type< RealIntegerType, CompatibleNumberIntegerType >:




### 9.68.1 Detailed Description

**template<typename RealIntegerType, typename CompatibleNumberIntegerType>**
**struct detail::is_compatible_integer_type< RealIntegerType, CompatibleNumberIntegerType >**

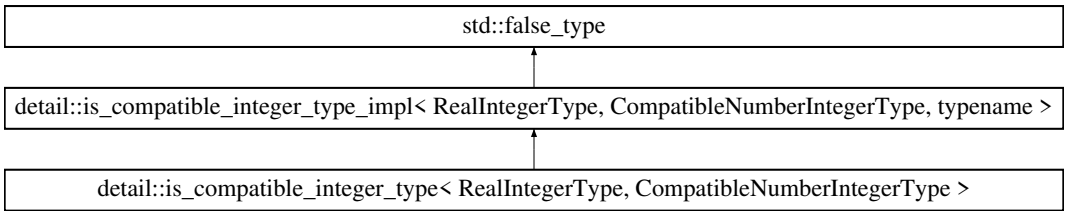
Definition at line 4086 of file json.hpp.

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

- include/External/json.hpp

# 9.69 detail::is_compatible_integer_type_impl< RealIntegerType, CompatibleNumberIntegerType, typename > Struct Template Reference

Inheritance diagram for detail::is_compatible_integer_type_impl< RealIntegerType, CompatibleNumberInteger↩ Type, typename >:




## 9.69.1 Detailed Description

**template<typename RealIntegerType, typename CompatibleNumberIntegerType, typename = void>**
**struct detail::is_compatible_integer_type_impl< RealIntegerType, CompatibleNumberIntegerType, typename >**

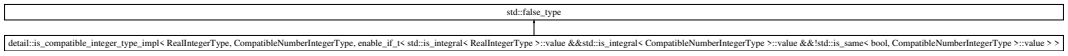
Definition at line 4065 of file json.hpp.

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

- include/External/json.hpp

# 9.70 detail::is_compatible_integer_type_impl< RealIntegerType, CompatibleNumberIntegerType, enable_if_t< std::is_integral< RealIntegerType >::value &&std::is_integral< CompatibleNumberIntegerType >::value &&!std::is_same< bool, CompatibleNumberIntegerType >::value > > Struct Template Reference

Inheritance diagram for detail::is_compatible_integer_type_impl< RealIntegerType, CompatibleNumberInteger↩ Type, enable_if_t< std::is_integral< RealIntegerType >::value &&std::is_integral< CompatibleNumberIntegerType >::value &&!std::is_same< bool, CompatibleNumberIntegerType >::value > >:




### Public Types

- using RealLimits = std::numeric_limits<RealIntegerType>
- using CompatibleLimits = std::numeric_limits<CompatibleNumberIntegerType>

**Static Public Attributes**

- static constexpr auto value

## 9.70.1 Detailed Description

**template<typename RealIntegerType, typename CompatibleNumberIntegerType>**
**struct detail::is_compatible_integer_type_impl< RealIntegerType, CompatibleNumberIntegerType, enable_if_t< std::is_integral< RealIntegerType >::value &&std::is_integral< CompatibleNumberInteger↩ Type >::value &&!std::is_same< bool, CompatibleNumberIntegerType >::value > >**

Definition at line 4068 of file json.hpp.

## 9.70.2 Member Typedef Documentation

### 9.70.2.1 CompatibleLimits

```
template<typename RealIntegerType, typename CompatibleNumberIntegerType>
using detail::is_compatible_integer_type_impl< RealIntegerType, CompatibleNumberIntegerType,
enable_if_t< std::is_integral< RealIntegerType >::value &&std::is_integral< Compatible↩
NumberIntegerType >::value &&!std::is_same< bool, CompatibleNumberIntegerType >::value >
>::CompatibleLimits = std::numeric_limits<CompatibleNumberIntegerType>
```

Definition at line 4076 of file json.hpp.

### 9.70.2.2 RealLimits

```
template<typename RealIntegerType, typename CompatibleNumberIntegerType>
using detail::is_compatible_integer_type_impl< RealIntegerType, CompatibleNumberIntegerType,
enable_if_t< std::is_integral< RealIntegerType >::value &&std::is_integral< Compatible↩
NumberIntegerType >::value &&!std::is_same< bool, CompatibleNumberIntegerType >::value >
>::RealLimits = std::numeric_limits<RealIntegerType>
```

Definition at line 4075 of file json.hpp.

## 9.70.3 Member Data Documentation

### 9.70.3.1 value

```
template<typename RealIntegerType, typename CompatibleNumberIntegerType>
auto detail::is_compatible_integer_type_impl< RealIntegerType, CompatibleNumberIntegerType,
enable_if_t< std::is_integral< RealIntegerType >::value &&std::is_integral< Compatible↩
NumberIntegerType >::value &&!std::is_same< bool, CompatibleNumberIntegerType >::value >
>::value  [static], [constexpr]
```

**Initial value:**

```
                        =
is_constructible<RealIntegerType,
CompatibleNumberIntegerType>::value &&
CompatibleLimits::is_integer &&
RealLimits::is_signed == CompatibleLimits::is_signed
```

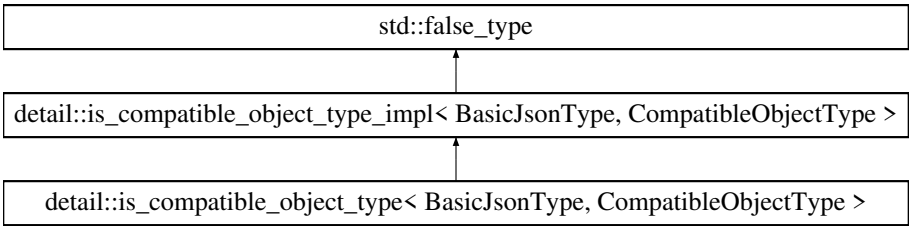
Definition at line 4078 of file json.hpp.

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

- include/External/json.hpp

# 9.71 detail::is_compatible_object_type< BasicJsonType, CompatibleObjectType > Struct Template Reference

Inheritance diagram for detail::is_compatible_object_type< BasicJsonType, CompatibleObjectType >:




## 9.71.1 Detailed Description

**template<typename BasicJsonType, typename CompatibleObjectType>**
**struct detail::is_compatible_object_type< BasicJsonType, CompatibleObjectType >**

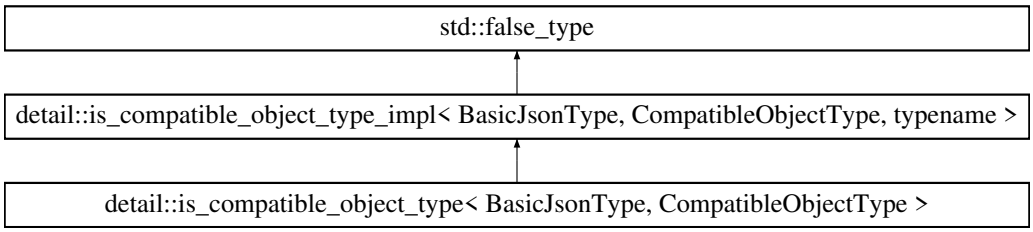
Definition at line 3937 of file json.hpp.

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

- include/External/json.hpp

# 9.72 detail::is_compatible_object_type_impl< BasicJsonType, CompatibleObjectType, typename > Struct Template Reference

Inheritance diagram for detail::is_compatible_object_type_impl< BasicJsonType, CompatibleObjectType, typename >:




## 9.72.1 Detailed Description

**template<typename BasicJsonType, typename CompatibleObjectType, typename = void>**
**struct detail::is_compatible_object_type_impl< BasicJsonType, CompatibleObjectType, typename >**

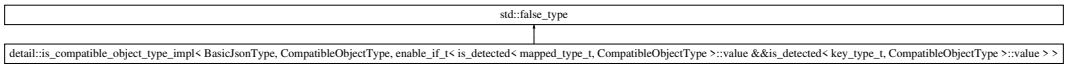
Definition at line 3918 of file json.hpp.

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

- include/External/json.hpp

# 9.73 detail::is_compatible_object_type_impl< BasicJsonType, CompatibleObjectType, enable_if_t< is_detected< mapped_type_t, CompatibleObjectType >::value &&is_detected< key_type_t, CompatibleObjectType >::value > > Struct Template Reference

Inheritance diagram for detail::is_compatible_object_type_impl< BasicJsonType, CompatibleObjectType, enable_if_t< is_detected< mapped_type_t, CompatibleObjectType >::value &&is_detected< key_type_t, CompatibleObjectType >::value > >:




### Public Types

- using object_t = typename BasicJsonType::object_t

### Static Public Attributes

- static constexpr bool value

## 9.73.1 Detailed Description

**template<typename BasicJsonType, typename CompatibleObjectType>**
**struct detail::is_compatible_object_type_impl< BasicJsonType, CompatibleObjectType, enable_if_t< is_detected< mapped_type_t, CompatibleObjectType >::value &&is_detected< key_type_t, CompatibleObjectType >::value > >**

Definition at line 3921 of file json.hpp.

## 9.73.2 Member Typedef Documentation

### 9.73.2.1 object_t

```
template<typename BasicJsonType, typename CompatibleObjectType>
using detail::is_compatible_object_type_impl< BasicJsonType, CompatibleObjectType, enable_if_t< is_detected< mapped_type_t, CompatibleObjectType >::value &&is_detected< key_type_t, CompatibleObjectType >::value > >::object_t = typename BasicJsonType::object_t
```

Definition at line 3926 of file json.hpp.

### 9.73.3 Member Data Documentation

#### 9.73.3.1 value

```
template<typename BasicJsonType, typename CompatibleObjectType>
bool detail::is_compatible_object_type_impl< BasicJsonType, CompatibleObjectType, enable_↩
if_t< is_detected< mapped_type_t, CompatibleObjectType >::value &&is_detected< key_type_t,
CompatibleObjectType >::value > >::value  [static], [constexpr]
```

**Initial value:**

```
                    =
is_constructible<typename object_t::key_type,
typename CompatibleObjectType::key_type>::value &&
is_constructible<typename object_t::mapped_type,
typename CompatibleObjectType::mapped_type>::value
```

Definition at line 3929 of file json.hpp.

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

- include/External/json.hpp

## 9.74 detail::is_compatible_string_type< BasicJsonType, CompatibleStringType > Struct Template Reference

**Static Public Attributes**

- static constexpr auto value

### 9.74.1 Detailed Description

**template<typename BasicJsonType, typename CompatibleStringType>**
**struct detail::is_compatible_string_type< BasicJsonType, CompatibleStringType >**

Definition at line 3974 of file json.hpp.

### 9.74.2 Member Data Documentation

#### 9.74.2.1 value

```
template<typename BasicJsonType, typename CompatibleStringType>
auto detail::is_compatible_string_type< BasicJsonType, CompatibleStringType >::value  [static],
[constexpr]
```

**Initial value:**

```
                    =
is_constructible<typename BasicJsonType::string_t, CompatibleStringType>::value
```

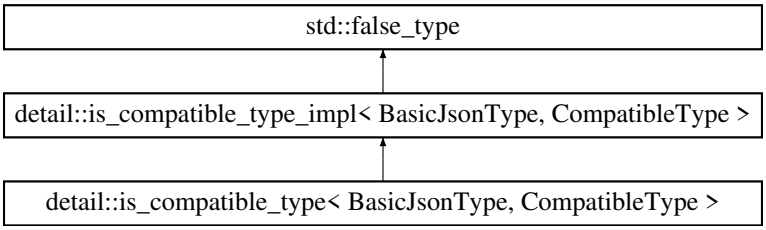
Definition at line 3976 of file json.hpp.

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

- include/External/json.hpp

## 9.75 detail::is_compatible_type< BasicJsonType, CompatibleType > Struct Template Reference

Inheritance diagram for detail::is_compatible_type< BasicJsonType, CompatibleType >:




### 9.75.1 Detailed Description

**template<typename BasicJsonType, typename CompatibleType>**
**struct detail::is_compatible_type< BasicJsonType, CompatibleType >**

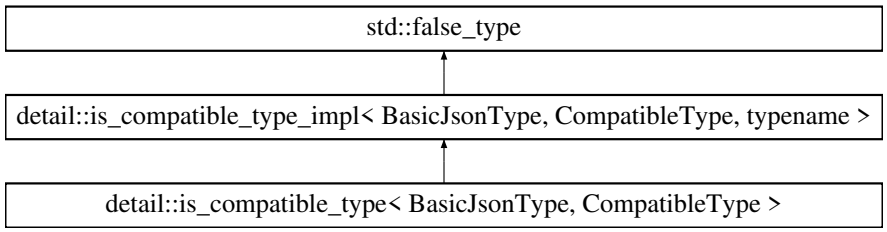
Definition at line 4103 of file json.hpp.

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

- include/External/json.hpp

## 9.76 detail::is_compatible_type_impl< BasicJsonType, CompatibleType, typename > Struct Template Reference

Inheritance diagram for detail::is_compatible_type_impl< BasicJsonType, CompatibleType, typename >:




### 9.76.1 Detailed Description

**template<typename BasicJsonType, typename CompatibleType, typename = void>**
**struct detail::is_compatible_type_impl< BasicJsonType, CompatibleType, typename >**

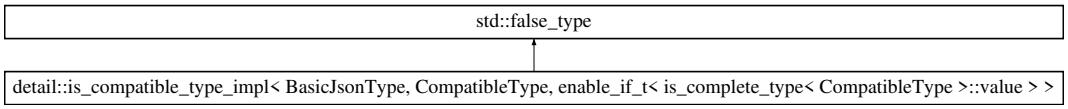
Definition at line 4091 of file json.hpp.

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

- include/External/json.hpp

# 9.77 detail::is_compatible_type_impl< BasicJsonType, CompatibleType, enable_if_t< is_complete_type< CompatibleType >::value > > Struct Template Reference

Inheritance diagram for detail::is_compatible_type_impl< BasicJsonType, CompatibleType, enable_if_t< is_complete_type< CompatibleType >::value > >:




**Static Public Attributes**

- static constexpr bool value

## 9.77.1 Detailed Description

**template<typename BasicJsonType, typename CompatibleType>**
**struct detail::is_compatible_type_impl< BasicJsonType, CompatibleType, enable_if_t< is_complete_type< CompatibleType >::value > >**

Definition at line 4094 of file json.hpp.

## 9.77.2 Member Data Documentation

### 9.77.2.1 value

```
template<typename BasicJsonType, typename CompatibleType>
bool detail::is_compatible_type_impl< BasicJsonType, CompatibleType, enable_if_t< is_complete_type<
CompatibleType >::value > >::value  [static], [constexpr]
```

**Initial value:**

```
=
has_to_json<BasicJsonType, CompatibleType>::value
```

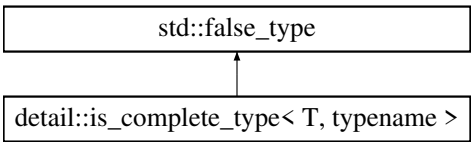
Definition at line 4098 of file json.hpp.

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

- include/External/json.hpp

## 9.78 detail::is_complete_type< T, typename > Struct Template Reference

Inheritance diagram for detail::is_complete_type< T, typename >:




### 9.78.1 Detailed Description

**template<typename T, typename = void>**
**struct detail::is_complete_type< T, typename >**

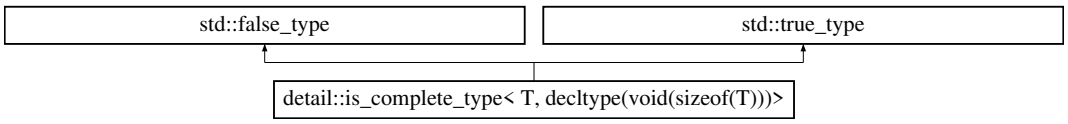
Definition at line 3911 of file json.hpp.

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

- include/External/json.hpp

## 9.79 detail::is_complete_type< T, decltype(void(sizeof(T)))> Struct Template Reference

Inheritance diagram for detail::is_complete_type< T, decltype(void(sizeof(T)))>:




### 9.79.1 Detailed Description

**template<typename T>**
**struct detail::is_complete_type< T, decltype(void(sizeof(T)))>**

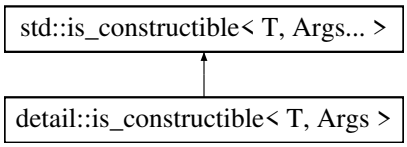
Definition at line 3914 of file json.hpp.

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

- include/External/json.hpp

## 9.80 detail::is_constructible< T, Args > Struct Template Reference

Inheritance diagram for detail::is_constructible< T, Args >:




### 9.80.1 Detailed Description

**template<typename T, typename... Args>**
**struct detail::is_constructible< T, Args >**

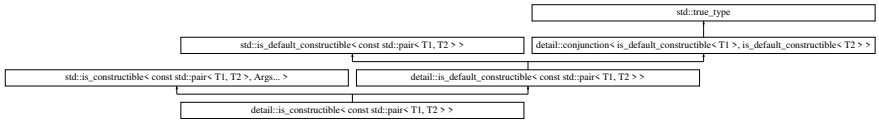
Definition at line 3849 of file json.hpp.

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

- include/External/json.hpp

## 9.81 detail::is_constructible< const std::pair< T1, T2 > > Struct Template Reference

Inheritance diagram for detail::is_constructible< const std::pair< T1, T2 > >:




### 9.81.1 Detailed Description

**template<typename T1, typename T2>**
**struct detail::is_constructible< const std::pair< T1, T2 > >**

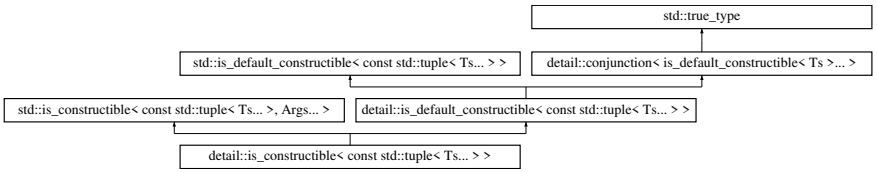
Definition at line 3855 of file json.hpp.

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

- include/External/json.hpp

## 9.82 detail::is_constructible< const std::tuple< Ts... > > Struct Template Reference

Inheritance diagram for detail::is_constructible< const std::tuple< Ts... > >:




### 9.82.1 Detailed Description

**template<typename... Ts>**
**struct detail::is_constructible< const std::tuple< Ts... > >**

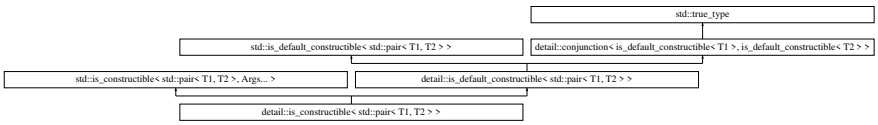
Definition at line 3861 of file json.hpp.

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

- include/External/json.hpp

## 9.83 detail::is_constructible< std::pair< T1, T2 > > Struct Template Reference

Inheritance diagram for detail::is_constructible< std::pair< T1, T2 > >:




### 9.83.1 Detailed Description

**template<typename T1, typename T2>**
**struct detail::is_constructible< std::pair< T1, T2 > >**

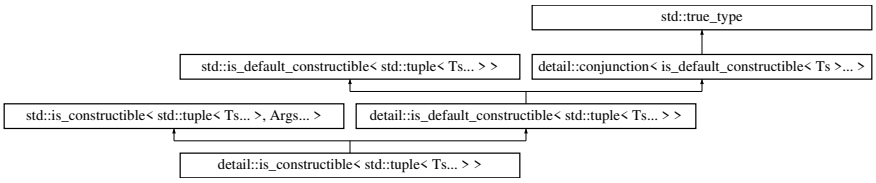
Definition at line 3852 of file json.hpp.

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

- include/External/json.hpp

# 9.84 detail::is_constructible< std::tuple< Ts... > > Struct Template Reference

Inheritance diagram for detail::is_constructible< std::tuple< Ts... > >:




## 9.84.1 Detailed Description

**template<typename... Ts>**
**struct detail::is_constructible< std::tuple< Ts... > >**

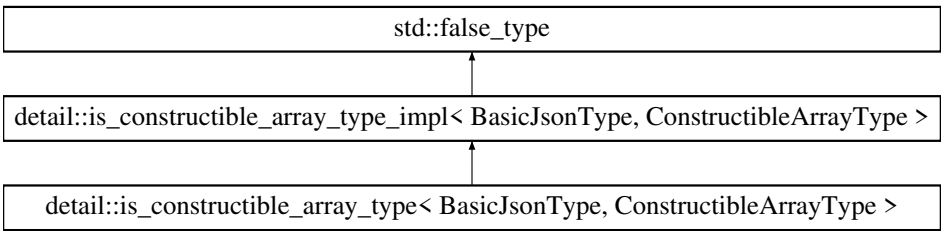
Definition at line 3858 of file json.hpp.

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

- include/External/json.hpp

# 9.85 detail::is_constructible_array_type< BasicJsonType, ConstructibleArrayType > Struct Template Reference

Inheritance diagram for detail::is_constructible_array_type< BasicJsonType, ConstructibleArrayType >:




## 9.85.1 Detailed Description

**template<typename BasicJsonType, typename ConstructibleArrayType>**
**struct detail::is_constructible_array_type< BasicJsonType, ConstructibleArrayType >**

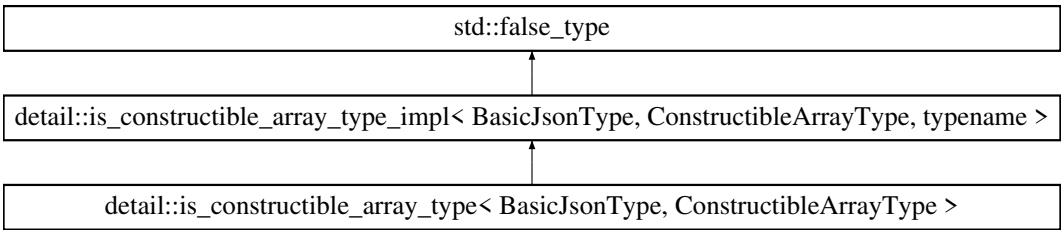
Definition at line 4060 of file json.hpp.

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

- include/External/json.hpp

## 9.86 detail::is_constructible_array_type_impl< BasicJsonType, ConstructibleArrayType, typename > Struct Template Reference

Inheritance diagram for detail::is_constructible_array_type_impl< BasicJsonType, ConstructibleArrayType, typename >:




### 9.86.1 Detailed Description

**template<typename BasicJsonType, typename ConstructibleArrayType, typename = void>**
**struct detail::is_constructible_array_type_impl< BasicJsonType, ConstructibleArrayType, typename >**

Definition at line 4020 of file json.hpp.

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

- include/External/json.hpp

## 9.87 detail::is_constructible_array_type_impl< BasicJsonType, ConstructibleArrayType, enable_if_t< !std::is_same< ConstructibleArrayType, typename BasicJsonType::value_type >::value &&!is_compatible_string_type< BasicJsonType, ConstructibleArrayType >::value &&is_default_constructible< ConstructibleArrayType >::value &&(std::is_move_assignable< ConstructibleArrayType >::value||std::is_copy_assignable< ConstructibleArrayType >::value)&&is_detected< iterator_t, ConstructibleArrayType >::value &&is_iterator_traits< iterator_traits< detected_t< iterator_t, ConstructibleArrayType > > >::value &&is_detected< range_value_t, ConstructibleArrayType >::value &&!std::is_same< ConstructibleArrayType, detected_t< range_value_t, ConstructibleArrayType > >::value &&is_complete_type< detected_t< range_value_t, ConstructibleArrayType > >::value > > Struct Template Reference

Inheritance diagram for detail::is_constructible_array_type_impl< BasicJsonType, ConstructibleArrayType, enable_if_t< !std::is_same< ConstructibleArrayType, typename BasicJsonType::value_type >::value &&!is_compatible_string_type< BasicJsonType, ConstructibleArrayType >::value &&is_default_constructible<

### Public Types

- using value_type = range_value_t<ConstructibleArrayType>

### Static Public Attributes

- static constexpr bool value

## 9.87.1 Detailed Description

**template<typename BasicJsonType, typename ConstructibleArrayType>**
**struct detail::is_constructible_array_type_impl< BasicJsonType, ConstructibleArrayType, enable_if_t< !std::is_same< ConstructibleArrayType, typename BasicJsonType::value_type >::value &&!is_compatible_string_type< BasicJsonType, ConstructibleArrayType >::value &&is_default_constructible< ConstructibleArrayType >::value &&(std::is_move_assignable< ConstructibleArrayType >::value||std::is_copy_assignable< ConstructibleArrayType >::value)&&is_detected< iterator_t, ConstructibleArrayType >::value &&is_iterator_traits< iterator_traits< detected_t< iterator_t, ConstructibleArrayType > > >::value &&is_detected< range_value_t, ConstructibleArrayType >::value &&!std::is_same< ConstructibleArrayType, detected_t< range_value_t, ConstructibleArrayType > >::value &&is_complete_type< detected_t< range_value_t, ConstructibleArrayType > >::value > >**

Definition at line 4030 of file json.hpp.

## 9.87.2 Member Typedef Documentation

### 9.87.2.1 value_type

```
template<typename BasicJsonType, typename ConstructibleArrayType>
using detail::is_constructible_array_type_impl< BasicJsonType, ConstructibleArrayType, enable↩
_if_t< !std::is_same< ConstructibleArrayType, typename BasicJsonType::value_type >::value
&&!is_compatible_string_type< BasicJsonType, ConstructibleArrayType >::value &&is_default_constructible<
ConstructibleArrayType >::value &&(std::is_move_assignable< ConstructibleArrayType >::↩
value||std::is_copy_assignable< ConstructibleArrayType >::value)&&is_detected< iterator_t,
ConstructibleArrayType >::value &&is_iterator_traits< iterator_traits< detected_t< iterator↩
_t, ConstructibleArrayType > > >::value &&is_detected< range_value_t, ConstructibleArrayType
>::value &&!std::is_same< ConstructibleArrayType, detected_t< range_value_t, Constructible↩
ArrayType > >::value &&is_complete_type< detected_t< range_value_t, ConstructibleArrayType >
>::value > >::value_type = range_value_t<ConstructibleArrayType>
```

Definition at line 4047 of file json.hpp.

### 9.87.3 Member Data Documentation

#### 9.87.3.1 value

```
template<typename BasicJsonType, typename ConstructibleArrayType>
bool detail::is_constructible_array_type_impl< BasicJsonType, ConstructibleArrayType, enable↩
_if_t< !std::is_same< ConstructibleArrayType, typename BasicJsonType::value_type >::value
&&!is_compatible_string_type< BasicJsonType, ConstructibleArrayType >::value &&is_default_constructible<
ConstructibleArrayType >::value &&(std::is_move_assignable< ConstructibleArrayType >::↩
value||std::is_copy_assignable< ConstructibleArrayType >::value)&&is_detected< iterator_t,
ConstructibleArrayType >::value &&is_iterator_traits< iterator_traits< detected_t< iterator↩
_t, ConstructibleArrayType > > >::value &&is_detected< range_value_t, ConstructibleArrayType
>::value &&!std::is_same< ConstructibleArrayType, detected_t< range_value_t, Constructible↩
ArrayType > >::value &&is_complete_type< detected_t< range_value_t, ConstructibleArrayType >
>::value > >::value  [static], [constexpr]
```

**Initial value:**

```
                        =
std::is_same<value_type,
typename BasicJsonType::array_t::value_type>::value ||
has_from_json<BasicJsonType,
value_type>::value ||
has_non_default_from_json <
BasicJsonType,
value_type >::value
```

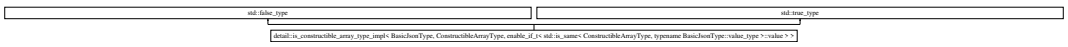
Definition at line 4049 of file json.hpp.

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

- include/External/json.hpp

## 9.88 detail::is_constructible_array_type_impl< BasicJsonType, ConstructibleArrayType, enable_if_t< std::is_same< ConstructibleArrayType, typename BasicJsonType::value_type >::value > > Struct Template Reference

Inheritance diagram for detail::is_constructible_array_type_impl< BasicJsonType, ConstructibleArrayType, enable_if_t< std::is_same< ConstructibleArrayType, typename BasicJsonType::value_type >::value > >:




### 9.88.1 Detailed Description

**template<typename BasicJsonType, typename ConstructibleArrayType>**
**struct detail::is_constructible_array_type_impl< BasicJsonType, ConstructibleArrayType, enable_if_t< std::is_same< ConstructibleArrayType, typename BasicJsonType::value_type >::value > >**

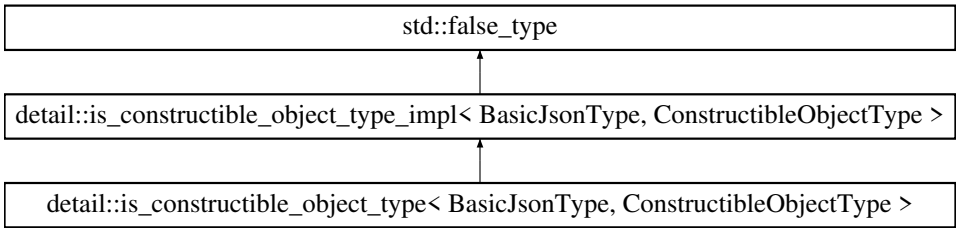
Definition at line 4023 of file json.hpp.

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

- include/External/json.hpp

# 9.89 detail::is_constructible_object_type< BasicJsonType, ConstructibleObjectType > Struct Template Reference

Inheritance diagram for detail::is_constructible_object_type< BasicJsonType, ConstructibleObjectType >:




## 9.89.1 Detailed Description

**template<typename BasicJsonType, typename ConstructibleObjectType>**
**struct detail::is_constructible_object_type< BasicJsonType, ConstructibleObjectType >**

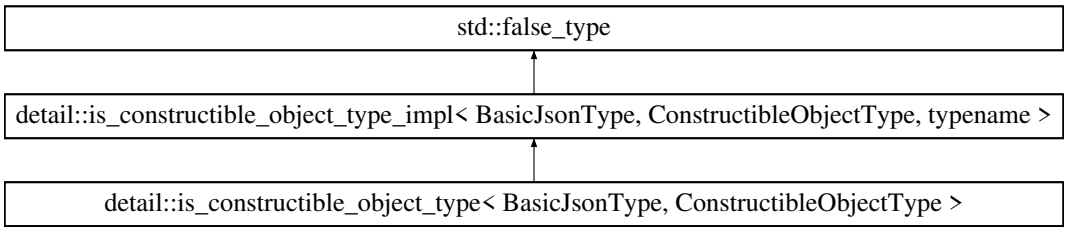
Definition at line 3969 of file json.hpp.

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

- include/External/json.hpp

# 9.90 detail::is_constructible_object_type_impl< BasicJsonType, ConstructibleObjectType, typename > Struct Template Reference

Inheritance diagram for detail::is_constructible_object_type_impl< BasicJsonType, ConstructibleObjectType, typename >:




## 9.90.1 Detailed Description

**template<typename BasicJsonType, typename ConstructibleObjectType, typename = void>**
**struct detail::is_constructible_object_type_impl< BasicJsonType, ConstructibleObjectType, typename >**

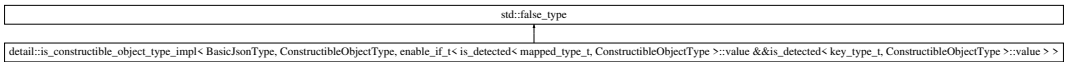
Definition at line 3942 of file json.hpp.

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

- include/External/json.hpp

# 9.91 detail::is_constructible_object_type_impl< BasicJsonType, ConstructibleObjectType, enable_if_t< is_detected< mapped_type_t, ConstructibleObjectType >::value &&is_detected< key_type_t, ConstructibleObjectType >::value > > Struct Template Reference

Inheritance diagram for detail::is_constructible_object_type_impl< BasicJsonType, ConstructibleObjectType, enable_if_t< is_detected< mapped_type_t, ConstructibleObjectType >::value &&is_detected< key_type_t, ConstructibleObjectType >::value > >:




### Public Types

- using object_t = typename BasicJsonType::object_t

### Static Public Attributes

- static constexpr bool value

## 9.91.1 Detailed Description

**template<typename BasicJsonType, typename ConstructibleObjectType>**
**struct detail::is_constructible_object_type_impl< BasicJsonType, ConstructibleObjectType, enable_if_t< is_detected< mapped_type_t, ConstructibleObjectType >::value &&is_detected< key_type_t, ConstructibleObjectType >::value > >**

Definition at line 3945 of file json.hpp.

## 9.91.2 Member Typedef Documentation

### 9.91.2.1 object_t

```
template<typename BasicJsonType, typename ConstructibleObjectType>
using detail::is_constructible_object_type_impl< BasicJsonType, ConstructibleObjectType, enable_if_t< is_detected< mapped_type_t, ConstructibleObjectType >::value &&is_detected< key_type_t, ConstructibleObjectType >::value > >::object_t = typename BasicJsonType::object_t
```

Definition at line 3950 of file json.hpp.

### 9.91.3 Member Data Documentation

#### 9.91.3.1 value

```
template<typename BasicJsonType, typename ConstructibleObjectType>
bool detail::is_constructible_object_type_impl< BasicJsonType, ConstructibleObjectType, enable↩
_if_t< is_detected< mapped_type_t, ConstructibleObjectType >::value &&is_detected< key_type↩
_t, ConstructibleObjectType >::value > >::value  [static], [constexpr]
```

**Initial value:**

```
                        =
(is_default_constructible<ConstructibleObjectType>::value &&
 (std::is_move_assignable<ConstructibleObjectType>::value ||
  std::is_copy_assignable<ConstructibleObjectType>::value) &&
 (is_constructible<typename ConstructibleObjectType::key_type,
  typename object_t::key_type>::value &&
  std::is_same <
  typename object_t::mapped_type,
  typename ConstructibleObjectType::mapped_type >::value)) ||
(has_from_json<BasicJsonType,
 typename ConstructibleObjectType::mapped_type>::value ||
 has_non_default_from_json <
 BasicJsonType,
 typename ConstructibleObjectType::mapped_type >::value)
```

Definition at line 3952 of file json.hpp.

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

- include/External/json.hpp

## 9.92 detail::is_constructible_string_type< BasicJsonType, ConstructibleStringType > Struct Template Reference

**Public Types**

- using laundered_type = ConstructibleStringType

**Static Public Attributes**

- static constexpr auto value

### 9.92.1 Detailed Description

**template<typename BasicJsonType, typename ConstructibleStringType>**
**struct detail::is_constructible_string_type< BasicJsonType, ConstructibleStringType >**

Definition at line 3981 of file json.hpp.

### 9.92.2 Member Typedef Documentation

#### 9.92.2.1 laundered_type

```
template<typename BasicJsonType, typename ConstructibleStringType>
using detail::is_constructible_string_type< BasicJsonType, ConstructibleStringType >::laundered_type = ConstructibleStringType
```

Definition at line 3987 of file json.hpp.

### 9.92.3 Member Data Documentation

#### 9.92.3.1 value

```
template<typename BasicJsonType, typename ConstructibleStringType>
auto detail::is_constructible_string_type< BasicJsonType, ConstructibleStringType >::value
[static], [constexpr]
```

**Initial value:**

```
                        =
conjunction <
is_constructible<laundered_type, typename BasicJsonType::string_t>,
is_detected_exact<typename BasicJsonType::string_t::value_type,
value_type_t, laundered_type »::value
```

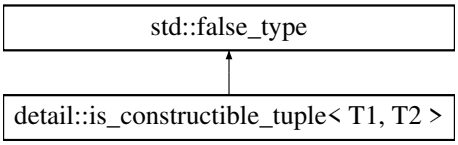
Definition at line 3990 of file json.hpp.

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

- include/External/json.hpp

## 9.93 detail::is_constructible_tuple< T1, T2 > Struct Template Reference

Inheritance diagram for detail::is_constructible_tuple< T1, T2 >:




### 9.93.1 Detailed Description

**template<typename T1, typename T2>**
**struct detail::is_constructible_tuple< T1, T2 >**

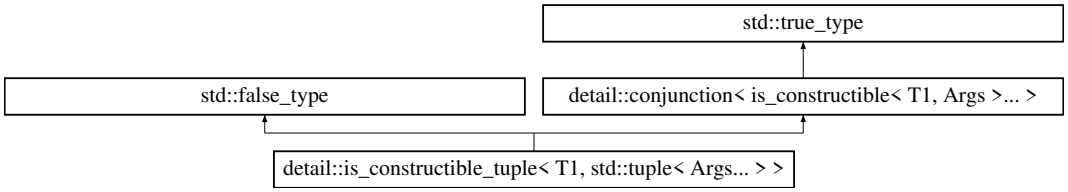
Definition at line 4107 of file json.hpp.

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

- include/External/json.hpp

## 9.94 detail::is_constructible_tuple< T1, std::tuple< Args... > > Struct Template Reference

Inheritance diagram for detail::is_constructible_tuple< T1, std::tuple< Args... > >:




### 9.94.1 Detailed Description

**template<typename T1, typename... Args>**
**struct detail::is_constructible_tuple< T1, std::tuple< Args... > >**

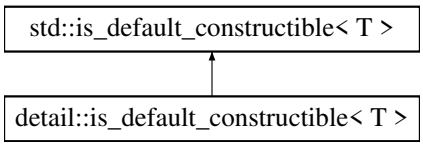
Definition at line 4110 of file json.hpp.

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

- include/External/json.hpp

## 9.95 detail::is_default_constructible< T > Struct Template Reference

Inheritance diagram for detail::is_default_constructible< T >:




### 9.95.1 Detailed Description

**template<typename T>**
**struct detail::is_default_constructible< T >**

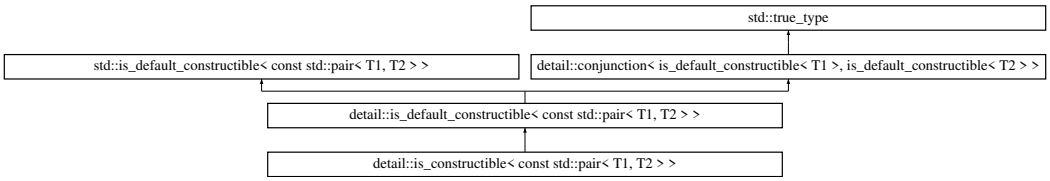
Definition at line 3830 of file json.hpp.

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

- include/External/json.hpp

## 9.96 detail::is_default_constructible< const std::pair< T1, T2 > > Struct Template Reference

Inheritance diagram for detail::is_default_constructible< const std::pair< T1, T2 > >:




### 9.96.1 Detailed Description

**template<typename T1, typename T2>**
**struct detail::is_default_constructible< const std::pair< T1, T2 > >**

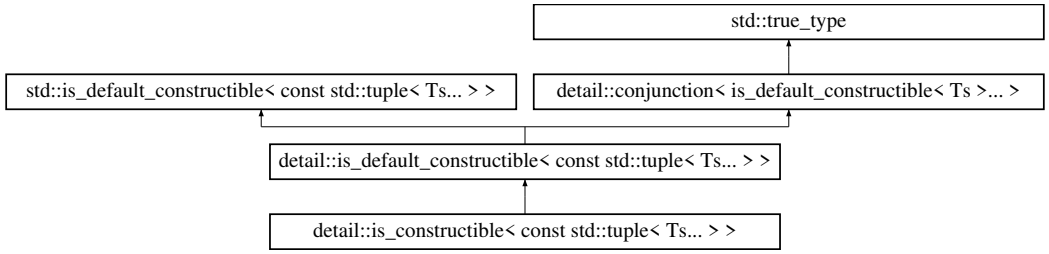
Definition at line 3837 of file json.hpp.

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

- include/External/json.hpp

## 9.97 detail::is_default_constructible< const std::tuple< Ts... > > Struct Template Reference

Inheritance diagram for detail::is_default_constructible< const std::tuple< Ts... > >:




### 9.97.1 Detailed Description

**template<typename... Ts>**
**struct detail::is_default_constructible< const std::tuple< Ts... > >**

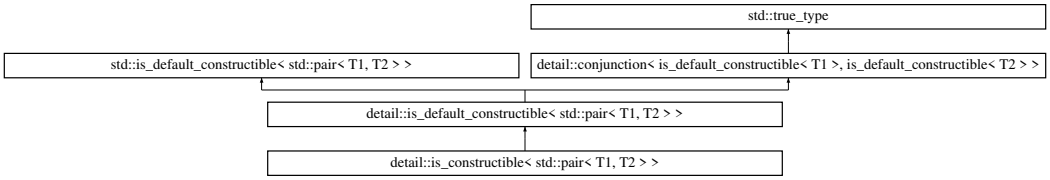
Definition at line 3845 of file json.hpp.

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

- include/External/json.hpp

# 9.98 detail::is_default_constructible< std::pair< T1, T2 > > Struct Template Reference

Inheritance diagram for detail::is_default_constructible< std::pair< T1, T2 > >:




## 9.98.1 Detailed Description

**template<typename T1, typename T2>**
**struct detail::is_default_constructible< std::pair< T1, T2 > >**

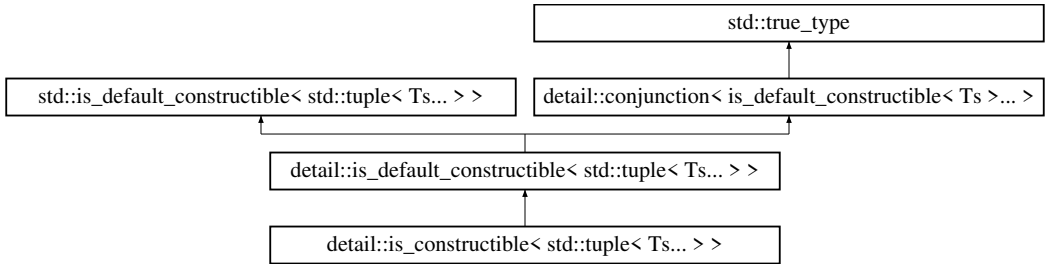
Definition at line 3833 of file json.hpp.

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

- include/External/json.hpp

# 9.99 detail::is_default_constructible< std::tuple< Ts... > > Struct Template Reference

Inheritance diagram for detail::is_default_constructible< std::tuple< Ts... > >:




## 9.99.1 Detailed Description

**template<typename... Ts>**
**struct detail::is_default_constructible< std::tuple< Ts... > >**

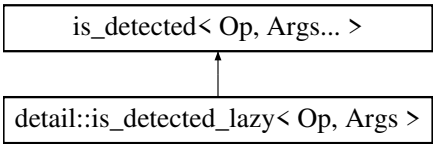
Definition at line 3841 of file json.hpp.

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

- include/External/json.hpp

## 9.100 detail::is_detected_lazy< Op, Args > Struct Template Reference

Inheritance diagram for detail::is_detected_lazy< Op, Args >:




### 9.100.1 Detailed Description

**template<template< class... > class Op, class... Args>**
**struct detail::is_detected_lazy< Op, Args >**

Definition at line 308 of file json.hpp.

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

- include/External/json.hpp

## 9.101 detail::is_getable< BasicJsonType, T > Struct Template Reference

**Static Public Attributes**

- static constexpr bool value = is_detected<get_template_function, const BasicJsonType&, T>::value

### 9.101.1 Detailed Description

**template<typename BasicJsonType, typename T>**
**struct detail::is_getable< BasicJsonType, T >**

Definition at line 3667 of file json.hpp.

### 9.101.2 Member Data Documentation

#### 9.101.2.1 value

```
template<typename BasicJsonType, typename T>
bool detail::is_getable< BasicJsonType, T >::value = is_detected<get_template_function, const
BasicJsonType&, T>::value  [static], [constexpr]
```

Definition at line 3669 of file json.hpp.

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

- include/External/json.hpp

# 9.102 detail::is_iterator_of_multibyte< T > Struct Template Reference

**Public Types**

- enum
- using value_type = typename std::iterator_traits<T>::value_type

## 9.102.1 Detailed Description

**template<typename T>**
**struct detail::is_iterator_of_multibyte< T >**

Definition at line 6915 of file json.hpp.

## 9.102.2 Member Typedef Documentation

### 9.102.2.1 value_type

```
template<typename T>
using detail::is_iterator_of_multibyte< T >::value_type = typename std::iterator_traits<T>::↩
value_type
```

Definition at line 6917 of file json.hpp.

## 9.102.3 Member Enumeration Documentation

### 9.102.3.1 anonymous enum

```
template<typename T>
anonymous enum
```

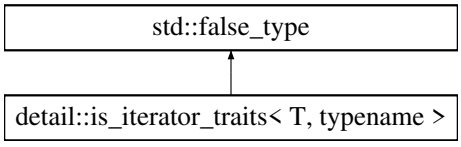
Definition at line 6918 of file json.hpp.

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

- include/External/json.hpp

# 9.103 detail::is_iterator_traits< T, typename > Struct Template Reference

Inheritance diagram for detail::is_iterator_traits< T, typename >:



### 9.103.1 Detailed Description

**template<typename T, typename = void>**
**struct detail::is_iterator_traits< T, typename >**

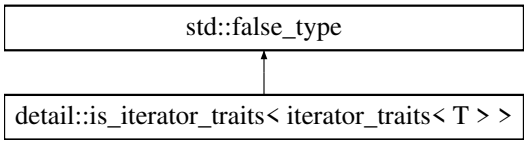
Definition at line 3864 of file json.hpp.

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

- include/External/json.hpp

## 9.104 detail::is_iterator_traits< iterator_traits< T > > Struct Template Reference

Inheritance diagram for detail::is_iterator_traits< iterator_traits< T > >:




#### Static Public Attributes

- static constexpr auto value

### 9.104.1 Detailed Description

**template<typename T>**
**struct detail::is_iterator_traits< iterator_traits< T > >**

Definition at line 3867 of file json.hpp.

### 9.104.2 Member Data Documentation

#### 9.104.2.1 value

```
template<typename T>
auto detail::is_iterator_traits< iterator_traits< T > >::value  [static], [constexpr]
```

**Initial value:**

```
                        =
is_detected<value_type_t, traits>::value &&
is_detected<difference_type_t, traits>::value &&
is_detected<pointer_t, traits>::value &&
is_detected<iterator_category_t, traits>::value &&
is_detected<reference_t, traits>::value
```

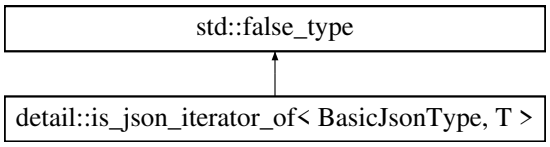
Definition at line 3873 of file json.hpp.

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

- include/External/json.hpp

## 9.105 detail::is_json_iterator_of< BasicJsonType, T > Struct Template Reference

Inheritance diagram for detail::is_json_iterator_of< BasicJsonType, T >:




### 9.105.1 Detailed Description

**template<typename BasicJsonType, typename T>**
**struct detail::is_json_iterator_of< BasicJsonType, T >**

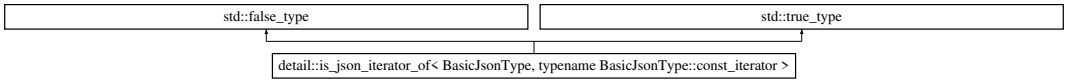
Definition at line 4113 of file json.hpp.

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

- include/External/json.hpp

## 9.106 detail::is_json_iterator_of< BasicJsonType, typename BasicJsonType::const_iterator > Struct Template Reference

Inheritance diagram for detail::is_json_iterator_of< BasicJsonType, typename BasicJsonType::const_iterator >:




### 9.106.1 Detailed Description

**template<typename BasicJsonType>**
**struct detail::is_json_iterator_of< BasicJsonType, typename BasicJsonType::const_iterator >**

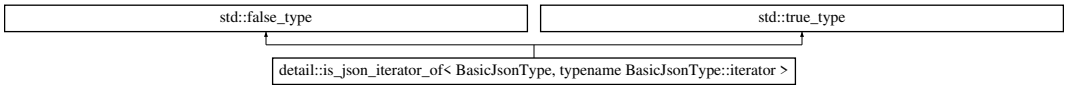
Definition at line 4119 of file json.hpp.

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

- include/External/json.hpp

## 9.107 detail::is_json_iterator_of< BasicJsonType, typename BasicJsonType::iterator > Struct Template Reference

Inheritance diagram for detail::is_json_iterator_of< BasicJsonType, typename BasicJsonType::iterator >:




### 9.107.1 Detailed Description

**template<typename BasicJsonType>**
**struct detail::is_json_iterator_of< BasicJsonType, typename BasicJsonType::iterator >**

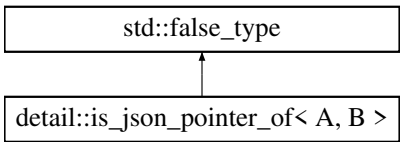
Definition at line 4116 of file json.hpp.

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

- include/External/json.hpp

## 9.108 detail::is_json_pointer_of< A, B > Struct Template Reference

Inheritance diagram for detail::is_json_pointer_of< A, B >:




### 9.108.1 Detailed Description

**template<typename A, typename B>**
**struct detail::is_json_pointer_of< A, B >**

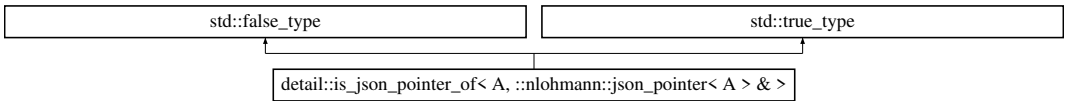
Definition at line 4134 of file json.hpp.

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

- include/External/json.hpp

# 9.109 detail::is_json_pointer_of< A, ::nlohmann::json_pointer< A > & > Struct Template Reference

Inheritance diagram for detail::is_json_pointer_of< A, ::nlohmann::json_pointer< A > & >:




## 9.109.1 Detailed Description

**template<typename A>**
**struct detail::is_json_pointer_of< A, ::nlohmann::json_pointer< A > & >**

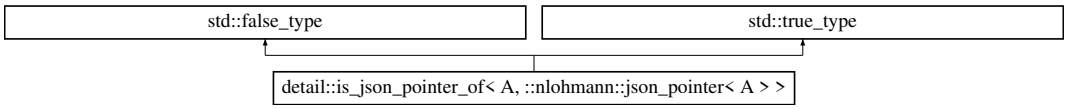
Definition at line 4140 of file json.hpp.

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

- include/External/json.hpp

# 9.110 detail::is_json_pointer_of< A, ::nlohmann::json_pointer< A > > Struct Template Reference

Inheritance diagram for detail::is_json_pointer_of< A, ::nlohmann::json_pointer< A > >:




## 9.110.1 Detailed Description

**template<typename A>**
**struct detail::is_json_pointer_of< A, ::nlohmann::json_pointer< A > >**

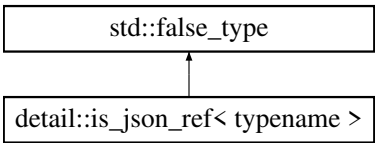
Definition at line 4137 of file json.hpp.

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

- include/External/json.hpp

## 9.111 detail::is_json_ref< typename > Struct Template Reference

Inheritance diagram for detail::is_json_ref< typename >:




### 9.111.1 Detailed Description

**template<typename>**
**struct detail::is_json_ref< typename >**

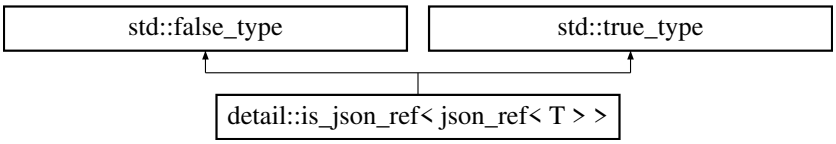
Definition at line 3619 of file json.hpp.

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

- include/External/json.hpp

## 9.112 detail::is_json_ref< json_ref< T > > Struct Template Reference

Inheritance diagram for detail::is_json_ref< json_ref< T > >:




### 9.112.1 Detailed Description

**template<typename T>**
**struct detail::is_json_ref< json_ref< T > >**

Definition at line 3622 of file json.hpp.

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

- include/External/json.hpp

## 9.113 detail::is_ordered_map< T > Struct Template Reference

**Classes**

- struct two

**Public Types**

- enum { **value** = sizeof(test<T>(nullptr)) == sizeof(char) }
- using one = char

**Static Public Member Functions**

- template<typename C>
  static one **test** (decltype(&C::capacity))
- template<typename C>
  static two **test** (...)

### 9.113.1 Detailed Description

**template<typename T>**
**struct detail::is_ordered_map< T >**

Definition at line 4208 of file json.hpp.

### 9.113.2 Member Typedef Documentation

#### 9.113.2.1 one

```
template<typename T>
using detail::is_ordered_map< T >::one = char
```

Definition at line 4210 of file json.hpp.

### 9.113.3 Member Enumeration Documentation

#### 9.113.3.1 anonymous enum

```
template<typename T>
anonymous enum
```

Definition at line 4220 of file json.hpp.

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

- include/External/json.hpp

## 9.114 detail::is_range< T > Struct Template Reference

**Static Public Attributes**

- static constexpr bool value = !std::is_same<iterator, nonesuch>::value && !std::is_same<sentinel, nonesuch>::value && is_iterator_begin

### 9.114.1 Detailed Description

**template<typename T>**
**struct detail::is_range< T >**

Definition at line 3882 of file json.hpp.

### 9.114.2 Member Data Documentation

#### 9.114.2.1 value

```
template<typename T>
bool detail::is_range< T >::value = !std::is_same<iterator, nonesuch>::value && !std::is_↩
same<sentinel, nonesuch>::value && is_iterator_begin  [static], [constexpr]
```

Definition at line 3897 of file json.hpp.

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

- include/External/json.hpp

## 9.115 detail::is_sax< SAX, BasicJsonType > Struct Template Reference

**Static Public Attributes**

- static constexpr bool value

### 9.115.1 Detailed Description

**template<typename SAX, typename BasicJsonType>**
**struct detail::is_sax< SAX, BasicJsonType >**

Definition at line 9785 of file json.hpp.

### 9.115.2 Member Data Documentation

#### 9.115.2.1 value

```
template<typename SAX, typename BasicJsonType>
bool detail::is_sax< SAX, BasicJsonType >::value  [static], [constexpr]
```

**Initial value:**

```
                    =
is_detected_exact<bool, null_function_t, SAX>::value &&
is_detected_exact<bool, boolean_function_t, SAX>::value &&
is_detected_exact<bool, number_integer_function_t, SAX, number_integer_t>::value &&
is_detected_exact<bool, number_unsigned_function_t, SAX, number_unsigned_t>::value &&
is_detected_exact<bool, number_float_function_t, SAX, number_float_t, string_t>::value &&
is_detected_exact<bool, string_function_t, SAX, string_t>::value &&
is_detected_exact<bool, binary_function_t, SAX, binary_t>::value &&
is_detected_exact<bool, start_object_function_t, SAX>::value &&
is_detected_exact<bool, key_function_t, SAX, string_t>::value &&
is_detected_exact<bool, end_object_function_t, SAX>::value &&
is_detected_exact<bool, start_array_function_t, SAX>::value &&
is_detected_exact<bool, end_array_function_t, SAX>::value &&
is_detected_exact<bool, parse_error_function_t, SAX, exception_t>::value
```

Definition at line 9799 of file json.hpp.

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

- include/External/json.hpp

# 9.116 detail::is_sax_static_asserts< SAX, BasicJsonType > Struct Template Reference

## 9.116.1 Detailed Description

**template<typename SAX, typename BasicJsonType>**
**struct detail::is_sax_static_asserts< SAX, BasicJsonType >**

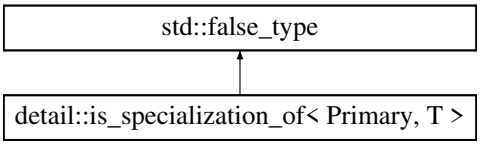
Definition at line 9816 of file json.hpp.

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

- include/External/json.hpp

# 9.117 detail::is_specialization_of< Primary, T > Struct Template Reference

Inheritance diagram for detail::is_specialization_of< Primary, T >:




## 9.117.1 Detailed Description

**template<template< typename... > class Primary, typename T>**
**struct detail::is_specialization_of< Primary, T >**

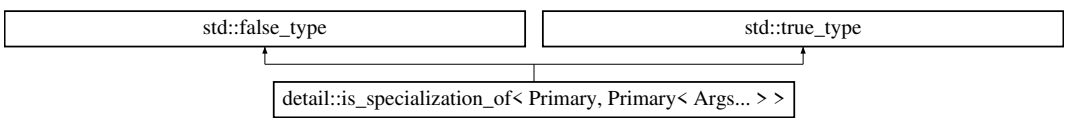
Definition at line 4124 of file json.hpp.

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

- include/External/json.hpp

# 9.118 detail::is_specialization_of< Primary, Primary< Args... > > Struct Template Reference

Inheritance diagram for detail::is_specialization_of< Primary, Primary< Args... > >:



### 9.118.1 Detailed Description

**template<template< typename... > class Primary, typename... Args>**
**struct detail::is_specialization_of< Primary, Primary< Args... > >**

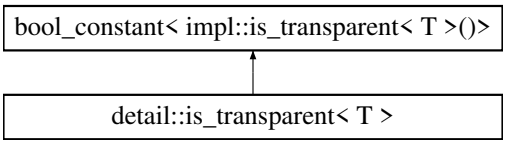
Definition at line 4127 of file json.hpp.

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

- include/External/json.hpp

## 9.119 detail::is_transparent< T > Struct Template Reference

Inheritance diagram for detail::is_transparent< T >:




### 9.119.1 Detailed Description

**template<typename T>**
**struct detail::is_transparent< T >**

Definition at line 4378 of file json.hpp.

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

- include/External/json.hpp

## 9.120 detail::iter_impl< BasicJsonType > Class Template Reference

#### Public Types

- using value_type = typename BasicJsonType::value_type
- using difference_type = typename BasicJsonType::difference_type

### Public Member Functions

- ∗defines a pointer to the type iterated **over** (value_type) using pointer
- ∗defines a reference to the type iterated **over** (value_type) using reference
- **iter_impl** (iter_impl &&) noexcept=default
- iter_impl & **operator=** (iter_impl &&) noexcept=default
- iter_impl (pointer object) noexcept

  *constructor for a given JSON instance*
- iter_impl (const iter_impl< const BasicJsonType > &other) noexcept

  *const copy constructor*
- iter_impl & operator= (const iter_impl< const BasicJsonType > &other) noexcept

  *converting assignment*
- 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*
- switch (m_object->m_data.m_type)
- void set_end () noexcept

  *set the iterator past the last value*
- 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)*
- template<typename IterImpl, detail::enable_if_t<(std::is_same< IterImpl, iter_impl >::value||std::is_same< IterImpl, other_iter_impl >::value), std::nullptr_t > = nullptr>
  bool operator== (const IterImpl &other) const

  *comparison: equal*
- template<typename IterImpl, detail::enable_if_t<(std::is_same< IterImpl, iter_impl >::value||std::is_same< IterImpl, other_iter_impl >::value), std::nullptr_t > = nullptr>
  bool operator!= (const IterImpl &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*
- const object_t::key_type & key () const
  *return the key of an object iterator*
- reference value () const
  *return the value of an iterator*

#### Public Attributes

- JSON_PRIVATE_UNLESS_TESTED : void set_begin() noexcept { JSON_ASSERT(m_object != nullptr)
- JSON_PRIVATE_UNLESS_TESTED : ∗ associated JSON instance pointer m_object = nullptr
- ∗the actual iterator of the associated instance internal_iterator< typename std::remove_const< BasicJson↩ Type >::type > m_it {}

#### Friends

- iter_impl operator+ (difference_type i, const iter_impl &it)
  *addition of distance and iterator*

### 9.120.1 Detailed Description

**template<typename BasicJsonType>**
**class detail::iter_impl< BasicJsonType >**

Definition at line 13707 of file json.hpp.

### 9.120.2 Member Typedef Documentation

#### 9.120.2.1 difference_type

```
template<typename BasicJsonType>
using detail::iter_impl< BasicJsonType >::difference_type = typename BasicJsonType::difference↩
_type
```

Definition at line 13738 of file json.hpp.

#### 9.120.2.2 value_type

```
template<typename BasicJsonType>
using detail::iter_impl< BasicJsonType >::value_type = typename BasicJsonType::value_type
```

Definition at line 13736 of file json.hpp.

## 9.120.3 Constructor & Destructor Documentation

### 9.120.3.1 iter_impl() [1/3]

```
template<typename BasicJsonType>
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`.

Definition at line 13760 of file json.hpp.

### 9.120.3.2 iter_impl() [2/3]

```
template<typename BasicJsonType>
detail::iter_impl< BasicJsonType >::iter_impl (
            const iter_impl< const BasicJsonType > & other)  [inline], [noexcept]
```

const copy 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* | const iterator to copy from |

**Note**

This copy constructor had to be defined explicitly to circumvent a bug occurring on msvc v19.0 compiler (VS 2015) debug build. For more information refer to: https://github.com/nlohmann/json/issues/1608

Definition at line 13810 of file json.hpp.

#### 9.120.3.3 iter_impl() [3/3]

```
template<typename BasicJsonType>
detail::iter_impl< BasicJsonType >::iter_impl (
            const iter_impl< typename std::remove_const< BasicJsonType >::type > & other)
[inline], [noexcept]
```

converting constructor

**Parameters**

| | | |
|---|---|---|
| in | *other* | non-const iterator to copy from |

**Note**

It is not checked whether *other* is initialized.

Definition at line 13835 of file json.hpp.

### 9.120.4 Member Function Documentation

#### 9.120.4.1 key()

```
template<typename BasicJsonType>
const object_t::key_type & detail::iter_impl< BasicJsonType >::key () const  [inline]
```

return the key of an object iterator

**Precondition**

The iterator is initialized; i.e. `m_object != nullptr`.

Definition at line 14391 of file json.hpp.

#### 9.120.4.2 operator"!=()

```
template<typename BasicJsonType>
template<typename IterImpl, detail::enable_if_t<(std::is_same< IterImpl, iter_impl >::value||std::↩
is_same< IterImpl, other_iter_impl >::value), std::nullptr_t > = nullptr>
bool detail::iter_impl< BasicJsonType >::operator!= (
            const IterImpl & other) const  [inline]
```

comparison: not equal

**Precondition**

(1) Both iterators are initialized to point to the same object, or (2) both iterators are value-initialized.

Definition at line 14169 of file json.hpp.

#### 9.120.4.3 operator*()

```
template<typename BasicJsonType>
reference 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`.

Definition at line 13940 of file json.hpp.

#### 9.120.4.4 operator+()

```
template<typename BasicJsonType>
iter_impl detail::iter_impl< BasicJsonType >::operator+ (
            difference_type i) const  [inline]
```

add to iterator

**Precondition**

The iterator is initialized; i.e. `m_object != nullptr`.

Definition at line 14291 of file json.hpp.

#### 9.120.4.5 operator++() [1/2]

```
template<typename BasicJsonType>
iter_impl & detail::iter_impl< BasicJsonType >::operator++ ()  [inline]
```

pre-increment (++it)

**Precondition**

The iterator is initialized; i.e. `m_object != nullptr`.

Definition at line 14037 of file json.hpp.

#### 9.120.4.6 operator++() [2/2]

```
template<typename BasicJsonType>
iter_impl detail::iter_impl< BasicJsonType >::operator++ (
            int ) &  [inline]
```

post-increment (it++)

**Precondition**

The iterator is initialized; i.e. `m_object != nullptr`.

Definition at line 14026 of file json.hpp.

#### 9.120.4.7 operator+=()

```
template<typename BasicJsonType>
iter_impl & detail::iter_impl< BasicJsonType >::operator+= (
            difference_type i)  [inline]
```

add to iterator

**Precondition**

The iterator is initialized; i.e. `m_object != nullptr`.

Definition at line 14245 of file json.hpp.

#### 9.120.4.8 operator-() [1/2]

```
template<typename BasicJsonType>
difference_type detail::iter_impl< BasicJsonType >::operator- (
            const iter_impl< BasicJsonType > & other) const  [inline]
```

return difference

**Precondition**

The iterator is initialized; i.e. `m_object != nullptr`.

Definition at line 14324 of file json.hpp.

#### 9.120.4.9 operator-() [2/2]

```
template<typename BasicJsonType>
iter_impl detail::iter_impl< BasicJsonType >::operator- (
            difference_type i) const  [inline]
```

subtract from iterator

**Precondition**

The iterator is initialized; i.e. `m_object != nullptr`.

Definition at line 14313 of file json.hpp.

#### 9.120.4.10 operator--() [1/2]

```
template<typename BasicJsonType>
iter_impl & detail::iter_impl< BasicJsonType >::operator-- ()  [inline]
```

pre-decrement (–it)

**Precondition**

The iterator is initialized; i.e. `m_object != nullptr`.

Definition at line 14088 of file json.hpp.

#### 9.120.4.11 operator--() [2/2]

```
template<typename BasicJsonType>
iter_impl detail::iter_impl< BasicJsonType >::operator-- (
            int ) &  [inline]
```

post-decrement (it–)

**Precondition**

The iterator is initialized; i.e. `m_object != nullptr`.

Definition at line 14077 of file json.hpp.

#### 9.120.4.12 operator-=()

```
template<typename BasicJsonType>
iter_impl & detail::iter_impl< BasicJsonType >::operator-= (
            difference_type i)  [inline]
```

subtract from iterator

**Precondition**

The iterator is initialized; i.e. `m_object != nullptr`.

Definition at line 14282 of file json.hpp.

#### 9.120.4.13 operator->()

```
template<typename BasicJsonType>
pointer detail::iter_impl< BasicJsonType >::operator-> () const  [inline]
```

dereference the iterator

**Precondition**

The iterator is initialized; i.e. `m_object != nullptr`.

Definition at line 13984 of file json.hpp.

#### 9.120.4.14 operator<()

```
template<typename BasicJsonType>
bool detail::iter_impl< BasicJsonType >::operator< (
            const iter_impl< BasicJsonType > & other) const  [inline]
```

comparison: smaller

**Precondition**

(1) Both iterators are initialized to point to the same object, or (2) both iterators are value-initialized.

Definition at line 14178 of file json.hpp.

#### 9.120.4.15 operator<=()

```
template<typename BasicJsonType>
bool detail::iter_impl< BasicJsonType >::operator<= (
            const iter_impl< BasicJsonType > & other) const  [inline]
```

comparison: less than or equal

**Precondition**

(1) Both iterators are initialized to point to the same object, or (2) both iterators are value-initialized.

Definition at line 14218 of file json.hpp.

#### 9.120.4.16 operator=() [1/2]

```
template<typename BasicJsonType>
iter_impl & detail::iter_impl< BasicJsonType >::operator= (
            const iter_impl< const BasicJsonType > & other)  [inline], [noexcept]
```

converting assignment

**Parameters**

| | | |
|---|---|---|
| in | *other* | const iterator to copy from |

**Returns**

const/non-const iterator

**Note**

It is not checked whether *other* is initialized.

Definition at line 13820 of file json.hpp.

#### 9.120.4.17 operator=() [2/2]

```
template<typename BasicJsonType>
iter_impl & detail::iter_impl< BasicJsonType >::operator= (
            const iter_impl< typename std::remove_const< BasicJsonType >::type > & other)
[inline], [noexcept]
```

converting assignment

**Parameters**

| | | |
|---|---|---|
| in | *other* | non-const iterator to copy from |

**Returns**

const/non-const iterator

**Note**

It is not checked whether *other* is initialized.

Definition at line 13845 of file json.hpp.

#### 9.120.4.18 operator==()

```
template<typename BasicJsonType>
template<typename IterImpl, detail::enable_if_t<(std::is_same< IterImpl, iter_impl >::value||std::↩
is_same< IterImpl, other_iter_impl >::value), std::nullptr_t > = nullptr>
bool detail::iter_impl< BasicJsonType >::operator== (
            const IterImpl & other) const  [inline]
```

comparison: equal

**Precondition**

(1) Both iterators are initialized to point to the same object, or (2) both iterators are value-initialized.

Definition at line 14129 of file json.hpp.

#### 9.120.4.19 operator>()

```
template<typename BasicJsonType>
bool detail::iter_impl< BasicJsonType >::operator> (
            const iter_impl< BasicJsonType > & other) const  [inline]
```

comparison: greater than

**Precondition**

(1) Both iterators are initialized to point to the same object, or (2) both iterators are value-initialized.

Definition at line 14227 of file json.hpp.

#### 9.120.4.20 operator>=()

```
template<typename BasicJsonType>
bool detail::iter_impl< BasicJsonType >::operator>= (
            const iter_impl< BasicJsonType > & other) const  [inline]
```

comparison: greater than or equal

**Precondition**

(1) The iterator is initialized; i.e. `m_object != nullptr`, or (2) both iterators are value-initialized.

Definition at line 14236 of file json.hpp.

#### 9.120.4.21 operator[]()

```
template<typename BasicJsonType>
reference detail::iter_impl< BasicJsonType >::operator[] (
            difference_type n) const  [inline]
```

access to successor

**Precondition**

The iterator is initialized; i.e. `m_object != nullptr`.

Definition at line 14353 of file json.hpp.

#### 9.120.4.22 set_end()

```
template<typename BasicJsonType>
void detail::iter_impl< BasicJsonType >::set_end ()  [inline], [noexcept]
```

set the iterator past the last value

**Precondition**

The iterator is initialized; i.e. `m_object != nullptr`.

Definition at line 13901 of file json.hpp.

#### 9.120.4.23 switch()

```
template<typename BasicJsonType>
detail::iter_impl< BasicJsonType >::switch (
            m_object->m_data.  m_type)  [inline]
```

Definition at line 13861 of file json.hpp.

#### 9.120.4.24 value()

```
template<typename BasicJsonType>
reference detail::iter_impl< BasicJsonType >::value () const  [inline]
```

return the value of an iterator

**Precondition**

The iterator is initialized; i.e. `m_object != nullptr`.

Definition at line 14407 of file json.hpp.

### 9.120.5 Friends And Related Symbol Documentation

#### 9.120.5.1 operator+

```
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`.

Definition at line 14302 of file json.hpp.

### 9.120.6 Member Data Documentation

#### 9.120.6.1 __pad0__

```
template<typename BasicJsonType>
JSON_PRIVATE_UNLESS_TESTED detail::iter_impl< BasicJsonType >::__pad0__
```

Definition at line 13852 of file json.hpp.

#### 9.120.6.2 __pad1__

```
template<typename BasicJsonType>
JSON_PRIVATE_UNLESS_TESTED detail::iter_impl< BasicJsonType >::__pad1__
```

Definition at line 14412 of file json.hpp.

#### 9.120.6.3 m_it

```
template<typename BasicJsonType>
* the actual iterator of the associated instance internal_iterator<typename std::remove_↩
const<BasicJsonType>::type> detail::iter_impl< BasicJsonType >::m_it {}
```

Definition at line 14416 of file json.hpp.

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

- include/External/json.hpp

## 9.121 detail::iteration_proxy< IteratorType > Class Template Reference

proxy class for the items() function

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

#### Public Member Functions

- iteration_proxy (typename IteratorType::reference cont) noexcept
  *construct iteration proxy from a container*
- **iteration_proxy** (iteration_proxy const &)=default
- iteration_proxy & **operator=** (iteration_proxy const &)=default
- **iteration_proxy** (iteration_proxy &&) noexcept=default
- iteration_proxy & **operator=** (iteration_proxy &&) noexcept=default
- iteration_proxy_value< IteratorType > begin () const noexcept
  *return iterator begin (needed for range-based for)*
- iteration_proxy_value< IteratorType > end () const noexcept
  *return iterator end (needed for range-based for)*

### 9.121.1 Detailed Description

**template<typename IteratorType>**
**class detail::iteration_proxy< IteratorType >**

proxy class for the items() function

Definition at line 5641 of file json.hpp.

### 9.121.2 Constructor & Destructor Documentation

#### 9.121.2.1 iteration_proxy()

```
template<typename IteratorType>
detail::iteration_proxy< IteratorType >::iteration_proxy (
            typename IteratorType::reference cont)  [inline], [explicit], [noexcept]
```

construct iteration proxy from a container

Definition at line 5651 of file json.hpp.

### 9.121.3 Member Function Documentation

#### 9.121.3.1 begin()

```
template<typename IteratorType>
iteration_proxy_value< IteratorType > detail::iteration_proxy< IteratorType >::begin () const
[inline], [noexcept]
```

return iterator begin (needed for range-based for)

Definition at line 5661 of file json.hpp.

#### 9.121.3.2 end()

```
template<typename IteratorType>
iteration_proxy_value< IteratorType > detail::iteration_proxy< IteratorType >::end () const
[inline], [noexcept]
```

return iterator end (needed for range-based for)

Definition at line 5667 of file json.hpp.

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

- include/External/json.hpp

# 9.122 detail::iteration_proxy_value< IteratorType > Class Template Reference

### Public Types

- using difference_type = std::ptrdiff_t
- using value_type = iteration_proxy_value
- using pointer = value_type *
- using reference = value_type &
- using iterator_category = std::forward_iterator_tag
- using string_type = typename std::remove_cv< typename std::remove_reference<decltype( std::↩ declval<IteratorType>().key() ) >::type >::type

### Public Member Functions

- iteration_proxy_value (IteratorType it, std::size_t array_index_=0) noexcept(std::is_nothrow_move_↩ constructible< IteratorType >::value &&std::is_nothrow_default_constructible< string_type >::value)
- **iteration_proxy_value** (iteration_proxy_value const &)=default
- iteration_proxy_value & **operator=** (iteration_proxy_value const &)=default
- **iteration_proxy_value** (iteration_proxy_value &&) noexcept(std::is_nothrow_move_constructible< Iterator↩ Type >::value &&std::is_nothrow_move_constructible< string_type >::value)=default
- iteration_proxy_value & **operator=** (iteration_proxy_value &&) noexcept(std::is_nothrow_move_assignable< IteratorType >::value &&std::is_nothrow_move_assignable< string_type >::value)=default
- const iteration_proxy_value & operator* () const

  *dereference operator (needed for range-based for)*
- iteration_proxy_value & operator++ ()

  *increment operator (needed for range-based for)*
- iteration_proxy_value operator++ (int) &
- bool operator== (const iteration_proxy_value &o) const

  *equality operator (needed for InputIterator)*
- bool operator!= (const iteration_proxy_value &o) const

  *inequality operator (needed for range-based for)*
- const string_type & key () const

  *return key of the iterator*
- IteratorType::reference value () const

  *return value of the iterator*

## 9.122.1 Detailed Description

**template<typename IteratorType>**
**class detail::iteration_proxy_value< IteratorType >**

Definition at line 5520 of file json.hpp.

## 9.122.2 Member Typedef Documentation

### 9.122.2.1 difference_type

```
template<typename IteratorType>
using detail::iteration_proxy_value< IteratorType >::difference_type = std::ptrdiff_t
```

Definition at line 5523 of file json.hpp.

#### 9.122.2.2 iterator_category

```
template<typename IteratorType>
using detail::iteration_proxy_value< IteratorType >::iterator_category = std::forward_iterator↩
_tag
```

Definition at line 5527 of file json.hpp.

#### 9.122.2.3 pointer

```
template<typename IteratorType>
using detail::iteration_proxy_value< IteratorType >::pointer = value_type *
```

Definition at line 5525 of file json.hpp.

#### 9.122.2.4 reference

```
template<typename IteratorType>
using detail::iteration_proxy_value< IteratorType >::reference = value_type &
```

Definition at line 5526 of file json.hpp.

#### 9.122.2.5 string_type

```
template<typename IteratorType>
using detail::iteration_proxy_value< IteratorType >::string_type = typename std::remove_cv<
typename std::remove_reference<decltype( std::declval<IteratorType>().key() ) >::type >::↩
type
```

Definition at line 5528 of file json.hpp.

#### 9.122.2.6 value_type

```
template<typename IteratorType>
using detail::iteration_proxy_value< IteratorType >::value_type = iteration_proxy_value
```

Definition at line 5524 of file json.hpp.

### 9.122.3 Constructor & Destructor Documentation

#### 9.122.3.1 iteration_proxy_value()

```
template<typename IteratorType>
detail::iteration_proxy_value< IteratorType >::iteration_proxy_value (
            IteratorType it,
            std::size_t array_index_ = 0)  [inline], [explicit], [noexcept]
```

Definition at line 5544 of file json.hpp.

## 9.122.4 Member Function Documentation

### 9.122.4.1 key()

```
template<typename IteratorType>
const string_type & detail::iteration_proxy_value< IteratorType >::key () const  [inline]
```

return key of the iterator

Definition at line 5598 of file json.hpp.

### 9.122.4.2 operator"!=()

```
template<typename IteratorType>
bool detail::iteration_proxy_value< IteratorType >::operator!= (
            const iteration_proxy_value< IteratorType > & o) const  [inline]
```

inequality operator (needed for range-based for)

Definition at line 5592 of file json.hpp.

### 9.122.4.3 operator*()

```
template<typename IteratorType>
const iteration_proxy_value & detail::iteration_proxy_value< IteratorType >::operator* ()
const  [inline]
```

dereference operator (needed for range-based for)

Definition at line 5563 of file json.hpp.

### 9.122.4.4 operator++() [1/2]

```
template<typename IteratorType>
iteration_proxy_value & detail::iteration_proxy_value< IteratorType >::operator++ ()  [inline]
```

increment operator (needed for range-based for)

Definition at line 5569 of file json.hpp.

### 9.122.4.5 operator++() [2/2]

```
template<typename IteratorType>
iteration_proxy_value detail::iteration_proxy_value< IteratorType >::operator++ (
            int ) &  [inline]
```

Definition at line 5577 of file json.hpp.

#### 9.122.4.6 operator==()

```
template<typename IteratorType>
bool detail::iteration_proxy_value< IteratorType >::operator== (
            const iteration_proxy_value< IteratorType > & o) const  [inline]
```

equality operator (needed for InputIterator)

Definition at line 5586 of file json.hpp.

#### 9.122.4.7 value()

```
template<typename IteratorType>
IteratorType::reference detail::iteration_proxy_value< IteratorType >::value () const  [inline]
```

return value of the iterator

Definition at line 5634 of file json.hpp.

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

- include/External/json.hpp

## 9.123 detail::iterator_input_adapter< IteratorType > Class Template Reference

### Public Types

- using char_type = typename std::iterator_traits<IteratorType>::value_type

### Public Member Functions

- iterator_input_adapter (IteratorType first, IteratorType last)
- char_traits< char_type >::int_type get_character ()
- template<class T>
  std::size_t get_elements (T *dest, std::size_t count=1)

### Friends

- template<typename BaseInputAdapter, size_t T>
  struct wide_string_input_helper

### 9.123.1 Detailed Description

**template<typename IteratorType>**
**class detail::iterator_input_adapter< IteratorType >**

Definition at line 6672 of file json.hpp.

## 9.123.2 Member Typedef Documentation

### 9.123.2.1 char_type

```
template<typename IteratorType>
using detail::iterator_input_adapter< IteratorType >::char_type = typename std::iterator_↩
traits<IteratorType>::value_type
```

Definition at line 6675 of file json.hpp.

## 9.123.3 Constructor & Destructor Documentation

### 9.123.3.1 iterator_input_adapter()

```
template<typename IteratorType>
detail::iterator_input_adapter< IteratorType >::iterator_input_adapter (
            IteratorType first,
            IteratorType last)  [inline]
```

Definition at line 6677 of file json.hpp.

## 9.123.4 Member Function Documentation

### 9.123.4.1 get_character()

```
template<typename IteratorType>
char_traits< char_type >::int_type detail::iterator_input_adapter< IteratorType >::get_↩
character ()  [inline]
```

Definition at line 6681 of file json.hpp.

### 9.123.4.2 get_elements()

```
template<typename IteratorType>
template<class T>
std::size_t detail::iterator_input_adapter< IteratorType >::get_elements (
            T * dest,
            std::size_t count = 1)  [inline]
```

Definition at line 6695 of file json.hpp.

## 9.123.5 Friends And Related Symbol Documentation

### 9.123.5.1 wide_string_input_helper

```
template<typename IteratorType>
template<typename BaseInputAdapter, size_t T>
friend struct wide_string_input_helper  [friend]
```

Definition at line 6718 of file json.hpp.

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

- include/External/json.hpp

# 9.124 detail::iterator_input_adapter_factory< IteratorType, Enable > Struct Template Reference

### Public Types

- using iterator_type = IteratorType
- using char_type = typename std::iterator_traits<iterator_type>::value_type
- using adapter_type = iterator_input_adapter<iterator_type>

### Static Public Member Functions

- static adapter_type create (IteratorType first, IteratorType last)

## 9.124.1 Detailed Description

**template<typename IteratorType, typename Enable = void>**
**struct detail::iterator_input_adapter_factory< IteratorType, Enable >**

Definition at line 6902 of file json.hpp.

## 9.124.2 Member Typedef Documentation

### 9.124.2.1 adapter_type

```
template<typename IteratorType, typename Enable = void>
using detail::iterator_input_adapter_factory< IteratorType, Enable >::adapter_type = iterator_input_adapter<i
_type>
```

Definition at line 6906 of file json.hpp.

### 9.124.2.2 char_type

```
template<typename IteratorType, typename Enable = void>
using detail::iterator_input_adapter_factory< IteratorType, Enable >::char_type = typename
std::iterator_traits<iterator_type>::value_type
```

Definition at line 6905 of file json.hpp.

### 9.124.2.3 iterator_type

```
template<typename IteratorType, typename Enable = void>
using detail::iterator_input_adapter_factory< IteratorType, Enable >::iterator_type = Iterator↩
Type
```

Definition at line 6904 of file json.hpp.

### 9.124.3 Member Function Documentation

#### 9.124.3.1 create()

```
template<typename IteratorType, typename Enable = void>
adapter_type detail::iterator_input_adapter_factory< IteratorType, Enable >::create (
            IteratorType first,
            IteratorType last)  [inline], [static]
```

Definition at line 6908 of file json.hpp.

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

- include/External/json.hpp

## 9.125 detail::iterator_input_adapter_factory< IteratorType, enable_if_t< is_iterator_of_multibyte< IteratorType >::value > > Struct Template Reference

#### Public Types

- using iterator_type = IteratorType
- using char_type = typename std::iterator_traits<iterator_type>::value_type
- using base_adapter_type = iterator_input_adapter<iterator_type>
- using adapter_type = wide_string_input_adapter<base_adapter_type, char_type>

#### Static Public Member Functions

- static adapter_type create (IteratorType first, IteratorType last)

### 9.125.1 Detailed Description

**template<typename IteratorType>**
**struct detail::iterator_input_adapter_factory< IteratorType, enable_if_t< is_iterator_of_multibyte< IteratorType >::value > >**

Definition at line 6925 of file json.hpp.

### 9.125.2 Member Typedef Documentation

#### 9.125.2.1 adapter_type

```
template<typename IteratorType>
using detail::iterator_input_adapter_factory< IteratorType, enable_if_t< is_iterator_of_multibyte<
IteratorType >::value > >::adapter_type = wide_string_input_adapter<base_adapter_type, char↩
_type>
```

Definition at line 6930 of file json.hpp.

#### 9.125.2.2 base_adapter_type

```
template<typename IteratorType>
using detail::iterator_input_adapter_factory< IteratorType, enable_if_t< is_iterator_of_multibyte<
IteratorType >::value > >::base_adapter_type = iterator_input_adapter<iterator_type>
```

Definition at line 6929 of file json.hpp.

#### 9.125.2.3 char_type

```
template<typename IteratorType>
using detail::iterator_input_adapter_factory< IteratorType, enable_if_t< is_iterator_of_multibyte<
IteratorType >::value > >::char_type = typename std::iterator_traits<iterator_type>::value_↩
type
```

Definition at line 6928 of file json.hpp.

#### 9.125.2.4 iterator_type

```
template<typename IteratorType>
using detail::iterator_input_adapter_factory< IteratorType, enable_if_t< is_iterator_of_multibyte<
IteratorType >::value > >::iterator_type = IteratorType
```

Definition at line 6927 of file json.hpp.

### 9.125.3 Member Function Documentation

#### 9.125.3.1 create()

```
template<typename IteratorType>
adapter_type detail::iterator_input_adapter_factory< IteratorType, enable_if_t< is_iterator_of_multibyte<
IteratorType >::value > >::create (
            IteratorType first,
            IteratorType last)  [inline], [static]
```

Definition at line 6932 of file json.hpp.

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

- include/External/json.hpp

## 9.126 detail::iterator_traits< T, typename > Struct Template Reference

### 9.126.1 Detailed Description

**template<typename T, typename = void>**
**struct detail::iterator_traits< T, typename >**

Definition at line 3423 of file json.hpp.

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

- include/External/json.hpp

# 9.127 detail::iterator_traits< T *, enable_if_t< std::is_object< T >::value > > Struct Template Reference

**Public Types**

- using iterator_category = std::random_access_iterator_tag
- using value_type = T
- using difference_type = ptrdiff_t
- using pointer = T*
- using reference = T&

## 9.127.1 Detailed Description

**template<typename T>**
**struct detail::iterator_traits< T *, enable_if_t< std::is_object< T >::value > >**

Definition at line 3434 of file json.hpp.

## 9.127.2 Member Typedef Documentation

### 9.127.2.1 difference_type

```
template<typename T>
using detail::iterator_traits< T *, enable_if_t< std::is_object< T >::value > >::difference↩
_type = ptrdiff_t
```

Definition at line 3438 of file json.hpp.

### 9.127.2.2 iterator_category

```
template<typename T>
using detail::iterator_traits< T *, enable_if_t< std::is_object< T >::value > >::iterator_↩
category = std::random_access_iterator_tag
```

Definition at line 3436 of file json.hpp.

### 9.127.2.3 pointer

```
template<typename T>
using detail::iterator_traits< T *, enable_if_t< std::is_object< T >::value > >::pointer =
T*
```

Definition at line 3439 of file json.hpp.

#### 9.127.2.4 reference

```
template<typename T>
using detail::iterator_traits< T *, enable_if_t< std::is_object< T >::value > >::reference =
T&
```

Definition at line 3440 of file json.hpp.

#### 9.127.2.5 value_type

```
template<typename T>
using detail::iterator_traits< T *, enable_if_t< std::is_object< T >::value > >::value_type
= T
```

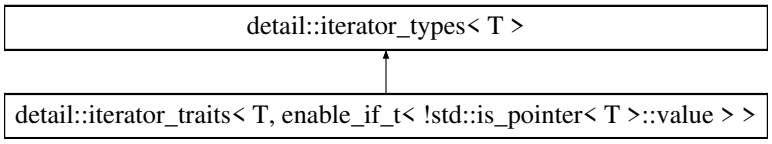
Definition at line 3437 of file json.hpp.

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

- include/External/json.hpp

## 9.128 detail::iterator_traits< T, enable_if_t< !std::is_pointer< T >::value > > Struct Template Reference

Inheritance diagram for detail::iterator_traits< T, enable_if_t< !std::is_pointer< T >::value > >:




### 9.128.1 Detailed Description

**template<typename T>**
**struct detail::iterator_traits< T, enable_if_t< !std::is_pointer< T >::value > >**

Definition at line 3428 of file json.hpp.

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

- include/External/json.hpp

# 9.129 detail::iterator_types< It, typename > Struct Template Reference

## 9.129.1 Detailed Description

**template<typename It, typename = void>**
**struct detail::iterator_types< It, typename >**

Definition at line 3405 of file json.hpp.

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

- include/External/json.hpp

# 9.130 detail::iterator_types< It, void_t< typename It::difference_type, typename It::value_type, typename It::pointer, typename It::reference, typename It::iterator_category > > Struct Template Reference

### Public Types

- using difference_type = typename It::difference_type
- using value_type = typename It::value_type
- using pointer = typename It::pointer
- using reference = typename It::reference
- using iterator_category = typename It::iterator_category

## 9.130.1 Detailed Description

**template<typename It>**
**struct detail::iterator_types< It, void_t< typename It::difference_type, typename It::value_type, typename It::pointer, typename It::reference, typename It::iterator_category > >**

Definition at line 3408 of file json.hpp.

## 9.130.2 Member Typedef Documentation

### 9.130.2.1 difference_type

```
template<typename It>
using detail::iterator_types< It, void_t< typename It::difference_type, typename It::value↩
_type, typename It::pointer, typename It::reference, typename It::iterator_category > >::↩
difference_type = typename It::difference_type
```

Definition at line 3413 of file json.hpp.

#### 9.130.2.2 iterator_category

```
template<typename It>
using detail::iterator_types< It, void_t< typename It::difference_type, typename It::value↩
_type, typename It::pointer, typename It::reference, typename It::iterator_category > >::↩
iterator_category = typename It::iterator_category
```

Definition at line 3417 of file json.hpp.

#### 9.130.2.3 pointer

```
template<typename It>
using detail::iterator_types< It, void_t< typename It::difference_type, typename It::value↩
_type, typename It::pointer, typename It::reference, typename It::iterator_category > >::↩
pointer = typename It::pointer
```

Definition at line 3415 of file json.hpp.

#### 9.130.2.4 reference

```
template<typename It>
using detail::iterator_types< It, void_t< typename It::difference_type, typename It::value↩
_type, typename It::pointer, typename It::reference, typename It::iterator_category > >::↩
reference = typename It::reference
```

Definition at line 3416 of file json.hpp.

#### 9.130.2.5 value_type

```
template<typename It>
using detail::iterator_types< It, void_t< typename It::difference_type, typename It::value↩
_type, typename It::pointer, typename It::reference, typename It::iterator_category > >::↩
value_type = typename It::value_type
```

Definition at line 3414 of file json.hpp.

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

- include/External/json.hpp

## 9.131 detail::json_default_base Struct Reference

Default base class of the basic_json class.

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

### 9.131.1 Detailed Description

Default base class of the basic_json class.

So that the correct implementations of the copy / move ctors / assign operators of basic_json do not require complex case distinctions (no base class / custom base class used as customization point), basic_json always has a base class. By default, this class is used because it is empty and thus has no effect on the behavior of basic_json.

Definition at line 14589 of file json.hpp.

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

- include/External/json.hpp

## 9.132 json_pointer< RefStringType > Class Template Reference

JSON Pointer defines a string syntax for identifying a specific value within a JSON document.

### 9.132.1 Detailed Description

**template<typename RefStringType>**
**class json_pointer< RefStringType >**

JSON Pointer defines a string syntax for identifying a specific value within a JSON document.

**See also**

https://json.nlohmann.me/api/json_pointer/

Definition at line 3549 of file json.hpp.

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

- include/External/json.hpp

## 9.133 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, enable_if_t< std::is_constructible< value_type, Args... >::value, int > = 0> json_ref (Args &&... args)
- **json_ref** (json_ref &&) noexcept=default
- **json_ref** (const json_ref &)=delete
- json_ref & **operator=** (const json_ref &)=delete
- json_ref & **operator=** (json_ref &&)=delete
- value_type moved_or_copied () const
- value_type const & operator* () const
- value_type const * operator-> () const

## 9.133.1 Detailed Description

**template<typename BasicJsonType>**
**class detail::json_ref< BasicJsonType >**

Definition at line 15620 of file json.hpp.

## 9.133.2 Member Typedef Documentation

### 9.133.2.1 value_type

```
template<typename BasicJsonType>
using detail::json_ref< BasicJsonType >::value_type = BasicJsonType
```

Definition at line 15623 of file json.hpp.

## 9.133.3 Constructor & Destructor Documentation

### 9.133.3.1 json_ref() [1/4]

```
template<typename BasicJsonType>
detail::json_ref< BasicJsonType >::json_ref (
            value_type && value)  [inline]
```

Definition at line 15625 of file json.hpp.

### 9.133.3.2 json_ref() [2/4]

```
template<typename BasicJsonType>
detail::json_ref< BasicJsonType >::json_ref (
            const value_type & value)  [inline]
```

Definition at line 15629 of file json.hpp.

#### 9.133.3.3 json_ref() [3/4]

```
template<typename BasicJsonType>
detail::json_ref< BasicJsonType >::json_ref (
            std::initializer_list< json_ref< BasicJsonType > > init)  [inline]
```

Definition at line 15633 of file json.hpp.

#### 9.133.3.4 json_ref() [4/4]

```
template<typename BasicJsonType>
template<class...  Args, enable_if_t< std::is_constructible< value_type, Args...  >::value,
int > = 0>
detail::json_ref< BasicJsonType >::json_ref (
            Args &&...  args)  [inline]
```

Definition at line 15640 of file json.hpp.

### 9.133.4 Member Function Documentation

#### 9.133.4.1 moved_or_copied()

```
template<typename BasicJsonType>
value_type detail::json_ref< BasicJsonType >::moved_or_copied () const  [inline]
```

Definition at line 15651 of file json.hpp.

#### 9.133.4.2 operator*()

```
template<typename BasicJsonType>
value_type const & detail::json_ref< BasicJsonType >::operator* () const  [inline]
```

Definition at line 15660 of file json.hpp.

#### 9.133.4.3 operator->()

```
template<typename BasicJsonType>
value_type const * detail::json_ref< BasicJsonType >::operator-> () const  [inline]
```

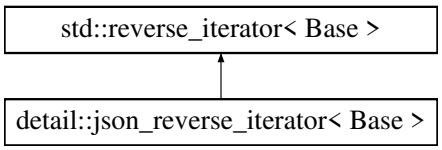
Definition at line 15665 of file json.hpp.

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

- include/External/json.hpp

## 9.134 detail::json_reverse_iterator< Base > Class Template Reference

Inheritance diagram for detail::json_reverse_iterator< Base >:




### Public Types

- using difference_type = std::ptrdiff_t
- using base_iterator = std::reverse_iterator<Base>
- using reference = typename Base::reference

### Public Member Functions

- ∗create reverse iterator from iterator json_reverse_iterator (const typename base_iterator::iterator_type &it) noexcept
- ∗create reverse iterator from base class explicit json_reverse_iterator (const base_iterator &it) noexcept
- ∗post increment (it++) json_reverse_iterator operator++(int) &
- ∗pre increment (++it) json_reverse_iterator &operator++()
- ∗post decrement (it--) json_reverse_iterator operator--(int) &
- ∗pre decrement (--it) json_reverse_iterator &operator--()
- ∗add to iterator json_reverse_iterator & operator+= (difference_type i)
- ∗add to iterator json_reverse_iterator operator+ (difference_type i) const
- ∗subtract from iterator json_reverse_iterator operator- (difference_type i) const
- ∗return difference difference_type operator- (const json_reverse_iterator &other) const
- ∗access to successor reference operator[ ] (difference_type n) const
- ∗return the key of an object iterator auto key () const -> decltype(std::declval< Base >().key())
- ∗return the value of an iterator reference value () const

### 9.134.1 Detailed Description

**template<typename Base>**
**class detail::json_reverse_iterator< Base >**

Definition at line 14469 of file json.hpp.

### 9.134.2 Member Typedef Documentation

#### 9.134.2.1 base_iterator

```
template<typename Base>
using detail::json_reverse_iterator< Base >::base_iterator = std::reverse_iterator<Base>
```

Definition at line 14474 of file json.hpp.

#### 9.134.2.2 difference_type

```
template<typename Base>
using detail::json_reverse_iterator< Base >::difference_type = std::ptrdiff_t
```

Definition at line 14472 of file json.hpp.

#### 9.134.2.3 reference

```
template<typename Base>
using detail::json_reverse_iterator< Base >::reference = typename Base::reference
```

Definition at line 14476 of file json.hpp.

### 9.134.3 Constructor & Destructor Documentation

#### 9.134.3.1 json_reverse_iterator() [1/2]

```
template<typename Base>
*create reverse iterator from iterator detail::json_reverse_iterator< Base >::json_reverse_↩
iterator (
            const typename base_iterator::iterator_type & it)  [inline], [explicit], [noexcept]
```

Definition at line 14479 of file json.hpp.

#### 9.134.3.2 json_reverse_iterator() [2/2]

```
template<typename Base>
*create reverse iterator from base class explicit detail::json_reverse_iterator< Base >::↩
json_reverse_iterator (
            const base_iterator & it)  [inline], [noexcept]
```

Definition at line 14483 of file json.hpp.

### 9.134.4 Member Function Documentation

#### 9.134.4.1 decrement() [1/2]

```
template<typename Base>
*pre detail::json_reverse_iterator< Base >::decrement (
            -- it) &  [inline]
```

Definition at line 14503 of file json.hpp.

#### 9.134.4.2 decrement() [2/2]

```
template<typename Base>
*post detail::json_reverse_iterator< Base >::decrement (
            it-- ) &  [inline]
```

Definition at line 14497 of file json.hpp.

#### 9.134.4.3 increment() [1/2]

```
template<typename Base>
*pre detail::json_reverse_iterator< Base >::increment (
            ++ it) &  [inline]
```

Definition at line 14491 of file json.hpp.

#### 9.134.4.4 increment() [2/2]

```
template<typename Base>
*post detail::json_reverse_iterator< Base >::increment (
            it++ ) &  [inline]
```

Definition at line 14485 of file json.hpp.

#### 9.134.4.5 key()

```
template<typename Base>
*return the key of an object iterator auto detail::json_reverse_iterator< Base >::key () const
-> decltype(std::declval< Base >().key())  [inline]
```

Definition at line 14540 of file json.hpp.

#### 9.134.4.6 operator+()

```
template<typename Base>
*add to iterator json_reverse_iterator detail::json_reverse_iterator< Base >::operator+ (
            difference_type i) const  [inline]
```

Definition at line 14516 of file json.hpp.

#### 9.134.4.7 operator+=()

```
template<typename Base>
*add to iterator json_reverse_iterator & detail::json_reverse_iterator< Base >::operator+= (
            difference_type i)  [inline]
```

Definition at line 14510 of file json.hpp.

#### 9.134.4.8 operator-() [1/2]

```
template<typename Base>
*return difference difference_type detail::json_reverse_iterator< Base >::operator- (
            const json_reverse_iterator< Base > & other) const  [inline]
```

Definition at line 14528 of file json.hpp.

#### 9.134.4.9 operator-() [2/2]

```
template<typename Base>
*subtract from iterator json_reverse_iterator detail::json_reverse_iterator< Base >::operator-
(
            difference_type i) const  [inline]
```

Definition at line 14522 of file json.hpp.

#### 9.134.4.10 operator[]()

```
template<typename Base>
*access to successor reference detail::json_reverse_iterator< Base >::operator[] (
            difference_type n) const  [inline]
```

Definition at line 14534 of file json.hpp.

#### 9.134.4.11 value()

```
template<typename Base>
*return the value of an iterator reference detail::json_reverse_iterator< Base >::value ()
const  [inline]
```

Definition at line 14547 of file json.hpp.

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

- include/External/json.hpp

## 9.135 json_sax< BasicJsonType > Struct Template Reference

SAX interface.

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

### Public Types

- using number_integer_t = typename BasicJsonType::number_integer_t
- using number_unsigned_t = typename BasicJsonType::number_unsigned_t
- using number_float_t = typename BasicJsonType::number_float_t
- using string_t = typename BasicJsonType::string_t
- using binary_t = typename BasicJsonType::binary_t

**Public Member Functions**

- virtual bool null ()=0
  *a null value was read*
- virtual bool boolean (bool val)=0
  *a boolean value was read*
- virtual bool number_integer (number_integer_t val)=0
  *an integer number was read*
- virtual bool number_unsigned (number_unsigned_t val)=0
  *an unsigned integer number was read*
- virtual bool number_float (number_float_t val, const string_t &s)=0
  *a floating-point number was read*
- virtual bool string (string_t &val)=0
  *a string value was read*
- virtual bool binary (binary_t &val)=0
  *a binary value was read*
- virtual bool start_object (std::size_t elements)=0
  *the beginning of an object was read*
- virtual bool key (string_t &val)=0
  *an object key was read*
- virtual bool end_object ()=0
  *the end of an object was read*
- virtual bool start_array (std::size_t elements)=0
  *the beginning of an array was read*
- virtual bool end_array ()=0
  *the end of an array was read*
- virtual bool parse_error (std::size_t position, const std::string &last_token, const detail::exception &ex)=0
  *a parse error occurred*
- **json_sax** (const json_sax &)=default
- **json_sax** (json_sax &&) noexcept=default
- json_sax & **operator=** (const json_sax &)=default
- json_sax & **operator=** (json_sax &&) noexcept=default

## 9.135.1 Detailed Description

**template**<**typename BasicJsonType**>
**struct json_sax**< **BasicJsonType** >

SAX interface.

This class describes the SAX interface used by nlohmann::json::sax_parse. Each function is called in different situations while the input is parsed. The boolean return value informs the parser whether to continue processing the input.

Definition at line 8748 of file json.hpp.

## 9.135.2 Member Typedef Documentation

### 9.135.2.1 binary_t

```
template<typename BasicJsonType>
using json_sax< BasicJsonType >::binary_t = typename BasicJsonType::binary_t
```

Definition at line 8754 of file json.hpp.

#### 9.135.2.2 number_float_t

```
template<typename BasicJsonType>
using json_sax< BasicJsonType >::number_float_t = typename BasicJsonType::number_float_t
```

Definition at line 8752 of file json.hpp.

#### 9.135.2.3 number_integer_t

```
template<typename BasicJsonType>
using json_sax< BasicJsonType >::number_integer_t = typename BasicJsonType::number_integer_t
```

Definition at line 8750 of file json.hpp.

#### 9.135.2.4 number_unsigned_t

```
template<typename BasicJsonType>
using json_sax< BasicJsonType >::number_unsigned_t = typename BasicJsonType::number_unsigned↩
_t
```

Definition at line 8751 of file json.hpp.

#### 9.135.2.5 string_t

```
template<typename BasicJsonType>
using json_sax< BasicJsonType >::string_t = typename BasicJsonType::string_t
```

Definition at line 8753 of file json.hpp.

### 9.135.3 Member Function Documentation

#### 9.135.3.1 binary()

```
template<typename BasicJsonType>
virtual bool json_sax< BasicJsonType >::binary (
            binary_t & val)  [pure virtual]
```

a binary value was read

**Parameters**

| | | |
|---|---|---|
| in | *val* | binary value |

**Returns**

whether parsing should proceed

**Note**

It is safe to move the passed binary value.

#### 9.135.3.2 boolean()

```
template<typename BasicJsonType>
virtual bool json_sax< BasicJsonType >::boolean (
            bool val)  [pure virtual]
```

a boolean value was read

**Parameters**

| | | |
|---|---|---|
| in | *val* | boolean value |

**Returns**

whether parsing should proceed

#### 9.135.3.3 end_array()

```
template<typename BasicJsonType>
virtual bool json_sax< BasicJsonType >::end_array ()  [pure virtual]
```

the end of an array was read

**Returns**

whether parsing should proceed

#### 9.135.3.4 end_object()

```
template<typename BasicJsonType>
virtual bool json_sax< BasicJsonType >::end_object ()  [pure virtual]
```

the end of an object was read

**Returns**

whether parsing should proceed

#### 9.135.3.5 key()

```
template<typename BasicJsonType>
virtual bool json_sax< BasicJsonType >::key (
            string_t & val)  [pure virtual]
```

an object key was read

**Parameters**

| | | |
|---|---|---|
| in | *val* | object key |

**Returns**

whether parsing should proceed

**Note**

It is safe to move the passed string.

#### 9.135.3.6 null()

```
template<typename BasicJsonType>
virtual bool json_sax< BasicJsonType >::null ()  [pure virtual]
```

a null value was read

**Returns**

whether parsing should proceed

#### 9.135.3.7 number_float()

```
template<typename BasicJsonType>
virtual bool json_sax< BasicJsonType >::number_float (
            number_float_t val,
            const string_t & s)  [pure virtual]
```

a floating-point number was read

**Parameters**

| | | |
|---|---|---|
| in | *val* | floating-point value |
| in | *s* | raw token value |

**Returns**

whether parsing should proceed

#### 9.135.3.8 number_integer()

```
template<typename BasicJsonType>
virtual bool json_sax< BasicJsonType >::number_integer (
            number_integer_t val)  [pure virtual]
```

an integer number was read

**Parameters**

| | | |
|---|---|---|
| in | *val* | integer value |

**Returns**

whether parsing should proceed

#### 9.135.3.9 number_unsigned()

```
template<typename BasicJsonType>
virtual bool json_sax< BasicJsonType >::number_unsigned (
            number_unsigned_t val)  [pure virtual]
```

an unsigned integer number was read

**Parameters**

| | | |
|---|---|---|
| in | *val* | unsigned integer value |

**Returns**

whether parsing should proceed

#### 9.135.3.10 parse_error()

```
template<typename BasicJsonType>
virtual bool json_sax< BasicJsonType >::parse_error (
            std::size_t position,
            const std::string & last_token,
            const detail::exception & ex)  [pure virtual]
```

a parse error occurred

**Parameters**

| | | |
|---|---|---|
| in | *position* | the position in the input where the error occurs |
| in | *last_token* | the last read token |
| in | *ex* | an exception object describing the error |

**Returns**

whether parsing should proceed (must return false)

#### 9.135.3.11 start_array()

```
template<typename BasicJsonType>
virtual bool json_sax< BasicJsonType >::start_array (
            std::size_t elements)  [pure virtual]
```

the beginning of an array was read

**Parameters**

| | | |
|---|---|---|
| in | *elements* | number of array elements or -1 if unknown |

**Returns**

whether parsing should proceed

**Note**

binary formats may report the number of elements

#### 9.135.3.12 start_object()

```
template<typename BasicJsonType>
virtual bool json_sax< BasicJsonType >::start_object (
            std::size_t elements)  [pure virtual]
```

the beginning of an object was read

**Parameters**

| | | |
|---|---|---|
| in | *elements* | number of object elements or -1 if unknown |

**Returns**

whether parsing should proceed

**Note**

binary formats may report the number of elements

#### 9.135.3.13 string()

```
template<typename BasicJsonType>
virtual bool json_sax< BasicJsonType >::string (
            string_t & val)  [pure virtual]
```

a string value was read

**Parameters**

| | | |
|---|---|---|
| in | *val* | string value |

**Returns**

whether parsing should proceed

**Note**

It is safe to move the passed string value.

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

- include/External/json.hpp

# 9.136 detail::json_sax_acceptor< BasicJsonType > Class Template Reference

### Public Types

- using number_integer_t = typename BasicJsonType::number_integer_t
- using number_unsigned_t = typename BasicJsonType::number_unsigned_t
- using number_float_t = typename BasicJsonType::number_float_t
- using string_t = typename BasicJsonType::string_t
- using binary_t = typename BasicJsonType::binary_t

### Public Member Functions

- bool null ()
- bool boolean (bool)
- bool number_integer (number_integer_t)
- bool number_unsigned (number_unsigned_t)
- bool number_float (number_float_t, const string_t &)
- bool string (string_t &)
- bool binary (binary_t &)
- bool start_object (std::size_t=detail::unknown_size())
- bool key (string_t &)
- bool end_object ()
- bool start_array (std::size_t=detail::unknown_size())
- bool end_array ()
- bool parse_error (std::size_t, const std::string &, const detail::exception &)

## 9.136.1 Detailed Description

**template<typename BasicJsonType>**
**class detail::json_sax_acceptor< BasicJsonType >**

Definition at line 9626 of file json.hpp.

## 9.136.2 Member Typedef Documentation

### 9.136.2.1 binary_t

```
template<typename BasicJsonType>
using detail::json_sax_acceptor< BasicJsonType >::binary_t = typename BasicJsonType::binary_t
```

Definition at line 9633 of file json.hpp.

### 9.136.2.2 number_float_t

```
template<typename BasicJsonType>
using detail::json_sax_acceptor< BasicJsonType >::number_float_t = typename BasicJsonType::↩
number_float_t
```

Definition at line 9631 of file json.hpp.

#### 9.136.2.3 number_integer_t

```
template<typename BasicJsonType>
using detail::json_sax_acceptor< BasicJsonType >::number_integer_t = typename BasicJson↩
Type::number_integer_t
```

Definition at line 9629 of file json.hpp.

#### 9.136.2.4 number_unsigned_t

```
template<typename BasicJsonType>
using detail::json_sax_acceptor< BasicJsonType >::number_unsigned_t = typename BasicJson↩
Type::number_unsigned_t
```

Definition at line 9630 of file json.hpp.

#### 9.136.2.5 string_t

```
template<typename BasicJsonType>
using detail::json_sax_acceptor< BasicJsonType >::string_t = typename BasicJsonType::string_t
```

Definition at line 9632 of file json.hpp.

### 9.136.3 Member Function Documentation

#### 9.136.3.1 binary()

```
template<typename BasicJsonType>
bool detail::json_sax_acceptor< BasicJsonType >::binary (
            binary_t & )  [inline]
```

Definition at line 9665 of file json.hpp.

#### 9.136.3.2 boolean()

```
template<typename BasicJsonType>
bool detail::json_sax_acceptor< BasicJsonType >::boolean (
            bool )  [inline]
```

Definition at line 9640 of file json.hpp.

#### 9.136.3.3 end_array()

```
template<typename BasicJsonType>
bool detail::json_sax_acceptor< BasicJsonType >::end_array ()  [inline]
```

Definition at line 9690 of file json.hpp.

#### 9.136.3.4 end_object()

```
template<typename BasicJsonType>
bool detail::json_sax_acceptor< BasicJsonType >::end_object ()  [inline]
```

Definition at line 9680 of file json.hpp.

#### 9.136.3.5 key()

```
template<typename BasicJsonType>
bool detail::json_sax_acceptor< BasicJsonType >::key (
            string_t & )  [inline]
```

Definition at line 9675 of file json.hpp.

#### 9.136.3.6 null()

```
template<typename BasicJsonType>
bool detail::json_sax_acceptor< BasicJsonType >::null ()  [inline]
```

Definition at line 9635 of file json.hpp.

#### 9.136.3.7 number_float()

```
template<typename BasicJsonType>
bool detail::json_sax_acceptor< BasicJsonType >::number_float (
            number_float_t ,
            const string_t & )  [inline]
```

Definition at line 9655 of file json.hpp.

#### 9.136.3.8 number_integer()

```
template<typename BasicJsonType>
bool detail::json_sax_acceptor< BasicJsonType >::number_integer (
            number_integer_t )  [inline]
```

Definition at line 9645 of file json.hpp.

#### 9.136.3.9 number_unsigned()

```
template<typename BasicJsonType>
bool detail::json_sax_acceptor< BasicJsonType >::number_unsigned (
            number_unsigned_t )  [inline]
```

Definition at line 9650 of file json.hpp.

#### 9.136.3.10 parse_error()

```
template<typename BasicJsonType>
bool detail::json_sax_acceptor< BasicJsonType >::parse_error (
            std::size_t ,
            const std::string & ,
            const detail::exception & )  [inline]
```

Definition at line 9695 of file json.hpp.

#### 9.136.3.11 start_array()

```
template<typename BasicJsonType>
bool detail::json_sax_acceptor< BasicJsonType >::start_array (
            std::size_t  = detail::unknown_size())  [inline]
```

Definition at line 9685 of file json.hpp.

#### 9.136.3.12 start_object()

```
template<typename BasicJsonType>
bool detail::json_sax_acceptor< BasicJsonType >::start_object (
            std::size_t  = detail::unknown_size())  [inline]
```

Definition at line 9670 of file json.hpp.

#### 9.136.3.13 string()

```
template<typename BasicJsonType>
bool detail::json_sax_acceptor< BasicJsonType >::string (
            string_t & )  [inline]
```

Definition at line 9660 of file json.hpp.

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

- include/External/json.hpp

## 9.137 detail::json_sax_dom_callback_parser< BasicJsonType, InputAdapterType > Class Template Reference

### Public Types

- using number_integer_t = typename BasicJsonType::number_integer_t
- using number_unsigned_t = typename BasicJsonType::number_unsigned_t
- using number_float_t = typename BasicJsonType::number_float_t
- using string_t = typename BasicJsonType::string_t
- using binary_t = typename BasicJsonType::binary_t
- using parser_callback_t = typename BasicJsonType::parser_callback_t
- using parse_event_t = typename BasicJsonType::parse_event_t
- using lexer_t = lexer<BasicJsonType, InputAdapterType>

**Public Member Functions**

- json_sax_dom_callback_parser (BasicJsonType &r, parser_callback_t cb, const bool allow_exceptions_↩ =true, lexer_t ∗lexer_=nullptr)
- **json_sax_dom_callback_parser** (const json_sax_dom_callback_parser &)=delete
- **json_sax_dom_callback_parser** (json_sax_dom_callback_parser &&)=default
- json_sax_dom_callback_parser & **operator=** (const json_sax_dom_callback_parser &)=delete
- json_sax_dom_callback_parser & **operator=** (json_sax_dom_callback_parser &&)=default
- bool null ()
- bool boolean (bool val)
- bool number_integer (number_integer_t val)
- bool number_unsigned (number_unsigned_t val)
- bool number_float (number_float_t val, const string_t &)
- bool string (string_t &val)
- bool binary (binary_t &val)
- bool start_object (std::size_t len)
- bool key (string_t &val)
- bool end_object ()
- bool start_array (std::size_t len)
- bool end_array ()
- template<class Exception>
  bool parse_error (std::size_t, const std::string &, const Exception &ex)
- constexpr bool is_errored () const

## 9.137.1 Detailed Description

**template<typename BasicJsonType, typename InputAdapterType>**
**class detail::json_sax_dom_callback_parser< BasicJsonType, InputAdapterType >**

Definition at line 9189 of file json.hpp.

## 9.137.2 Member Typedef Documentation

### 9.137.2.1 binary_t

```
template<typename BasicJsonType, typename InputAdapterType>
using detail::json_sax_dom_callback_parser< BasicJsonType, InputAdapterType >::binary_t =
typename BasicJsonType::binary_t
```

Definition at line 9196 of file json.hpp.

### 9.137.2.2 lexer_t

```
template<typename BasicJsonType, typename InputAdapterType>
using detail::json_sax_dom_callback_parser< BasicJsonType, InputAdapterType >::lexer_t =
lexer<BasicJsonType, InputAdapterType>
```

Definition at line 9199 of file json.hpp.

#### 9.137.2.3 number_float_t

```
template<typename BasicJsonType, typename InputAdapterType>
using detail::json_sax_dom_callback_parser< BasicJsonType, InputAdapterType >::number_float_t
= typename BasicJsonType::number_float_t
```

Definition at line 9194 of file json.hpp.

#### 9.137.2.4 number_integer_t

```
template<typename BasicJsonType, typename InputAdapterType>
using detail::json_sax_dom_callback_parser< BasicJsonType, InputAdapterType >::number_integer↩
_t = typename BasicJsonType::number_integer_t
```

Definition at line 9192 of file json.hpp.

#### 9.137.2.5 number_unsigned_t

```
template<typename BasicJsonType, typename InputAdapterType>
using detail::json_sax_dom_callback_parser< BasicJsonType, InputAdapterType >::number_unsigned↩
_t = typename BasicJsonType::number_unsigned_t
```

Definition at line 9193 of file json.hpp.

#### 9.137.2.6 parse_event_t

```
template<typename BasicJsonType, typename InputAdapterType>
using detail::json_sax_dom_callback_parser< BasicJsonType, InputAdapterType >::parse_event_t =
typename BasicJsonType::parse_event_t
```

Definition at line 9198 of file json.hpp.

#### 9.137.2.7 parser_callback_t

```
template<typename BasicJsonType, typename InputAdapterType>
using detail::json_sax_dom_callback_parser< BasicJsonType, InputAdapterType >::parser_callback↩
_t = typename BasicJsonType::parser_callback_t
```

Definition at line 9197 of file json.hpp.

#### 9.137.2.8 string_t

```
template<typename BasicJsonType, typename InputAdapterType>
using detail::json_sax_dom_callback_parser< BasicJsonType, InputAdapterType >::string_t =
typename BasicJsonType::string_t
```

Definition at line 9195 of file json.hpp.

## 9.137.3 Constructor & Destructor Documentation

### 9.137.3.1 json_sax_dom_callback_parser()

```
template<typename BasicJsonType, typename InputAdapterType>
detail::json_sax_dom_callback_parser< BasicJsonType, InputAdapterType >::json_sax_dom_callback↩
_parser (
            BasicJsonType & r,
            parser_callback_t cb,
            const bool allow_exceptions_ = true,
            lexer_t * lexer_ = nullptr)  [inline]
```

Definition at line 9201 of file json.hpp.

## 9.137.4 Member Function Documentation

### 9.137.4.1 binary()

```
template<typename BasicJsonType, typename InputAdapterType>
bool detail::json_sax_dom_callback_parser< BasicJsonType, InputAdapterType >::binary (
            binary_t & val)  [inline]
```

Definition at line 9253 of file json.hpp.

### 9.137.4.2 boolean()

```
template<typename BasicJsonType, typename InputAdapterType>
bool detail::json_sax_dom_callback_parser< BasicJsonType, InputAdapterType >::boolean (
            bool val)  [inline]
```

Definition at line 9223 of file json.hpp.

### 9.137.4.3 end_array()

```
template<typename BasicJsonType, typename InputAdapterType>
bool detail::json_sax_dom_callback_parser< BasicJsonType, InputAdapterType >::end_array ()
[inline]
```

Definition at line 9390 of file json.hpp.

### 9.137.4.4 end_object()

```
template<typename BasicJsonType, typename InputAdapterType>
bool detail::json_sax_dom_callback_parser< BasicJsonType, InputAdapterType >::end_object ()
[inline]
```

Definition at line 9308 of file json.hpp.

#### 9.137.4.5 is_errored()

```
template<typename BasicJsonType, typename InputAdapterType>
bool detail::json_sax_dom_callback_parser< BasicJsonType, InputAdapterType >::is_errored ()
const  [inline], [constexpr]
```

Definition at line 9449 of file json.hpp.

#### 9.137.4.6 key()

```
template<typename BasicJsonType, typename InputAdapterType>
bool detail::json_sax_dom_callback_parser< BasicJsonType, InputAdapterType >::key (
            string_t & val)  [inline]
```

Definition at line 9291 of file json.hpp.

#### 9.137.4.7 null()

```
template<typename BasicJsonType, typename InputAdapterType>
bool detail::json_sax_dom_callback_parser< BasicJsonType, InputAdapterType >::null ()  [inline]
```

Definition at line 9217 of file json.hpp.

#### 9.137.4.8 number_float()

```
template<typename BasicJsonType, typename InputAdapterType>
bool detail::json_sax_dom_callback_parser< BasicJsonType, InputAdapterType >::number_float (
            number_float_t val,
            const string_t & )  [inline]
```

Definition at line 9241 of file json.hpp.

#### 9.137.4.9 number_integer()

```
template<typename BasicJsonType, typename InputAdapterType>
bool detail::json_sax_dom_callback_parser< BasicJsonType, InputAdapterType >::number_integer (
            number_integer_t val)  [inline]
```

Definition at line 9229 of file json.hpp.

#### 9.137.4.10 number_unsigned()

```
template<typename BasicJsonType, typename InputAdapterType>
bool detail::json_sax_dom_callback_parser< BasicJsonType, InputAdapterType >::number_unsigned
(
            number_unsigned_t val)  [inline]
```

Definition at line 9235 of file json.hpp.

#### 9.137.4.11 parse_error()

```
template<typename BasicJsonType, typename InputAdapterType>
template<class Exception>
bool detail::json_sax_dom_callback_parser< BasicJsonType, InputAdapterType >::parse_error (
            std::size_t ,
            const std::string & ,
            const Exception & ex)  [inline]
```

Definition at line 9437 of file json.hpp.

#### 9.137.4.12 start_array()

```
template<typename BasicJsonType, typename InputAdapterType>
bool detail::json_sax_dom_callback_parser< BasicJsonType, InputAdapterType >::start_array (
            std::size_t len)  [inline]
```

Definition at line 9358 of file json.hpp.

#### 9.137.4.13 start_object()

```
template<typename BasicJsonType, typename InputAdapterType>
bool detail::json_sax_dom_callback_parser< BasicJsonType, InputAdapterType >::start_object (
            std::size_t len)  [inline]
```

Definition at line 9259 of file json.hpp.

#### 9.137.4.14 string()

```
template<typename BasicJsonType, typename InputAdapterType>
bool detail::json_sax_dom_callback_parser< BasicJsonType, InputAdapterType >::string (
            string_t & val)  [inline]
```

Definition at line 9247 of file json.hpp.

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

- include/External/json.hpp

## 9.138 detail::json_sax_dom_parser< BasicJsonType, InputAdapterType > Class Template Reference

SAX implementation to create a JSON value from SAX events.

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

### Public Types

- using number_integer_t = typename BasicJsonType::number_integer_t
- using number_unsigned_t = typename BasicJsonType::number_unsigned_t
- using number_float_t = typename BasicJsonType::number_float_t
- using string_t = typename BasicJsonType::string_t
- using binary_t = typename BasicJsonType::binary_t
- using lexer_t = lexer<BasicJsonType, InputAdapterType>

### Public Member Functions

- json_sax_dom_parser (BasicJsonType &r, const bool allow_exceptions_=true, lexer_t *lexer_=nullptr)
- **json_sax_dom_parser** (const json_sax_dom_parser &)=delete
- **json_sax_dom_parser** (json_sax_dom_parser &&)=default
- json_sax_dom_parser & **operator=** (const json_sax_dom_parser &)=delete
- json_sax_dom_parser & **operator=** (json_sax_dom_parser &&)=default
- bool null ()
- bool boolean (bool val)
- bool number_integer (number_integer_t val)
- bool number_unsigned (number_unsigned_t val)
- bool number_float (number_float_t val, const string_t &)
- bool string (string_t &val)
- bool binary (binary_t &val)
- bool start_object (std::size_t len)
- bool key (string_t &val)
- bool end_object ()
- bool start_array (std::size_t len)
- bool end_array ()
- template<class Exception>
  bool parse_error (std::size_t, const std::string &, const Exception &ex)
- constexpr bool is_errored () const

## 9.138.1 Detailed Description

**template<typename BasicJsonType, typename InputAdapterType>**
**class detail::json_sax_dom_parser< BasicJsonType, InputAdapterType >**

SAX implementation to create a JSON value from SAX events.

This class implements the json_sax interface and processes the SAX events to create a JSON value which makes it basically a DOM parser. The structure or hierarchy of the JSON value is managed by the stack `ref_stack` which contains a pointer to the respective array or object for each recursion depth.

After successful parsing, the value that is passed by reference to the constructor contains the parsed value.

**Template Parameters**

| | |
|---|---|
| *BasicJsonType* | the JSON type |

Definition at line 8883 of file json.hpp.

## 9.138.2 Member Typedef Documentation

### 9.138.2.1 binary_t

```
template<typename BasicJsonType, typename InputAdapterType>
using detail::json_sax_dom_parser< BasicJsonType, InputAdapterType >::binary_t = typename
BasicJsonType::binary_t
```

Definition at line 8890 of file json.hpp.

### 9.138.2.2 lexer_t

```
template<typename BasicJsonType, typename InputAdapterType>
using detail::json_sax_dom_parser< BasicJsonType, InputAdapterType >::lexer_t = lexer<Basic↩
JsonType, InputAdapterType>
```

Definition at line 8891 of file json.hpp.

### 9.138.2.3 number_float_t

```
template<typename BasicJsonType, typename InputAdapterType>
using detail::json_sax_dom_parser< BasicJsonType, InputAdapterType >::number_float_t = typename
BasicJsonType::number_float_t
```

Definition at line 8888 of file json.hpp.

### 9.138.2.4 number_integer_t

```
template<typename BasicJsonType, typename InputAdapterType>
using detail::json_sax_dom_parser< BasicJsonType, InputAdapterType >::number_integer_t =
typename BasicJsonType::number_integer_t
```

Definition at line 8886 of file json.hpp.

### 9.138.2.5 number_unsigned_t

```
template<typename BasicJsonType, typename InputAdapterType>
using detail::json_sax_dom_parser< BasicJsonType, InputAdapterType >::number_unsigned_t =
typename BasicJsonType::number_unsigned_t
```

Definition at line 8887 of file json.hpp.

### 9.138.2.6 string_t

```
template<typename BasicJsonType, typename InputAdapterType>
using detail::json_sax_dom_parser< BasicJsonType, InputAdapterType >::string_t = typename
BasicJsonType::string_t
```

Definition at line 8889 of file json.hpp.

## 9.138.3 Constructor & Destructor Documentation

### 9.138.3.1 json_sax_dom_parser()

```
template<typename BasicJsonType, typename InputAdapterType>
detail::json_sax_dom_parser< BasicJsonType, InputAdapterType >::json_sax_dom_parser (
            BasicJsonType & r,
            const bool allow_exceptions_ = true,
            lexer_t * lexer_ = nullptr)  [inline], [explicit]
```

**Parameters**

| | | |
|---|---|---|
| in,out | *r* | reference to a JSON value that is manipulated while parsing |

| | | |
|---|---|---|
| in | *allow_↩ exceptions_* | whether parse errors yield exceptions |

Definition at line 8898 of file json.hpp.

## 9.138.4 Member Function Documentation

### 9.138.4.1 binary()

```
template<typename BasicJsonType, typename InputAdapterType>
bool detail::json_sax_dom_parser< BasicJsonType, InputAdapterType >::binary (
            binary_t & val)  [inline]
```

Definition at line 8945 of file json.hpp.

### 9.138.4.2 boolean()

```
template<typename BasicJsonType, typename InputAdapterType>
bool detail::json_sax_dom_parser< BasicJsonType, InputAdapterType >::boolean (
            bool val)  [inline]
```

Definition at line 8915 of file json.hpp.

### 9.138.4.3 end_array()

```
template<typename BasicJsonType, typename InputAdapterType>
bool detail::json_sax_dom_parser< BasicJsonType, InputAdapterType >::end_array ()  [inline]
```

Definition at line 9023 of file json.hpp.

### 9.138.4.4 end_object()

```
template<typename BasicJsonType, typename InputAdapterType>
bool detail::json_sax_dom_parser< BasicJsonType, InputAdapterType >::end_object ()  [inline]
```

Definition at line 8984 of file json.hpp.

### 9.138.4.5 is_errored()

```
template<typename BasicJsonType, typename InputAdapterType>
bool detail::json_sax_dom_parser< BasicJsonType, InputAdapterType >::is_errored () const
[inline], [constexpr]
```

Definition at line 9054 of file json.hpp.

#### 9.138.4.6 key()

```
template<typename BasicJsonType, typename InputAdapterType>
bool detail::json_sax_dom_parser< BasicJsonType, InputAdapterType >::key (
            string_t & val)  [inline]
```

Definition at line 8974 of file json.hpp.

#### 9.138.4.7 null()

```
template<typename BasicJsonType, typename InputAdapterType>
bool detail::json_sax_dom_parser< BasicJsonType, InputAdapterType >::null ()  [inline]
```

Definition at line 8909 of file json.hpp.

#### 9.138.4.8 number_float()

```
template<typename BasicJsonType, typename InputAdapterType>
bool detail::json_sax_dom_parser< BasicJsonType, InputAdapterType >::number_float (
            number_float_t val,
            const string_t & )  [inline]
```

Definition at line 8933 of file json.hpp.

#### 9.138.4.9 number_integer()

```
template<typename BasicJsonType, typename InputAdapterType>
bool detail::json_sax_dom_parser< BasicJsonType, InputAdapterType >::number_integer (
            number_integer_t val)  [inline]
```

Definition at line 8921 of file json.hpp.

#### 9.138.4.10 number_unsigned()

```
template<typename BasicJsonType, typename InputAdapterType>
bool detail::json_sax_dom_parser< BasicJsonType, InputAdapterType >::number_unsigned (
            number_unsigned_t val)  [inline]
```

Definition at line 8927 of file json.hpp.

#### 9.138.4.11 parse_error()

```
template<typename BasicJsonType, typename InputAdapterType>
template<class Exception>
bool detail::json_sax_dom_parser< BasicJsonType, InputAdapterType >::parse_error (
            std::size_t ,
            const std::string & ,
            const Exception & ex)  [inline]
```

Definition at line 9042 of file json.hpp.

#### 9.138.4.12 start_array()

```
template<typename BasicJsonType, typename InputAdapterType>
bool detail::json_sax_dom_parser< BasicJsonType, InputAdapterType >::start_array (
            std::size_t len)  [inline]
```

Definition at line 9002 of file json.hpp.

#### 9.138.4.13 start_object()

```
template<typename BasicJsonType, typename InputAdapterType>
bool detail::json_sax_dom_parser< BasicJsonType, InputAdapterType >::start_object (
            std::size_t len)  [inline]
```

Definition at line 8951 of file json.hpp.

#### 9.138.4.14 string()

```
template<typename BasicJsonType, typename InputAdapterType>
bool detail::json_sax_dom_parser< BasicJsonType, InputAdapterType >::string (
            string_t & val)  [inline]
```

Definition at line 8939 of file json.hpp.

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

- include/External/json.hpp

## 9.139 std::less< ::nlohmann::detail::value_t > Struct Reference

### Public Member Functions

- bool operator() (::nlohmann::detail::value_t lhs, ::nlohmann::detail::value_t rhs) const noexcept
  *compare two value_t enum values*

### 9.139.1 Detailed Description

Definition at line 25468 of file json.hpp.

### 9.139.2 Member Function Documentation

#### 9.139.2.1 operator()()

```
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

Definition at line 25474 of file json.hpp.

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

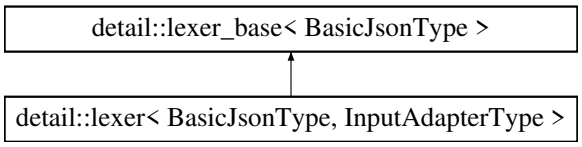
- include/External/json.hpp

## 9.140 detail::lexer< BasicJsonType, InputAdapterType > Class Template Reference

lexical analysis

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

Inheritance diagram for detail::lexer< BasicJsonType, InputAdapterType >:




**Public Types**

- using token_type = typename lexer_base<BasicJsonType>::token_type

### Public Types inherited from detail::lexer_base< BasicJsonType >

- enum class token_type {
  uninitialized , literal_true , literal_false , literal_null ,
  value_string , value_unsigned , value_integer , value_float ,
  begin_array , begin_object , end_array , end_object ,
  name_separator , value_separator , parse_error , end_of_input ,
  literal_or_value }

  *token types for the parser*

**Public Member Functions**

- lexer (InputAdapterType &&adapter, bool ignore_comments_=false) noexcept
- **lexer** (const lexer &)=delete
- **lexer** (lexer &&)=default
- lexer & **operator=** (lexer &)=delete
- lexer & **operator=** (lexer &&)=default
- 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*
- string_t & get_string ()

  *return current string value (implicitly resets the token; useful only once)*
- constexpr position_t get_position () const noexcept

  *return position of last read token*
- std::string get_token_string () const
- JSON_HEDLEY_RETURNS_NON_NULL constexpr const char * get_error_message () const noexcept

  *return syntax error message*
- bool skip_bom ()

  *skip the UTF-8 byte order mark*
- void skip_whitespace ()
- token_type scan ()

**Additional Inherited Members**

## Static Public Member Functions inherited from detail::lexer_base< BasicJsonType >

- JSON_HEDLEY_RETURNS_NON_NULL static JSON_HEDLEY_CONST const char * token_type_name (const token_type t) noexcept

  *return name of values of type token_type (only used for errors)*

## 9.140.1 Detailed Description

**template<typename BasicJsonType, typename InputAdapterType>**
**class detail::lexer< BasicJsonType, InputAdapterType >**

lexical analysis

This class organizes the lexical analysis during JSON deserialization.

Definition at line 7200 of file json.hpp.

## 9.140.2 Member Typedef Documentation

### 9.140.2.1 token_type

```
template<typename BasicJsonType, typename InputAdapterType>
using detail::lexer< BasicJsonType, InputAdapterType >::token_type = typename lexer_base<Basic↩
JsonType>::token_type
```

Definition at line 7210 of file json.hpp.

## 9.140.3 Constructor & Destructor Documentation

### 9.140.3.1 lexer()

```
template<typename BasicJsonType, typename InputAdapterType>
detail::lexer< BasicJsonType, InputAdapterType >::lexer (
            InputAdapterType && adapter,
            bool ignore_comments_ = false)  [inline], [explicit], [noexcept]
```

Definition at line 7212 of file json.hpp.

## 9.140.4 Member Function Documentation

### 9.140.4.1 get_error_message()

```
template<typename BasicJsonType, typename InputAdapterType>
JSON_HEDLEY_RETURNS_NON_NULL constexpr const char * detail::lexer< BasicJsonType, Input↩
AdapterType >::get_error_message () const  [inline], [constexpr], [noexcept]
```

return syntax error message

Definition at line 8570 of file json.hpp.

### 9.140.4.2 get_number_float()

```
template<typename BasicJsonType, typename InputAdapterType>
number_float_t detail::lexer< BasicJsonType, InputAdapterType >::get_number_float () const
[inline], [constexpr], [noexcept]
```

return floating-point value

Definition at line 8516 of file json.hpp.

### 9.140.4.3 get_number_integer()

```
template<typename BasicJsonType, typename InputAdapterType>
number_integer_t detail::lexer< BasicJsonType, InputAdapterType >::get_number_integer () const
[inline], [constexpr], [noexcept]
```

return integer value

Definition at line 8504 of file json.hpp.

### 9.140.4.4 get_number_unsigned()

```
template<typename BasicJsonType, typename InputAdapterType>
number_unsigned_t detail::lexer< BasicJsonType, InputAdapterType >::get_number_unsigned ()
const  [inline], [constexpr], [noexcept]
```

return unsigned integer value

Definition at line 8510 of file json.hpp.

#### 9.140.4.5 get_position()

```
template<typename BasicJsonType, typename InputAdapterType>
position_t detail::lexer< BasicJsonType, InputAdapterType >::get_position () const  [inline],
[constexpr], [noexcept]
```

return position of last read token

Definition at line 8537 of file json.hpp.

#### 9.140.4.6 get_string()

```
template<typename BasicJsonType, typename InputAdapterType>
string_t & detail::lexer< BasicJsonType, InputAdapterType >::get_string ()  [inline]
```

return current string value (implicitly resets the token; useful only once)

Definition at line 8522 of file json.hpp.

#### 9.140.4.7 get_token_string()

```
template<typename BasicJsonType, typename InputAdapterType>
std::string detail::lexer< BasicJsonType, InputAdapterType >::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.

Definition at line 8545 of file json.hpp.

#### 9.140.4.8 scan()

```
template<typename BasicJsonType, typename InputAdapterType>
token_type detail::lexer< BasicJsonType, InputAdapterType >::scan ()  [inline]
```

Definition at line 8606 of file json.hpp.

#### 9.140.4.9 skip_bom()

```
template<typename BasicJsonType, typename InputAdapterType>
bool detail::lexer< BasicJsonType, InputAdapterType >::skip_bom ()  [inline]
```

skip the UTF-8 byte order mark

**Returns**

true iff there is no BOM or the correct BOM has been skipped

Definition at line 8583 of file json.hpp.

#### 9.140.4.10 skip_whitespace()

```
template<typename BasicJsonType, typename InputAdapterType>
void detail::lexer< BasicJsonType, InputAdapterType >::skip_whitespace ()  [inline]
```

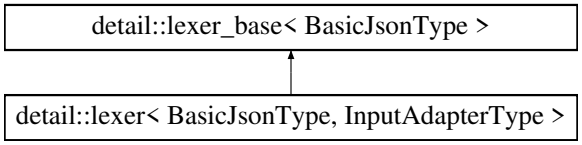
Definition at line 8597 of file json.hpp.

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

- include/External/json.hpp

## 9.141 detail::lexer_base< BasicJsonType > Class Template Reference

Inheritance diagram for detail::lexer_base< BasicJsonType >:




### Public Types

- enum class token_type {
  uninitialized , literal_true , literal_false , literal_null ,
  value_string , value_unsigned , value_integer , value_float ,
  begin_array , begin_object , end_array , end_object ,
  name_separator , value_separator , parse_error , end_of_input ,
  literal_or_value }
  *token types for the parser*

### Static Public Member Functions

- JSON_HEDLEY_RETURNS_NON_NULL static JSON_HEDLEY_CONST const char * token_type_name (const token_type t) noexcept
  *return name of values of type token_type (only used for errors)*

### 9.141.1 Detailed Description

**template<typename BasicJsonType>**
**class detail::lexer_base< BasicJsonType >**

Definition at line 7123 of file json.hpp.

### 9.141.2 Member Enumeration Documentation

#### 9.141.2.1 token_type

```
template<typename BasicJsonType>
enum class detail::lexer_base::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) |

Definition at line 7127 of file json.hpp.

### 9.141.3 Member Function Documentation

#### 9.141.3.1 token_type_name()

```
template<typename BasicJsonType>
JSON_HEDLEY_RETURNS_NON_NULL static JSON_HEDLEY_CONST const char * detail::lexer_base< Basic↩
JsonType >::token_type_name (
            const token_type t)  [inline], [static], [noexcept]
```

return name of values of type token_type (only used for errors)

Definition at line 7151 of file json.hpp.

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

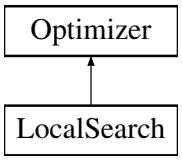
- include/External/json.hpp

## 9.142 LocalSearch Class Reference

Implements a local search optimization algorithm.

`#include <LocalSearch.h>`

Inheritance diagram for LocalSearch:



#### Public Member Functions

- LocalSearch (SolutionBuilder &solutionBuilder, Problem &problem, int maxIterations, double delta, int num↩ Neighbors)

  *Constructs a LocalSearch optimizer.*
- double optimize () override

  *Executes the local search optimization process.*

### Public Member Functions inherited from Optimizer

- Optimizer (SolutionBuilder &solutionBuilder, Problem &problem, int maxIterations)

  *Constructs an Optimizer.*
- virtual ~**Optimizer** ()=default

  *Virtual destructor for safe polymorphic deletion.*

- double getBestFitness ()
- std::vector< double > & getBestSolution ()
- std::vector< double > & getBestFitnesses ()
- std::vector< std::vector< double > > & getSolutions ()
- int getMaxIterations ()
- Problem & getProblem ()
- SolutionBuilder & getSolutionBuilder ()

#### Additional Inherited Members

### Protected Attributes inherited from Optimizer

- Problem & problem

  *Optimization problem definition.*
- SolutionBuilder & solutionBuilder

  *Solution generation utility.*
- int maxIterations

  *Maximum number of iterations.*
- std::vector< double > bestSolution

  *Best solution found.*
- std::vector< double > bestFitnesses

  *Best fitness value so far per iteration.*
- std::vector< std::vector< double > > solutions

  *All generated solutions.*

## 9.142.1 Detailed Description

Implements a local search optimization algorithm.

The LocalSearch optimizer explores the neighborhood of the current solution by generating candidate neighbors within a fixed delta. The search continues until a local minimum is reached or the maximum number of iterations is exceeded.

This class supports both single-run local search and repeated local search depending on the iteration limit provided.

Definition at line 26 of file LocalSearch.h.

### 9.142.2 Constructor & Destructor Documentation

#### 9.142.2.1 LocalSearch()

```
LocalSearch::LocalSearch (
            SolutionBuilder & solutionBuilder,
            Problem & problem,
            int maxIterations,
            double delta,
            int numNeighbors)  [inline]
```

Constructs a LocalSearch optimizer.

**Parameters**

| | |
|---|---|
| *solutionBuilder* | Reference to the solution generator. |
| *problem* | Reference to the optimization problem. |
| *maxIterations* | Maximum number of local search iterations. |
| *delta* | Neighborhood radius for neighbor generation. |
| *numNeighbors* | Number of neighbors sampled per iteration. |

Definition at line 52 of file LocalSearch.h.

### 9.142.3 Member Function Documentation

#### 9.142.3.1 optimize()

```
double LocalSearch::optimize ()  [override], [virtual]
```

Executes the local search optimization process.

**Returns**

The functions execution time.

Implements Optimizer.

Definition at line 55 of file LocalSearch.cpp.

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

- include/Optimizer/LocalSearch.h
- src/Optimizer/LocalSearch.cpp

## 9.143 detail::make_void< Ts > Struct Template Reference

**Public Types**

- using type = void

### 9.143.1 Detailed Description

**template<typename ... Ts>**
**struct detail::make_void< Ts >**

Definition at line 262 of file json.hpp.

### 9.143.2 Member Typedef Documentation

#### 9.143.2.1 type

```
template<typename ...  Ts>
using detail::make_void< Ts >::type = void
```

Definition at line 264 of file json.hpp.

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

- include/External/json.hpp

## 9.144 MersenneTwister Class Reference

```
#include <mt.h>
```

#### Public Member Functions

- MersenneTwister (void)
- ~MersenneTwister (void)
- double random (void)
- void print (void)
- void init_genrand (unsigned long s)
- void init_by_array (unsigned long *init_key, int key_length)
- unsigned long genrand_int32 (void)
- long genrand_int31 (void)
- double genrand_real1 (void)
- double genrand_real2 (void)
- double genrand_real3 (void)
- double genrand_res53 (void)

### 9.144.1 Detailed Description

mt.h: Mersenne Twister header file

Jason R. Blevins jrblevin@sdf.lonestar.org Durham, March 7, 2007 Mersenne Twister.

M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Trans. on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3-30 (1998).

http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html.

Definition at line 21 of file mt.h.

## 9.144.2 Constructor & Destructor Documentation

### 9.144.2.1 MersenneTwister()

```
MersenneTwister::MersenneTwister (
            void )
```

C++ Mersenne Twister wrapper class written by Jason R. Blevins jrblevin@sdf.lonestar.org on July 24, 2006. Based on the original MT19937 C code by Takuji Nishimura and Makoto Matsumoto. Constructor

Definition at line 58 of file mt.cpp.

### 9.144.2.2 ~MersenneTwister()

```
MersenneTwister::~MersenneTwister (
            void )
```

Destructor

Definition at line 71 of file mt.cpp.

## 9.144.3 Member Function Documentation

### 9.144.3.1 genrand_int31()

```
long MersenneTwister::genrand_int31 (
            void )
```

Generates a random integer on [0,0x7fffffff].

**Returns**

a random integer on [0,0x7fffffff]

Definition at line 197 of file mt.cpp.

### 9.144.3.2 genrand_int32()

```
unsigned long MersenneTwister::genrand_int32 (
            void )
```

Generates a random number on [0,0xffffffff]-interval

**Returns**

random number on [0, 0xffffffff]

Definition at line 155 of file mt.cpp.

#### 9.144.3.3 genrand_real1()

```
double MersenneTwister::genrand_real1 (
            void )
```

Generates a random real number on [0,1].

**Returns**

a random real number on [0,1]

Definition at line 207 of file mt.cpp.

#### 9.144.3.4 genrand_real2()

```
double MersenneTwister::genrand_real2 (
            void )
```

Generates a random real number on [0,1).

**Returns**

a random real number on [0,1)

Definition at line 218 of file mt.cpp.

#### 9.144.3.5 genrand_real3()

```
double MersenneTwister::genrand_real3 (
            void )
```

Generates a random real number on (0,1).

**Returns**

a random real number on (0,1)

Definition at line 229 of file mt.cpp.

#### 9.144.3.6 genrand_res53()

```
double MersenneTwister::genrand_res53 (
            void )
```

Generates a random real number on [0,1) with 53-bit precision.

**Returns**

a random 53-bit real number on [0,1)

Definition at line 240 of file mt.cpp.

#### 9.144.3.7 init_by_array()

```
void MersenneTwister::init_by_array (
            unsigned long * init_key,
            int key_length)
```

Seed the Mersenne Twister using an array.

**Parameters**

| | |
|---|---|
| *init_key* | an array for initializing keys |

| | |
|---|---|
| *key_length* | the length of *init_key* |

Definition at line 112 of file mt.cpp.

#### 9.144.3.8 init_genrand()

```
void MersenneTwister::init_genrand (
            unsigned long s)
```

Initializes the Mersenne Twister with a seed.

**Parameters**

| | |
|---|---|
| *s* | seed |

Definition at line 87 of file mt.cpp.

#### 9.144.3.9 print()

```
void MersenneTwister::print (
            void )
```

Print interesting information about the Mersenne Twister.

Definition at line 251 of file mt.cpp.

#### 9.144.3.10 random()

```
double MersenneTwister::random (
            void )  [inline]
```

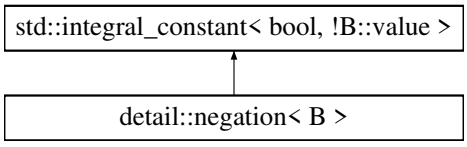
Definition at line 27 of file mt.h.

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

- include/External/mt.h
- src/External/mt.cpp

## 9.145 detail::negation< B > Struct Template Reference

Inheritance diagram for detail::negation< B >:



### 9.145.1 Detailed Description

**template<class B>**
**struct detail::negation< B >**

Definition at line 3824 of file json.hpp.

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

- include/External/json.hpp

## 9.146 detail::nonesuch Struct Reference

**Public Member Functions**

- **nonesuch** (nonesuch const &)=delete
- **nonesuch** (nonesuch const &&)=delete
- void **operator=** (nonesuch const &)=delete
- void **operator=** (nonesuch &&)=delete

### 9.146.1 Detailed Description

Definition at line 277 of file json.hpp.

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

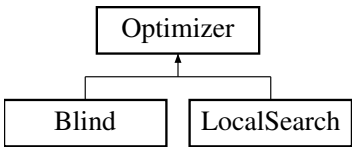
- include/External/json.hpp

## 9.147 Optimizer Class Reference

Abstract base class for all optimization algorithms.

`#include <Optimizer.h>`

Inheritance diagram for Optimizer:



### Public Member Functions

- Optimizer (SolutionBuilder &solutionBuilder, Problem &problem, int maxIterations)

  *Constructs an Optimizer.*
- virtual ~**Optimizer** ()=default

  *Virtual destructor for safe polymorphic deletion.*
- virtual double optimize ()=0

  *Runs the optimization algorithm.*

#### Accessors

- double getBestFitness ()
- std::vector< double > & getBestSolution ()
- std::vector< double > & getBestFitnesses ()
- std::vector< std::vector< double > > & getSolutions ()
- int getMaxIterations ()
- Problem & getProblem ()
- SolutionBuilder & getSolutionBuilder ()

### Protected Attributes

- Problem & problem

  *Optimization problem definition.*
- SolutionBuilder & solutionBuilder

  *Solution generation utility.*
- int maxIterations

  *Maximum number of iterations.*
- std::vector< double > bestSolution

  *Best solution found.*
- std::vector< double > bestFitnesses

  *Best fitness value so far per iteration.*
- std::vector< std::vector< double > > solutions

  *All generated solutions.*

## 9.147.1 Detailed Description

Abstract base class for all optimization algorithms.

The Optimizer class defines a common interface and shared state for optimization algorithms operating on benchmark problems. Derived classes must implement the optimize() method. Results for each run are stored in fields and accessed through getter methods

Definition at line 28 of file Optimizer.h.

## 9.147.2 Constructor & Destructor Documentation

### 9.147.2.1 Optimizer()

```
Optimizer::Optimizer (
            SolutionBuilder & solutionBuilder,
            Problem & problem,
            int maxIterations)  [inline]
```

Constructs an Optimizer.

**Parameters**

| | |
|---|---|
| *solutionBuilder* | Reference to the solution generator. |

| | |
|---|---|
| *problem* | Reference to the optimization problem. |
| *maxIterations* | Maximum number of iterations allowed. |

Definition at line 37 of file Optimizer.h.

### 9.147.3 Member Function Documentation

#### 9.147.3.1 getBestFitness()

```
double Optimizer::getBestFitness ()  [inline]
```

**Returns**

The final best fitness value

Definition at line 58 of file Optimizer.h.

#### 9.147.3.2 getBestFitnesses()

```
std::vector< double > & Optimizer::getBestFitnesses ()  [inline]
```

**Returns**

Best fitness value so far per iteration

Definition at line 64 of file Optimizer.h.

#### 9.147.3.3 getBestSolution()

```
std::vector< double > & Optimizer::getBestSolution ()  [inline]
```

**Returns**

Reference to the best solution vector

Definition at line 61 of file Optimizer.h.

#### 9.147.3.4 getMaxIterations()

```
int Optimizer::getMaxIterations ()  [inline]
```

**Returns**

Maximum number of iterations

Definition at line 70 of file Optimizer.h.

#### 9.147.3.5 getProblem()

```
Problem & Optimizer::getProblem ()  [inline]
```

**Returns**

Reference to the optimization problem

Definition at line 73 of file Optimizer.h.

#### 9.147.3.6 getSolutionBuilder()

```
SolutionBuilder & Optimizer::getSolutionBuilder ()  [inline]
```

**Returns**

Reference to the solution builder

Definition at line 76 of file Optimizer.h.

#### 9.147.3.7 getSolutions()

```
std::vector< std::vector< double > > & Optimizer::getSolutions ()  [inline]
```

**Returns**

All solutions evaluated during optimization

Definition at line 67 of file Optimizer.h.

#### 9.147.3.8 optimize()

```
virtual double Optimizer::optimize ()  [pure virtual]
```

Runs the optimization algorithm.

**Returns**

Execution time of the algorithm

Implemented in Blind, and LocalSearch.

### 9.147.4 Member Data Documentation

#### 9.147.4.1 bestFitnesses

```
std::vector<double> Optimizer::bestFitnesses  [protected]
```

Best fitness value so far per iteration.

Definition at line 95 of file Optimizer.h.

#### 9.147.4.2 bestSolution

```
std::vector<double> Optimizer::bestSolution  [protected]
```

Best solution found.

Definition at line 92 of file Optimizer.h.

#### 9.147.4.3 maxIterations

```
int Optimizer::maxIterations  [protected]
```

Maximum number of iterations.

Definition at line 89 of file Optimizer.h.

#### 9.147.4.4 problem

```
Problem& Optimizer::problem  [protected]
```

Optimization problem definition.

Definition at line 83 of file Optimizer.h.

#### 9.147.4.5 solutionBuilder

```
SolutionBuilder& Optimizer::solutionBuilder  [protected]
```

Solution generation utility.

Definition at line 86 of file Optimizer.h.

#### 9.147.4.6 solutions

```
std::vector<std::vector<double> > Optimizer::solutions  [protected]
```

All generated solutions.

Definition at line 98 of file Optimizer.h.

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

- include/Optimizer/Optimizer.h

## 9.148 OptimizerFactory Class Reference

Factory class for creating optimizer instances.

```
#include <OptimizerFactory.h>
```

**Static Public Member Functions**

- static std::unique_ptr< Optimizer > initOptimizer (Problem &problem, ExperimentConfig &config, SolutionBuilder &builder)

  *Initializes an optimizer based on configuration settings.*

### 9.148.1 Detailed Description

Factory class for creating optimizer instances.

The OptimizerFactory encapsulates the logic for selecting and constructing the appropriate optimization algorithm based on experimental configuration parameters.

Definition at line 28 of file OptimizerFactory.h.

### 9.148.2 Member Function Documentation

#### 9.148.2.1 initOptimizer()

```
std::unique_ptr< Optimizer > OptimizerFactory::initOptimizer (
            Problem & problem,
            ExperimentConfig & config,
            SolutionBuilder & builder)  [inline], [static]
```

Initializes an optimizer based on configuration settings.

**Parameters**

| | |
|---|---|
| *problem* | Reference to the optimization problem. |
| *config* | Experiment configuration specifying optimizer type and parameters. |
| *builder* | Reference to the solution builder. |

**Returns**

A unique pointer to the initialized Optimizer instance, or nullptr if the optimizer type is unsupported.

Definition at line 40 of file OptimizerFactory.h.

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

- include/Optimizer/OptimizerFactory.h

## 9.149 ordered_map< Key, T, IgnoredLess, Allocator > Struct Template Reference

a minimal map-like container that preserves insertion order

### 9.149.1 Detailed Description

**template<class Key, class T, class IgnoredLess, class Allocator>**
**struct ordered_map< Key, T, IgnoredLess, Allocator >**

a minimal map-like container that preserves insertion order

**See also**

https://json.nlohmann.me/api/ordered_map/

Definition at line 3560 of file json.hpp.

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

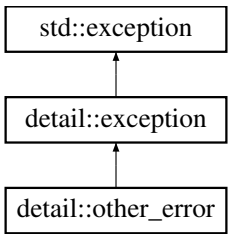
- include/External/json.hpp

## 9.150 detail::other_error Class Reference

exception indicating other library errors

`#include <json.hpp>`

Inheritance diagram for detail::other_error:




#### Static Public Member Functions

- template<typename BasicJsonContext, enable_if_t< is_basic_json_context< BasicJsonContext >::value, int > = 0>
  static other_error create (int id_, const std::string &what_arg, BasicJsonContext context)

#### Additional Inherited Members

### Public Member Functions inherited from detail::exception

- const char * what () const noexcept override
  *returns the explanatory string*

### Public Attributes inherited from detail::exception

- const int id

  *the id of the exception*

### Protected Member Functions inherited from detail::exception

- exception (int id_, const char *what_arg)

### Static Protected Member Functions inherited from detail::exception

- static std::string name (const std::string &ename, int id_)
- static std::string diagnostics (std::nullptr_t)
- template<typename BasicJsonType>
  static std::string diagnostics (const BasicJsonType *leaf_element)

## 9.150.1 Detailed Description

exception indicating other library errors

**See also**

https://json.nlohmann.me/api/basic_json/other_error/

Definition at line 4779 of file json.hpp.

## 9.150.2 Member Function Documentation

### 9.150.2.1 create()

```
template<typename BasicJsonContext, enable_if_t< is_basic_json_context< BasicJsonContext
>::value, int > = 0>
other_error detail::other_error::create (
            int id_,
            const std::string & what_arg,
            BasicJsonContext context)  [inline], [static]
```

Definition at line 4783 of file json.hpp.

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

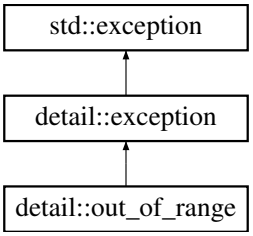
- include/External/json.hpp

# 9.151 detail::out_of_range Class Reference

exception indicating access out of the defined range

`#include <json.hpp>`

Inheritance diagram for detail::out_of_range:




### Static Public Member Functions

- template<typename BasicJsonContext, enable_if_t< is_basic_json_context< BasicJsonContext >::value, int > = 0>
  static out_of_range create (int id_, const std::string &what_arg, BasicJsonContext context)

### Additional Inherited Members

## Public Member Functions inherited from detail::exception

- const char * what () const noexcept override
  *returns the explanatory string*

## Public Attributes inherited from detail::exception

- const int id
  *the id of the exception*

## Protected Member Functions inherited from detail::exception

- exception (int id_, const char *what_arg)

## Static Protected Member Functions inherited from detail::exception

- static std::string name (const std::string &ename, int id_)
- static std::string diagnostics (std::nullptr_t)
- template<typename BasicJsonType>
  static std::string diagnostics (const BasicJsonType *leaf_element)

### 9.151.1 Detailed Description

exception indicating access out of the defined range

**See also**

https://json.nlohmann.me/api/basic_json/out_of_range/

Definition at line 4762 of file json.hpp.

### 9.151.2 Member Function Documentation

#### 9.151.2.1 create()

```
template<typename BasicJsonContext, enable_if_t< is_basic_json_context< BasicJsonContext
>::value, int > = 0>
out_of_range detail::out_of_range::create (
            int id_,
            const std::string & what_arg,
            BasicJsonContext context)  [inline], [static]
```

Definition at line 4766 of file json.hpp.

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

- include/External/json.hpp

## 9.152 detail::output_adapter< CharType, StringType > Class Template Reference

**Public Member Functions**

- template<typename AllocatorType = std::allocator<CharType>>
  output_adapter (std::vector< CharType, AllocatorType > &vec)
- output_adapter (std::basic_ostream< CharType > &s)
- output_adapter (StringType &s)
- operator output_adapter_t< CharType > ()

### 9.152.1 Detailed Description

**template<typename CharType, typename StringType = std::basic_string<CharType>>**
**class detail::output_adapter< CharType, StringType >**

Definition at line 15839 of file json.hpp.

### 9.152.2 Constructor & Destructor Documentation

#### 9.152.2.1 output_adapter() [1/3]

```
template<typename CharType, typename StringType = std::basic_string<CharType>>
template<typename AllocatorType = std::allocator<CharType>>
detail::output_adapter< CharType, StringType >::output_adapter (
            std::vector< CharType, AllocatorType > & vec)  [inline]
```

Definition at line 15843 of file json.hpp.

#### 9.152.2.2 output_adapter() [2/3]

```
template<typename CharType, typename StringType = std::basic_string<CharType>>
detail::output_adapter< CharType, StringType >::output_adapter (
            std::basic_ostream< CharType > & s)  [inline]
```

Definition at line 15847 of file json.hpp.

#### 9.152.2.3 output_adapter() [3/3]

```
template<typename CharType, typename StringType = std::basic_string<CharType>>
detail::output_adapter< CharType, StringType >::output_adapter (
            StringType & s)  [inline]
```

Definition at line 15851 of file json.hpp.

### 9.152.3 Member Function Documentation

#### 9.152.3.1 operator output_adapter_t< CharType >()

```
template<typename CharType, typename StringType = std::basic_string<CharType>>
detail::output_adapter< CharType, StringType >::operator output_adapter_t< CharType > ()
[inline]
```

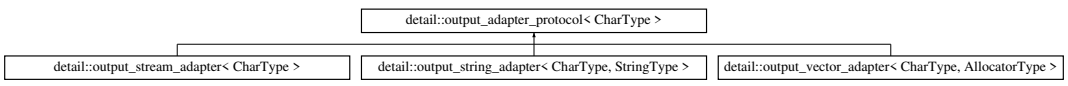
Definition at line 15854 of file json.hpp.

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

- include/External/json.hpp

## 9.153 detail::output_adapter_protocol< CharType > Struct Template Reference

Inheritance diagram for 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
- **output_adapter_protocol** (const output_adapter_protocol &)=default
- **output_adapter_protocol** (output_adapter_protocol &&) noexcept=default
- output_adapter_protocol & **operator=** (const output_adapter_protocol &)=default
- output_adapter_protocol & **operator=** (output_adapter_protocol &&) noexcept=default

### 9.153.1 Detailed Description

**template**<**typename CharType**>
**struct detail::output_adapter_protocol**< **CharType** >

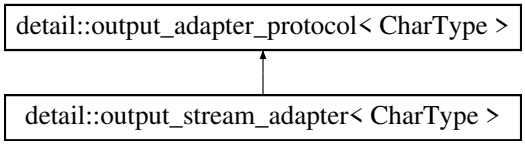
Definition at line 15747 of file json.hpp.

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

- include/External/json.hpp

## 9.154 detail::output_stream_adapter< CharType > Class Template Reference

Inheritance diagram for detail::output_stream_adapter< CharType >:




**Public Member Functions**

- output_stream_adapter (std::basic_ostream< CharType > &s) noexcept
- void write_character (CharType c) override
- void write_characters (const CharType ∗s, std::size_t length) override

### Public Member Functions inherited from detail::output_adapter_protocol< CharType >

- **output_adapter_protocol** (const output_adapter_protocol &)=default
- **output_adapter_protocol** (output_adapter_protocol &&) noexcept=default
- output_adapter_protocol & **operator=** (const output_adapter_protocol &)=default
- output_adapter_protocol & **operator=** (output_adapter_protocol &&) noexcept=default

## 9.154.1 Detailed Description

**template<typename CharType>**
**class detail::output_stream_adapter< CharType >**

Definition at line 15791 of file json.hpp.

## 9.154.2 Constructor & Destructor Documentation

### 9.154.2.1 output_stream_adapter()

```
template<typename CharType>
detail::output_stream_adapter< CharType >::output_stream_adapter (
            std::basic_ostream< CharType > & s)  [inline], [explicit], [noexcept]
```

Definition at line 15794 of file json.hpp.

## 9.154.3 Member Function Documentation

### 9.154.3.1 write_character()

```
template<typename CharType>
void detail::output_stream_adapter< CharType >::write_character (
            CharType c)  [inline], [override], [virtual]
```

Implements detail::output_adapter_protocol< CharType >.

Definition at line 15798 of file json.hpp.

### 9.154.3.2 write_characters()

```
template<typename CharType>
void detail::output_stream_adapter< CharType >::write_characters (
            const CharType * s,
            std::size_t length)  [inline], [override], [virtual]
```

Implements detail::output_adapter_protocol< CharType >.

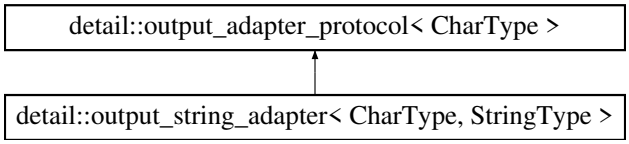
Definition at line 15804 of file json.hpp.

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

- include/External/json.hpp

# 9.155 detail::output_string_adapter< CharType, StringType > Class Template Reference

Inheritance diagram for detail::output_string_adapter< CharType, StringType >:




### Public Member Functions

- output_string_adapter (StringType &s) noexcept
- void write_character (CharType c) override
- void write_characters (const CharType *s, std::size_t length) override

## Public Member Functions inherited from detail::output_adapter_protocol< CharType >

- **output_adapter_protocol** (const output_adapter_protocol &)=default
- **output_adapter_protocol** (output_adapter_protocol &&) noexcept=default
- output_adapter_protocol & **operator=** (const output_adapter_protocol &)=default
- output_adapter_protocol & **operator=** (output_adapter_protocol &&) noexcept=default

## 9.155.1 Detailed Description

**template<typename CharType, typename StringType = std::basic_string<CharType>>**
**class detail::output_string_adapter< CharType, StringType >**

Definition at line 15816 of file json.hpp.

## 9.155.2 Constructor & Destructor Documentation

### 9.155.2.1 output_string_adapter()

```
template<typename CharType, typename StringType = std::basic_string<CharType>>
detail::output_string_adapter< CharType, StringType >::output_string_adapter (
            StringType & s)  [inline], [explicit], [noexcept]
```

Definition at line 15819 of file json.hpp.

## 9.155.3 Member Function Documentation

### 9.155.3.1 write_character()

```
template<typename CharType, typename StringType = std::basic_string<CharType>>
void detail::output_string_adapter< CharType, StringType >::write_character (
            CharType c)  [inline], [override], [virtual]
```

Implements detail::output_adapter_protocol< CharType >.

Definition at line 15823 of file json.hpp.

#### 9.155.3.2 write_characters()

```
template<typename CharType, typename StringType = std::basic_string<CharType>>
void detail::output_string_adapter< CharType, StringType >::write_characters (
            const CharType * s,
            std::size_t length)  [inline], [override], [virtual]
```

Implements detail::output_adapter_protocol< CharType >.

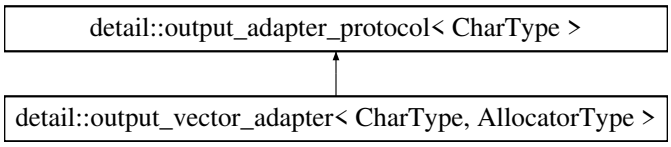
Definition at line 15829 of file json.hpp.

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

- include/External/json.hpp

## 9.156 detail::output_vector_adapter< CharType, AllocatorType > Class Template Reference

Inheritance diagram for detail::output_vector_adapter< CharType, AllocatorType >:




### Public Member Functions

- output_vector_adapter (std::vector< CharType, AllocatorType > &vec) noexcept
- void write_character (CharType c) override
- void write_characters (const CharType *s, std::size_t length) override

### Public Member Functions inherited from detail::output_adapter_protocol< CharType >

- **output_adapter_protocol** (const output_adapter_protocol &)=default
- **output_adapter_protocol** (output_adapter_protocol &&) noexcept=default
- output_adapter_protocol & **operator=** (const output_adapter_protocol &)=default
- output_adapter_protocol & **operator=** (output_adapter_protocol &&) noexcept=default

### 9.156.1 Detailed Description

**template<typename CharType, typename AllocatorType = std::allocator<CharType>>**
**class detail::output_vector_adapter< CharType, AllocatorType >**

Definition at line 15766 of file json.hpp.

### 9.156.2 Constructor & Destructor Documentation

#### 9.156.2.1 output_vector_adapter()

```
template<typename CharType, typename AllocatorType = std::allocator<CharType>>
detail::output_vector_adapter< CharType, AllocatorType >::output_vector_adapter (
            std::vector< CharType, AllocatorType > & vec)  [inline], [explicit], [noexcept]
```

Definition at line 15769 of file json.hpp.

### 9.156.3 Member Function Documentation

#### 9.156.3.1 write_character()

```
template<typename CharType, typename AllocatorType = std::allocator<CharType>>
void detail::output_vector_adapter< CharType, AllocatorType >::write_character (
            CharType c)  [inline], [override], [virtual]
```

Implements detail::output_adapter_protocol< CharType >.

Definition at line 15773 of file json.hpp.

#### 9.156.3.2 write_characters()

```
template<typename CharType, typename AllocatorType = std::allocator<CharType>>
void detail::output_vector_adapter< CharType, AllocatorType >::write_characters (
            const CharType * s,
            std::size_t length)  [inline], [override], [virtual]
```

Implements detail::output_adapter_protocol< CharType >.

Definition at line 15779 of file json.hpp.

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

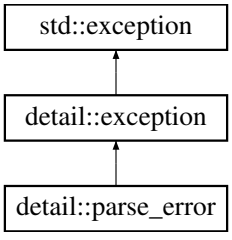
- include/External/json.hpp

## 9.157 detail::parse_error Class Reference

exception indicating a parse error

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

Inheritance diagram for detail::parse_error:



### Static Public Member Functions

- template<typename BasicJsonContext, enable_if_t< is_basic_json_context< BasicJsonContext >::value, int > = 0>
  static parse_error create (int id_, const position_t &pos, const std::string &what_arg, BasicJsonContext context)

  *create a parse error exception*
- template<typename BasicJsonContext, enable_if_t< is_basic_json_context< BasicJsonContext >::value, int > = 0>
  static parse_error create (int id_, std::size_t byte_, const std::string &what_arg, BasicJsonContext context)

### Public Attributes

- const std::size_t byte

  *byte index of the parse error*

## Public Attributes inherited from detail::exception

- const int id

  *the id of the exception*

### Additional Inherited Members

## Public Member Functions inherited from detail::exception

- const char * what () const noexcept override

  *returns the explanatory string*

## Protected Member Functions inherited from detail::exception

- exception (int id_, const char *what_arg)

## Static Protected Member Functions inherited from detail::exception

- static std::string name (const std::string &ename, int id_)
- static std::string diagnostics (std::nullptr_t)
- template<typename BasicJsonType>
  static std::string diagnostics (const BasicJsonType *leaf_element)

### 9.157.1 Detailed Description

exception indicating a parse error

**See also**

https://json.nlohmann.me/api/basic_json/parse_error/

Definition at line 4674 of file json.hpp.

## 9.157.2 Member Function Documentation

### 9.157.2.1 create() [1/2]

```
template<typename BasicJsonContext, enable_if_t< is_basic_json_context< BasicJsonContext
>::value, int > = 0>
parse_error detail::parse_error::create (
            int id_,
            const position_t & pos,
            const std::string & what_arg,
            BasicJsonContext context)  [inline], [static]
```

create a parse error exception

**Parameters**

| | | |
|---|---|---|
| in | *id_* | the id of the exception |
| in | *pos* | the position where the error occurred (or with chars_read_total=0 if the position cannot be determined) |
| in | *what_arg* | the explanatory string |

**Returns**

parse_error object

Definition at line 4687 of file json.hpp.

### 9.157.2.2 create() [2/2]

```
template<typename BasicJsonContext, enable_if_t< is_basic_json_context< BasicJsonContext
>::value, int > = 0>
parse_error detail::parse_error::create (
            int id_,
            std::size_t byte_,
            const std::string & what_arg,
            BasicJsonContext context)  [inline], [static]
```

Definition at line 4695 of file json.hpp.

## 9.157.3 Member Data Documentation

### 9.157.3.1 byte

```
const std::size_t 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).

Definition at line 4712 of file json.hpp.

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

- include/External/json.hpp

# 9.158 detail::parser< BasicJsonType, InputAdapterType > Class Template Reference

syntax analysis

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

### Public Member Functions

- ∗a parser reading from an input adapter parser (InputAdapterType &&adapter, parser_callback_t< BasicJsonType > cb=nullptr, const bool allow_exceptions_=true, const bool ignore_comments=false, const bool ignore_trailing_commas_=false)
- void parse (const bool strict, BasicJsonType &result)

  *public parser interface*
- bool accept (const bool strict=true)

  *public accept interface*
- template<typename SAX>
  bool sax_parse (SAX ∗sax, const bool strict=true)

## 9.158.1 Detailed Description

**template<typename BasicJsonType, typename InputAdapterType>**
**class detail::parser< BasicJsonType, InputAdapterType >**

syntax analysis

This class implements a recursive descent parser.

Definition at line 13003 of file json.hpp.

## 9.158.2 Constructor & Destructor Documentation

### 9.158.2.1 parser()

```
template<typename BasicJsonType, typename InputAdapterType>
*a parser reading from an input adapter detail::parser< BasicJsonType, InputAdapterType >::↩
parser (
            InputAdapterType && adapter,
            parser_callback_t< BasicJsonType > cb = nullptr,
            const bool allow_exceptions_ = true,
            const bool ignore_comments = false,
            const bool ignore_trailing_commas_ = false)  [inline], [explicit]
```

Definition at line 13014 of file json.hpp.

## 9.158.3 Member Function Documentation

### 9.158.3.1 accept()

```
template<typename BasicJsonType, typename InputAdapterType>
bool detail::parser< BasicJsonType, InputAdapterType >::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

Definition at line 13098 of file json.hpp.

### 9.158.3.2 parse()

```
template<typename BasicJsonType, typename InputAdapterType>
void detail::parser< BasicJsonType, InputAdapterType >::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 |

Definition at line 13038 of file json.hpp.

### 9.158.3.3 sax_parse()

```
template<typename BasicJsonType, typename InputAdapterType>
template<typename SAX>
bool detail::parser< BasicJsonType, InputAdapterType >::sax_parse (
            SAX * sax,
            const bool strict = true)  [inline]
```

Definition at line 13106 of file json.hpp.

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

- include/External/json.hpp

# 9.159 Population Class Reference

**Public Member Functions**

- Population (int populationSize, int dimension)
- int initialize (double lower, double upper, int seed)
- std::vector< double > evaluate (const Problem &problem)
- int generateNeighbors (const std::vector< double > &center, double delta, double lower, double upper, int seed)
- const std::vector< std::vector< double > > & getSolutions () const

## 9.159.1 Detailed Description

Definition at line 7 of file Population.h.

## 9.159.2 Constructor & Destructor Documentation

### 9.159.2.1 Population()

```
Population::Population (
            int populationSize,
            int dimension)
```

Definition at line 8 of file Population.cpp.

## 9.159.3 Member Function Documentation

### 9.159.3.1 evaluate()

```
std::vector< double > Population::evaluate (
            const Problem & problem)
```

Definition at line 59 of file Population.cpp.

### 9.159.3.2 generateNeighbors()

```
int Population::generateNeighbors (
            const std::vector< double > & center,
            double delta,
            double lower,
            double upper,
            int seed)
```

Definition at line 31 of file Population.cpp.

#### 9.159.3.3 getSolutions()

```
const std::vector< std::vector< double > > & Population::getSolutions () const
```

Definition at line 69 of file Population.cpp.

#### 9.159.3.4 initialize()

```
int Population::initialize (
            double lower,
            double upper,
            int seed)
```

Definition at line 13 of file Population.cpp.

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

- include/Population.h
- src/Population.cpp

## 9.160 detail::position_t Struct Reference

struct to capture the start position of the current token

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

### Public Member Functions

- constexpr operator size_t () const
  *conversion to size_t to preserve SAX interface*

### Public Attributes

- std::size_t chars_read_total = 0
  *the total number of characters read*
- std::size_t chars_read_current_line = 0
  *the number of characters read in the current line*
- std::size_t lines_read = 0
  *the number of lines read*

### 9.160.1 Detailed Description

struct to capture the start position of the current token

Definition at line 3166 of file json.hpp.

## 9.160.2 Member Function Documentation

### 9.160.2.1 operator size_t()

```
detail::position_t::operator size_t () const  [inline], [constexpr]
```

conversion to size_t to preserve SAX interface

Definition at line 3176 of file json.hpp.

## 9.160.3 Member Data Documentation

### 9.160.3.1 chars_read_current_line

```
std::size_t detail::position_t::chars_read_current_line = 0
```

the number of characters read in the current line

Definition at line 3171 of file json.hpp.

### 9.160.3.2 chars_read_total

```
std::size_t detail::position_t::chars_read_total = 0
```

the total number of characters read

Definition at line 3169 of file json.hpp.

### 9.160.3.3 lines_read

```
std::size_t detail::position_t::lines_read = 0
```

the number of lines read

Definition at line 3173 of file json.hpp.

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

- include/External/json.hpp

# 9.161 detail::primitive_iterator_t Class Reference

### Public Member Functions

- constexpr difference_type get_value () const noexcept
- ∗set iterator to a defined beginning void set_begin () noexcept
- ∗set iterator to a defined past the end void set_end () noexcept
- ∗return whether the iterator can be dereferenced constexpr bool is_begin () const noexcept
- ∗return whether the iterator is at end constexpr bool is_end () const noexcept
- primitive_iterator_t operator+ (difference_type n) noexcept
- primitive_iterator_t & operator++ () noexcept
- primitive_iterator_t operator++ (int) &noexcept
- primitive_iterator_t & operator-- () noexcept
- primitive_iterator_t operator-- (int) &noexcept
- primitive_iterator_t & operator+= (difference_type n) noexcept
- primitive_iterator_t & operator-= (difference_type n) noexcept

**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

## 9.161.1 Detailed Description

Definition at line 13524 of file json.hpp.

## 9.161.2 Member Function Documentation

### 9.161.2.1 get_value()

```
difference_type detail::primitive_iterator_t::get_value () const  [inline], [constexpr], [noexcept]
```

Definition at line 13536 of file json.hpp.

### 9.161.2.2 is_begin()

```
*return whether the iterator can be dereferenced constexpr bool detail::primitive_iterator_↩
t::is_begin () const  [inline], [constexpr], [noexcept]
```

Definition at line 13554 of file json.hpp.

### 9.161.2.3 is_end()

```
*return whether the iterator is at end constexpr bool detail::primitive_iterator_t::is_end ()
const  [inline], [constexpr], [noexcept]
```

Definition at line 13560 of file json.hpp.

### 9.161.2.4 operator+()

```
primitive_iterator_t detail::primitive_iterator_t::operator+ (
            difference_type n)  [inline], [noexcept]
```

Definition at line 13575 of file json.hpp.

### 9.161.2.5 operator++() [1/2]

```
primitive_iterator_t & detail::primitive_iterator_t::operator++ ()  [inline], [noexcept]
```

Definition at line 13587 of file json.hpp.

#### 9.161.2.6 operator++() [2/2]

```
primitive_iterator_t detail::primitive_iterator_t::operator++ (
            int ) &  [inline], [noexcept]
```

Definition at line 13593 of file json.hpp.

#### 9.161.2.7 operator+=()

```
primitive_iterator_t & detail::primitive_iterator_t::operator+= (
            difference_type n)  [inline], [noexcept]
```

Definition at line 13613 of file json.hpp.

#### 9.161.2.8 operator--() [1/2]

```
primitive_iterator_t & detail::primitive_iterator_t::operator-- ()  [inline], [noexcept]
```

Definition at line 13600 of file json.hpp.

#### 9.161.2.9 operator--() [2/2]

```
primitive_iterator_t detail::primitive_iterator_t::operator-- (
            int ) &  [inline], [noexcept]
```

Definition at line 13606 of file json.hpp.

#### 9.161.2.10 operator-=()

```
primitive_iterator_t & detail::primitive_iterator_t::operator-= (
            difference_type n)  [inline], [noexcept]
```

Definition at line 13619 of file json.hpp.

#### 9.161.2.11 set_begin()

```
*set iterator to a defined beginning void detail::primitive_iterator_t::set_begin ()  [inline],
[noexcept]
```

Definition at line 13542 of file json.hpp.

#### 9.161.2.12 set_end()

```
*set iterator to a defined past the end void detail::primitive_iterator_t::set_end ()  [inline],
[noexcept]
```

Definition at line 13548 of file json.hpp.

### 9.161.3 Friends And Related Symbol Documentation

#### 9.161.3.1 operator-

```
difference_type operator- (
            primitive_iterator_t lhs,
            primitive_iterator_t rhs)  [friend]
```

Definition at line 13582 of file json.hpp.

#### 9.161.3.2 operator<

```
bool operator< (
            primitive_iterator_t lhs,
            primitive_iterator_t rhs)  [friend]
```

Definition at line 13570 of file json.hpp.

#### 9.161.3.3 operator==

```
bool operator== (
            primitive_iterator_t lhs,
            primitive_iterator_t rhs)  [friend]
```

Definition at line 13565 of file json.hpp.

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

- include/External/json.hpp

## 9.162 detail::priority_tag< N > Struct Template Reference

### 9.162.1 Detailed Description

**template<unsigned N>**
**struct detail::priority_tag< N >**

Definition at line 3337 of file json.hpp.

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

- include/External/json.hpp

## 9.163 detail::priority_tag< 0 > Struct Reference

### 9.163.1 Detailed Description

Definition at line 3338 of file json.hpp.

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

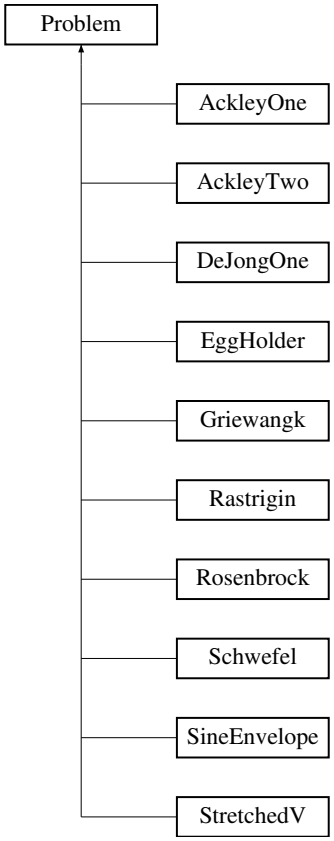
- include/External/json.hpp

## 9.164 Problem Class Reference

Abstract base class for all optimization benchmark problems.

`#include <Problem.h>`

Inheritance diagram for Problem:



**Public Member Functions**

- Problem (double lb, double ub, const std::string_view n)

  *Constructs a Problem instance.*
- virtual ~**Problem** ()=default

  *Virtual destructor for safe polymorphic cleanup.*
- virtual double evaluate (const std::vector< double > &x) const =0

  *Evaluates the fitness of a candidate solution.*

**Accessors**

- double getLowerBound () const
- double getUpperBound () const
- const std::string getName () const

**Protected Attributes**

- const double lowerBound

  *Lower bound of the search space.*
- const double upperBound

  *Upper bound of the search space.*
- const std::string name

  *Name of the benchmark function.*

## 9.164.1 Detailed Description

Abstract base class for all optimization benchmark problems.

Provides a common interface for evaluating fitness and retrieving search space boundaries.

Definition at line 21 of file Problem.h.

## 9.164.2 Constructor & Destructor Documentation

### 9.164.2.1 Problem()

```
Problem::Problem (
            double lb,
            double ub,
            const std::string_view n)  [inline]
```

Constructs a Problem instance.

- **Parameters**

| | |
|---|---|
| *lb* | Lower bound for the solution space. |

| | |
|---|---|
| *ub* | Upper bound for the solution space. |
| *n* | Name of the optimization function. |

Definition at line 34 of file Problem.h.

## 9.164.3 Member Function Documentation

### 9.164.3.1 evaluate()

```
virtual double Problem::evaluate (
            const std::vector< double > & x) const  [pure virtual]
```

Evaluates the fitness of a candidate solution.

- This is a pure virtual function that must be implemented by specific benchmark functions (e.g., Ackley, Rosenbrock).

- **Parameters**

| | |
|---|---|
| *x* | The solution vector to evaluate. |

**Returns**

The scalar fitness value (cost).

Implemented in AckleyOne, AckleyTwo, DeJongOne, EggHolder, Griewangk, Rastrigin, Rosenbrock, Schwefel, SineEnvelope, and StretchedV.

### 9.164.3.2 getLowerBound()

```
double Problem::getLowerBound () const  [inline]
```

**Returns**

The lower boundary of the search space.

Definition at line 55 of file Problem.h.

### 9.164.3.3 getName()

```
const std::string Problem::getName () const  [inline]
```

**Returns**

The name of the benchmark function.

Definition at line 61 of file Problem.h.

#### 9.164.3.4 getUpperBound()

```
double Problem::getUpperBound () const  [inline]
```

**Returns**

The upper boundary of the search space.

Definition at line 58 of file Problem.h.

### 9.164.4 Member Data Documentation

#### 9.164.4.1 lowerBound

```
const double Problem::lowerBound  [protected]
```

Lower bound of the search space.

Definition at line 23 of file Problem.h.

#### 9.164.4.2 name

```
const std::string Problem::name  [protected]
```

Name of the benchmark function.

Definition at line 25 of file Problem.h.

#### 9.164.4.3 upperBound

```
const double Problem::upperBound  [protected]
```

Upper bound of the search space.

Definition at line 24 of file Problem.h.

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

- include/Problem/Problem.h

## 9.165 ProblemFactory Class Reference

Utility to create problem instances dynamically.

```
#include <ProblemFactory.h>
```

**Static Public Member Functions**

- static std::unique_ptr< Problem > create (int id)

  *Factory method to instantiate a specific benchmark problem.*

### 9.165.1 Detailed Description

Utility to create problem instances dynamically.

Definition at line 20 of file ProblemFactory.h.

### 9.165.2 Member Function Documentation

#### 9.165.2.1 create()

```
std::unique_ptr< Problem > ProblemFactory::create (
            int id)  [static]
```

Factory method to instantiate a specific benchmark problem.

- **Parameters**

| | |
|---|---|
| *id* | Integer ID corresponding to the desired problem (e.g., 1 for Ackley, 2 for DeJong). |

**Returns**

std::unique_ptr<Problem> A pointer to the newly created problem instance.

**Exceptions**

| | |
|---|---|
| *std::invalid_argument* | if the ID does not match a known problem. |

Definition at line 77 of file ProblemFactory.cpp.

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

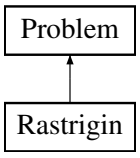
- include/ProblemFactory.h
- src/ProblemFactory.cpp

## 9.166 Rastrigin Class Reference

Implements the Rastrigin benchmark function.

```
#include <Rastrigin.h>
```

Inheritance diagram for Rastrigin:



#### Public Member Functions

- double evaluate (const std::vector< double > &x) const override

  *Evaluates the fitness of a candidate solution.*

### Public Member Functions inherited from Problem

- Problem (double lb, double ub, const std::string_view n)

  *Constructs a Problem instance.*
- virtual ~**Problem** ()=default

  *Virtual destructor for safe polymorphic cleanup.*

- double getLowerBound () const
- double getUpperBound () const
- const std::string getName () const

#### Additional Inherited Members

### Protected Attributes inherited from Problem

- const double lowerBound

  *Lower bound of the search space.*
- const double upperBound

  *Upper bound of the search space.*
- const std::string name

  *Name of the benchmark function.*

## 9.166.1 Detailed Description

Implements the Rastrigin benchmark function.

Definition at line 22 of file Rastrigin.h.

## 9.166.2 Constructor & Destructor Documentation

### 9.166.2.1 Rastrigin()

```
Rastrigin::Rastrigin () [inline]
```

Definition at line 29 of file Rastrigin.h.

### 9.166.3 Member Function Documentation

#### 9.166.3.1 evaluate()

```
double Rastrigin::evaluate (
            const std::vector< double > & x) const  [inline], [override], [virtual]
```

Evaluates the fitness of a candidate solution.

- This is a pure virtual function that must be implemented by specific benchmark functions (e.g., Ackley, Rosenbrock).

- **Parameters**

| | |
|---|---|
| *x* | The solution vector to evaluate. |

**Returns**

The scalar fitness value (cost).

Implements Problem.

Definition at line 31 of file Rastrigin.h.

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

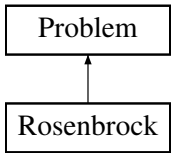
- include/Problem/Rastrigin.h

## 9.167 Rosenbrock Class Reference

Implements the Rosenbrock benchmark function.

```
#include <Rosenbrock.h>
```

Inheritance diagram for Rosenbrock:




**Public Member Functions**

- double evaluate (const std::vector< double > &x) const override

  *Evaluates the fitness of a candidate solution.*

## Public Member Functions inherited from Problem

- Problem (double lb, double ub, const std::string_view n)

  *Constructs a Problem instance.*
- virtual ~**Problem** ()=default

  *Virtual destructor for safe polymorphic cleanup.*

- double getLowerBound () const
- double getUpperBound () const
- const std::string getName () const

#### Additional Inherited Members

## Protected Attributes inherited from Problem

- const double lowerBound

  *Lower bound of the search space.*
- const double upperBound

  *Upper bound of the search space.*
- const std::string name

  *Name of the benchmark function.*

### 9.167.1 Detailed Description

Implements the Rosenbrock benchmark function.

Definition at line 20 of file Rosenbrock.h.

### 9.167.2 Constructor & Destructor Documentation

#### 9.167.2.1 Rosenbrock()

```
Rosenbrock::Rosenbrock ()  [inline]
```

Definition at line 27 of file Rosenbrock.h.

### 9.167.3 Member Function Documentation

#### 9.167.3.1 evaluate()

```
double Rosenbrock::evaluate (
            const std::vector< double > & x) const  [inline], [override], [virtual]
```

Evaluates the fitness of a candidate solution.

- This is a pure virtual function that must be implemented by specific benchmark functions (e.g., Ackley, Rosenbrock).

- **Parameters**

| | |
|---|---|
| *x* | The solution vector to evaluate. |

**Returns**

The scalar fitness value (cost).

Implements Problem.

Definition at line 29 of file Rosenbrock.h.

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

- include/Problem/Rosenbrock.h

## 9.168 RunExperiments Class Reference

High-level controller that orchestrates the benchmarking process.

```
#include <RunExperiments.h>
```

**Public Member Functions**

- RunExperiments (const std::string inputFile, const std::string &outputFile)
  *Constructs the runner and immediately triggers configuration loading.*
- int runExperiments ()
  *The main execution loop for all loaded experiments.*

### 9.168.1 Detailed Description

High-level controller that orchestrates the benchmarking process.

- This class is responsible for:

1. Parsing the JSON configuration file into ExperimentConfig structures.
2. Iterating through experiments and instantiating the required Problems and Optimizers.
3. Collecting performance metrics (fitness and runtime).
4. Exporting results to CSV files for analysis.

Definition at line 29 of file RunExperiments.h.

### 9.168.2 Constructor & Destructor Documentation

#### 9.168.2.1 RunExperiments()

```
RunExperiments::RunExperiments (
            const std::string inputFile,
            const std::string & outputFile)  [inline]
```

Constructs the runner and immediately triggers configuration loading.

**Parameters**

| | |
|---|---|
| *inputFile* | Path to the JSON configuration. |
| *outputFile* | Path to the directory where results will be saved. |

Definition at line 74 of file RunExperiments.h.

### 9.168.3 Member Function Documentation

#### 9.168.3.1 runExperiments()

```
int RunExperiments::runExperiments ()
```

The main execution loop for all loaded experiments.

- Iterates through all configurations, initializes the Problem and Optimizer factories, runs the optimization, and triggers the CSV export.

  **Returns**

- int The total number of experiments successfully processed.

Definition at line 160 of file RunExperiments.cpp.

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

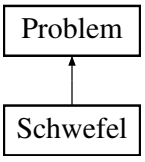
- include/RunExperiments.h
- src/RunExperiments.cpp

## 9.169 Schwefel Class Reference

Implements the Schwefel benchmark function.

```
#include <Schwefel.h>
```

Inheritance diagram for Schwefel:



#### Public Member Functions

- double evaluate (const std::vector< double > &x) const override

  *Evaluates the fitness of a candidate solution.*

### Public Member Functions inherited from Problem

- Problem (double lb, double ub, const std::string_view n)

  *Constructs a Problem instance.*
- virtual ~**Problem** ()=default

  *Virtual destructor for safe polymorphic cleanup.*

- double getLowerBound () const
- double getUpperBound () const
- const std::string getName () const

#### Additional Inherited Members

### Protected Attributes inherited from Problem

- const double lowerBound

  *Lower bound of the search space.*
- const double upperBound

  *Upper bound of the search space.*
- const std::string name

  *Name of the benchmark function.*

## 9.169.1 Detailed Description

Implements the Schwefel benchmark function.

Definition at line 20 of file Schwefel.h.

## 9.169.2 Constructor & Destructor Documentation

### 9.169.2.1 Schwefel()

```
Schwefel::Schwefel () [inline]
```

Definition at line 27 of file Schwefel.h.

### 9.169.3 Member Function Documentation

#### 9.169.3.1 evaluate()

```
double Schwefel::evaluate (
            const std::vector< double > & x) const  [inline], [override], [virtual]
```

Evaluates the fitness of a candidate solution.

- This is a pure virtual function that must be implemented by specific benchmark functions (e.g., Ackley, Rosenbrock).
- **Parameters**

| | |
|---|---|
| *x* | The solution vector to evaluate. |

**Returns**

The scalar fitness value (cost).

Implements Problem.

Definition at line 29 of file Schwefel.h.

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

- include/Problem/Schwefel.h

## 9.170 detail::serializer< BasicJsonType > Class Template Reference

**Public Member Functions**

- serializer (output_adapter_t< char > s, const char ichar, error_handler_t error_handler_=error_handler_t::strict)
- **serializer** (const serializer &)=delete
- serializer & **operator=** (const serializer &)=delete
- **serializer** (serializer &&)=delete
- serializer & **operator=** (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*
- for (std::size_t i=0;i< s.size();++i)
- if (JSON_HEDLEY_LIKELY(state==UTF8_ACCEPT))

**Public Attributes**

- JSON_PRIVATE_UNLESS_TESTED : void dump_escaped(const string_t& s
- JSON_PRIVATE_UNLESS_TESTED const bool ensure_ascii
- std::uint8_t state = UTF8_ACCEPT
- std::size_t bytes = 0
- std::size_t bytes_after_last_accept = 0
- std::size_t undumped_chars = 0
- else
- enable_if_t< std::is_signed< NumberType >::value, int >
- enable_if_t< std::is_unsigned< NumberType >::value, int >
- ∗a ∗(hopefully) large enough character buffer std the locale const std::lconv ∗ loc = nullptr
- ∗the locale s thousand separator character const char thousands_sep = '\0'
- ∗the locale s decimal point character const char decimal_point = '\0'
- ∗string buffer std::array< char, 512 > string_buffer {{}}
- ∗the indentation character const char indent_char
- ∗the indentation string string_t indent_string
- ∗error_handler how to react on decoding errors const error_handler_t error_handler

## 9.170.1 Detailed Description

**template<typename BasicJsonType>**
**class detail::serializer< BasicJsonType >**

Definition at line 18887 of file json.hpp.

## 9.170.2 Constructor & Destructor Documentation

### 9.170.2.1 serializer()

```
template<typename BasicJsonType>
detail::serializer< BasicJsonType >::serializer (
            output_adapter_t< char > s,
            const char ichar,
            error_handler_t error_handler_ = error_handler_t::strict)  [inline]
```

**Parameters**

| | | |
|---|---|---|
| in | *s* | output stream to serialize to |
| in | *ichar* | indentation character to use |
| in | *error_↩ handler_* | how to react on decoding errors |

Definition at line 18903 of file json.hpp.

## 9.170.3 Member Function Documentation

### 9.170.3.1 dump()

```
template<typename BasicJsonType>
void 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
- binary values are serialized as objects containing the subtype and the byte array

**Parameters**

| | | |
|---|---|---|
| in | *val* | value to serialize |
| in | *pretty_print* | whether the output shall be pretty-printed |
| in | *ensure_ascii* | If *ensure_ascii* is true, all non-ASCII characters in the output are escaped with `\uXXXX` sequences, and the result consists of ASCII characters only. |
| in | *indent_step* | the indent level |
| in | *current_indent* | the current indent level (only used internally) |

Definition at line 18943 of file json.hpp.

### 9.170.3.2 for()

```
template<typename BasicJsonType>
detail::serializer< BasicJsonType >::for ()  [inline]
```

Definition at line 19237 of file json.hpp.

### 9.170.3.3 if()

```
template<typename BasicJsonType>
detail::serializer< BasicJsonType >::if (
            JSON_HEDLEY_LIKELY(state==UTF8_ACCEPT) )  [inline]
```

Definition at line 19426 of file json.hpp.

## 9.170.4 Member Data Documentation

### 9.170.4.1 __pad0__

```
template<typename BasicJsonType>
JSON_PRIVATE_UNLESS_TESTED detail::serializer< BasicJsonType >::__pad0__
```

Definition at line 19212 of file json.hpp.

### 9.170.4.2 bytes

```
template<typename BasicJsonType>
std::size_t detail::serializer< BasicJsonType >::bytes = 0
```

Definition at line 19231 of file json.hpp.

### 9.170.4.3 bytes_after_last_accept

```
template<typename BasicJsonType>
std::size_t detail::serializer< BasicJsonType >::bytes_after_last_accept = 0
```

Definition at line 19234 of file json.hpp.

### 9.170.4.4 decimal_point

```
template<typename BasicJsonType>
* the locale s decimal point character const char detail::serializer< BasicJsonType >::↩
decimal_point = '\0'
```

Definition at line 19809 of file json.hpp.

### 9.170.4.5 else

```
template<typename BasicJsonType>
detail::serializer< BasicJsonType >::else
```

Definition at line 19434 of file json.hpp.

### 9.170.4.6 enable_if_t< std::is_signed< NumberType >::value, int >

```
template<typename BasicJsonType>
detail::serializer< BasicJsonType >::enable_if_t< std::is_signed< NumberType >::value, int >
```

**Initial value:**

```
          = 0>
bool is_negative_number(NumberType x)
{
    return x < 0;
}

template < typename NumberType
```

Definition at line 19434 of file json.hpp.

#### 9.170.4.7 enable_if_t< std::is_unsigned< NumberType >::value, int >

```
template<typename BasicJsonType>
detail::serializer< BasicJsonType >::enable_if_t< std::is_unsigned< NumberType >::value, int
>
```

**Initial value:**

```
           = 0 >
bool is_negative_number(NumberType )
{
    return false;
}

template < typename NumberType
```

Definition at line 19434 of file json.hpp.

#### 9.170.4.8 ensure_ascii

```
template<typename BasicJsonType>
JSON_PRIVATE_UNLESS_TESTED const bool detail::serializer< BasicJsonType >::ensure_ascii
```

**Initial value:**

```
{
        std::uint32_t codepoint{}
```

Definition at line 19227 of file json.hpp.

#### 9.170.4.9 error_handler

```
template<typename BasicJsonType>
* error_handler how to react on decoding errors const error_handler_t detail::serializer<
BasicJsonType >::error_handler
```

Definition at line 19820 of file json.hpp.

#### 9.170.4.10 indent_char

```
template<typename BasicJsonType>
* the indentation character const char detail::serializer< BasicJsonType >::indent_char
```

Definition at line 19815 of file json.hpp.

#### 9.170.4.11 indent_string

```
template<typename BasicJsonType>
* the indentation string string_t detail::serializer< BasicJsonType >::indent_string
```

Definition at line 19817 of file json.hpp.

#### 9.170.4.12 loc

```
template<typename BasicJsonType>
* a* (hopefully) large enough character buffer std the locale const std::lconv* detail::serializer<
BasicJsonType >::loc = nullptr
```

Definition at line 19805 of file json.hpp.

#### 9.170.4.13 state

```
template<typename BasicJsonType>
std::uint8_t detail::serializer< BasicJsonType >::state = UTF8_ACCEPT
```

Definition at line 19230 of file json.hpp.

#### 9.170.4.14 string_buffer

```
template<typename BasicJsonType>
* string buffer std::array<char, 512> detail::serializer< BasicJsonType >::string_buffer {{}}
```

Definition at line 19812 of file json.hpp.

#### 9.170.4.15 thousands_sep

```
template<typename BasicJsonType>
* the locale s thousand separator character const char detail::serializer< BasicJsonType >::↩
thousands_sep = '\0'
```

Definition at line 19807 of file json.hpp.

#### 9.170.4.16 undumped_chars

```
template<typename BasicJsonType>
std::size_t detail::serializer< BasicJsonType >::undumped_chars = 0
```

Definition at line 19235 of file json.hpp.

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

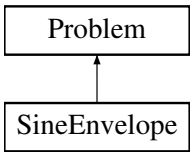
- include/External/json.hpp

## 9.171 SineEnvelope Class Reference

Implements the Sine Envelope benchmark function.

`#include <SineEnvelope.h>`

Inheritance diagram for SineEnvelope:



#### Public Member Functions

- double evaluate (const std::vector< double > &x) const override

  *Evaluates the fitness of a candidate solution.*

### Public Member Functions inherited from Problem

- Problem (double lb, double ub, const std::string_view n)

  *Constructs a Problem instance.*
- virtual ~**Problem** ()=default

  *Virtual destructor for safe polymorphic cleanup.*

- double getLowerBound () const
- double getUpperBound () const
- const std::string getName () const

#### Additional Inherited Members

### Protected Attributes inherited from Problem

- const double lowerBound

  *Lower bound of the search space.*
- const double upperBound

  *Upper bound of the search space.*
- const std::string name

  *Name of the benchmark function.*

## 9.171.1 Detailed Description

Implements the Sine Envelope benchmark function.

Definition at line 21 of file SineEnvelope.h.

## 9.171.2 Constructor & Destructor Documentation

### 9.171.2.1 SineEnvelope()

```
SineEnvelope::SineEnvelope () [inline]
```

Definition at line 28 of file SineEnvelope.h.

### 9.171.3 Member Function Documentation

#### 9.171.3.1 evaluate()

```
double SineEnvelope::evaluate (
            const std::vector< double > & x) const  [inline], [override], [virtual]
```

Evaluates the fitness of a candidate solution.

- This is a pure virtual function that must be implemented by specific benchmark functions (e.g., Ackley, Rosenbrock).

- **Parameters**

| | |
|---|---|
| *x* | The solution vector to evaluate. |

**Returns**

The scalar fitness value (cost).

Implements Problem.

Definition at line 30 of file SineEnvelope.h.

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

- include/Problem/SineEnvelope.h

## 9.172 SolutionBuilder Class Reference

Responsible for creating random solutions and neighborhood samples.

```
#include <SolutionBuilder.h>
```

### Public Member Functions

- SolutionBuilder (int dimensions, int lower, int upper, int seed)

  *Constructs a SolutionBuilder with specific space constraints.*
- std::vector< double > getRand ()

  *Generates a single random solution vector within bounds.*
- std::vector< std::vector< double > > getNeighbors (const std::vector< double > &center, int numNeighbors, double maxDelta)

  *Generates a set of neighboring solutions around a central point.*
- double getDimensions ()

  *Returns the dimensionality of the solution space.*

### 9.172.1 Detailed Description

Responsible for creating random solutions and neighborhood samples.

This class encapsulates the logic for generating initial random positions within the search space and perturbing existing solutions to find neighbors. It utilizes the Mersenne Twister algorithm for high-quality random number generation.

Definition at line 26 of file SolutionBuilder.h.

### 9.172.2 Constructor & Destructor Documentation

#### 9.172.2.1 SolutionBuilder()

```
SolutionBuilder::SolutionBuilder (
            int dimensions,
            int lower,
            int upper,
            int seed)  [inline]
```

Constructs a SolutionBuilder with specific space constraints.

**Parameters**

| | |
|---|---|
| *dimensions* | Number of variables in the solution vector. |
| *lower* | Minimum value for any given dimension. |
| *upper* | Maximum value for any given dimension. |
| *seed* | Value used to initialize the Mersenne Twister generator. |

Definition at line 47 of file SolutionBuilder.h.

### 9.172.3 Member Function Documentation

#### 9.172.3.1 getDimensions()

```
double SolutionBuilder::getDimensions ()  [inline]
```

Returns the dimensionality of the solution space.

Definition at line 79 of file SolutionBuilder.h.

#### 9.172.3.2 getNeighbors()

```
std::vector< std::vector< double > > SolutionBuilder::getNeighbors (
            const std::vector< double > & center,
            int numNeighbors,
            double maxDelta)
```

Generates a set of neighboring solutions around a central point.

Used primarily by Local Search algorithms to explore the immediate vicinity of the current best candidate.

**Parameters**

| | |
|---|---|
| *center* | The original solution vector to perturb. |

| | |
|---|---|
| *numNeighbors* | Number of neighbor vectors to generate. |
| *maxDelta* | The maximum step size allowed for perturbation in any dimension. |

**Returns**

A collection of neighboring solution vectors.

Definition at line 25 of file SolutionBuilder.cpp.

#### 9.172.3.3 getRand()

```
std::vector< double > SolutionBuilder::getRand ()
```

Generates a single random solution vector within bounds.

**Returns**

A vector of size dimensions with values in range [lower, upper].

Definition at line 4 of file SolutionBuilder.cpp.

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

- include/SolutionBuilder.h
- src/SolutionBuilder.cpp

## 9.173 detail::span_input_adapter Class Reference

### Public Member Functions

- template<typename CharT, typename std::enable_if< std::is_pointer< CharT >::value &&std::is_integral< typename std::remove_↩ pointer< CharT >::type >::value &&sizeof(typename std::remove_pointer< CharT >::type)==1, int >::type = 0> span_input_adapter (CharT b, std::size_t l)
- template<class IteratorType, typename std::enable_if< std::is_same< typename iterator_traits< IteratorType >::iterator_category, std::random_access_iterator_tag >::value, int >::type = 0> span_input_adapter (IteratorType first, IteratorType last)
- contiguous_bytes_input_adapter && get ()

### 9.173.1 Detailed Description

Definition at line 7034 of file json.hpp.

### 9.173.2 Constructor & Destructor Documentation

#### 9.173.2.1 span_input_adapter() [1/2]

```
template<typename CharT, typename std::enable_if< std::is_pointer< CharT >::value &&std::↩
is_integral< typename std::remove_pointer< CharT >::type >::value &&sizeof(typename std::↩
remove_pointer< CharT >::type)==1, int >::type = 0>
detail::span_input_adapter::span_input_adapter (
            CharT b,
            std::size_t l)  [inline]
```

Definition at line 7043 of file json.hpp.

#### 9.173.2.2 span_input_adapter() [2/2]

```
template<class IteratorType, typename std::enable_if< std::is_same< typename iterator_traits<
IteratorType >::iterator_category, std::random_access_iterator_tag >::value, int >::type = 0>
detail::span_input_adapter::span_input_adapter (
            IteratorType first,
            IteratorType last)  [inline]
```

Definition at line 7050 of file json.hpp.

### 9.173.3 Member Function Documentation

#### 9.173.3.1 get()

```
contiguous_bytes_input_adapter && detail::span_input_adapter::get ()  [inline]
```

Definition at line 7053 of file json.hpp.

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

- include/External/json.hpp

## 9.174 detail::static_const< T > Struct Template Reference

**Static Public Attributes**

- static JSON_INLINE_VARIABLE constexpr T value {}

### 9.174.1 Detailed Description

**template<typename T>**
**struct detail::static_const< T >**

Definition at line 3342 of file json.hpp.

### 9.174.2 Member Data Documentation

#### 9.174.2.1 value

```
template<typename T>
JSON_INLINE_VARIABLE constexpr T detail::static_const< T >::value {}  [static], [constexpr]
```

Definition at line 3344 of file json.hpp.

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

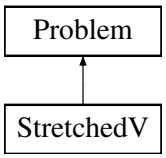
- include/External/json.hpp

## 9.175 StretchedV Class Reference

Implements the StretchedV benchmark function.

```
#include <StretchedV.h>
```

Inheritance diagram for StretchedV:




#### Public Member Functions

- double evaluate (const std::vector< double > &x) const override

  *Evaluates the fitness of a candidate solution.*

### Public Member Functions inherited from Problem

- Problem (double lb, double ub, const std::string_view n)

  *Constructs a Problem instance.*
- virtual ~**Problem** ()=default

  *Virtual destructor for safe polymorphic cleanup.*

- double getLowerBound () const
- double getUpperBound () const
- const std::string getName () const

**Additional Inherited Members**

## Protected Attributes inherited from Problem

- const double lowerBound

  *Lower bound of the search space.*
- const double upperBound

  *Upper bound of the search space.*
- const std::string name

  *Name of the benchmark function.*

### 9.175.1 Detailed Description

Implements the StretchedV benchmark function.

Definition at line 21 of file StretchedV.h.

### 9.175.2 Constructor & Destructor Documentation

#### 9.175.2.1 StretchedV()

```
StretchedV::StretchedV () [inline]
```

Definition at line 28 of file StretchedV.h.

### 9.175.3 Member Function Documentation

#### 9.175.3.1 evaluate()

```
double StretchedV::evaluate (
            const std::vector< double > & x) const [inline], [override], [virtual]
```

Evaluates the fitness of a candidate solution.

- This is a pure virtual function that must be implemented by specific benchmark functions (e.g., Ackley, Rosenbrock).
- **Parameters**

| | |
|---|---|
| *x* | The solution vector to evaluate. |

**Returns**

The scalar fitness value (cost).

Implements Problem.

Definition at line 30 of file StretchedV.h.

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

- include/Problem/StretchedV.h

# 9.176 string_t_helper< T > Struct Template Reference

**Public Types**

- using type = T

## 9.176.1 Detailed Description

**template<typename T>**
**struct string_t_helper< T >**

Definition at line 14651 of file json.hpp.

## 9.176.2 Member Typedef Documentation

### 9.176.2.1 type

```
template<typename T>
using string_t_helper< T >::type = T
```

Definition at line 14653 of file json.hpp.

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

- include/External/json.hpp

# 9.177 string_t_helper< NLOHMANN_BASIC_JSON_TPL > Struct Reference

**Public Types**

- using type = StringType

## 9.177.1 Detailed Description

Definition at line 14657 of file json.hpp.

## 9.177.2 Member Typedef Documentation

### 9.177.2.1 type

```
using string_t_helper< NLOHMANN_BASIC_JSON_TPL >::type = StringType
```

Definition at line 14659 of file json.hpp.

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

- include/External/json.hpp

## 9.178 detail::to_json_fn Struct Reference

### Public Member Functions

- template<typename BasicJsonType, typename T>
  auto operator() (BasicJsonType &j, T &&val) const noexcept(noexcept(to_json(j, std::forward< T >(val)))) -> decltype(to_json(j, std::forward< T >(val)), void())

### 9.178.1 Detailed Description

Definition at line 6167 of file json.hpp.

### 9.178.2 Member Function Documentation

#### 9.178.2.1 operator()()

```
template<typename BasicJsonType, typename T>
auto detail::to_json_fn::operator() (
            BasicJsonType & j,
            T && val) const -> decltype(to_json(j, std::forward< T >(val)), void())  [inline],
[noexcept]
```

Definition at line 6170 of file json.hpp.

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

- include/External/json.hpp

## 9.179 std::tuple_element< N, ::nlohmann::detail::iteration_proxy_value< IteratorType > > Class Template Reference

### Public Types

- using type

### 9.179.1 Detailed Description

**template<std::size_t N, typename IteratorType>**
**class std::tuple_element< N, ::nlohmann::detail::iteration_proxy_value< IteratorType > >**

Definition at line 5710 of file json.hpp.

### 9.179.2 Member Typedef Documentation

#### 9.179.2.1 type

```
template<std::size_t N, typename IteratorType>
using std::tuple_element< N, ::nlohmann::detail::iteration_proxy_value< IteratorType > >::↩
type
```

**Initial value:**

```
decltype(
                    get<N>(std::declval <
                           ::nlohmann::detail::iteration_proxy_value<IteratorType » ()))
```

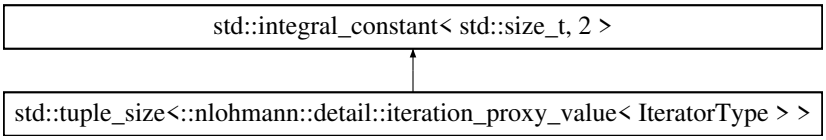
Definition at line 5713 of file json.hpp.

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

- include/External/json.hpp

## 9.180 std::tuple_size<::nlohmann::detail::iteration_proxy_value< IteratorType > > Class Template Reference

Inheritance diagram for std::tuple_size<::nlohmann::detail::iteration_proxy_value< IteratorType > >:




### 9.180.1 Detailed Description

**template<typename IteratorType>**
**class std::tuple_size<::nlohmann::detail::iteration_proxy_value< IteratorType > >**

Definition at line 5706 of file json.hpp.

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

- include/External/json.hpp

## 9.181 detail::is_ordered_map< T >::two Struct Reference

**Public Attributes**

- char x [2]

### 9.181.1 Detailed Description

**template<typename T>**
**struct detail::is_ordered_map< T >::two**

Definition at line 4212 of file json.hpp.

### 9.181.2 Member Data Documentation

#### 9.181.2.1 x

```
template<typename T>
char detail::is_ordered_map< T >::two::x[2]
```

Definition at line 4214 of file json.hpp.

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

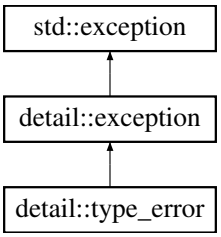
- include/External/json.hpp

## 9.182 detail::type_error Class Reference

exception indicating executing a member function with a wrong type

`#include <json.hpp>`

Inheritance diagram for detail::type_error:




#### Static Public Member Functions

- template<typename BasicJsonContext, enable_if_t< is_basic_json_context< BasicJsonContext >::value, int > = 0>
  static type_error create (int id_, const std::string &what_arg, BasicJsonContext context)

#### Additional Inherited Members

### Public Member Functions inherited from detail::exception

- const char * what () const noexcept override
  *returns the explanatory string*

## Public Attributes inherited from detail::exception

- const int id

  *the id of the exception*

## Protected Member Functions inherited from detail::exception

- exception (int id_, const char *what_arg)

## Static Protected Member Functions inherited from detail::exception

- static std::string name (const std::string &ename, int id_)
- static std::string diagnostics (std::nullptr_t)
- template<typename BasicJsonType>
  static std::string diagnostics (const BasicJsonType *leaf_element)

### 9.182.1 Detailed Description

exception indicating executing a member function with a wrong type

**See also**

https://json.nlohmann.me/api/basic_json/type_error/

Definition at line 4745 of file json.hpp.

### 9.182.2 Member Function Documentation

#### 9.182.2.1 create()

```
template<typename BasicJsonContext, enable_if_t< is_basic_json_context< BasicJsonContext
>::value, int > = 0>
type_error detail::type_error::create (
            int id_,
            const std::string & what_arg,
            BasicJsonContext context)  [inline], [static]
```

Definition at line 4749 of file json.hpp.

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

- include/External/json.hpp

## 9.183 detail::value_in_range_of_impl1< OfType, T, NeverOutOfRange, typename > Struct Template Reference

### 9.183.1 Detailed Description

**template<typename OfType, typename T, bool NeverOutOfRange = never_out_of_range<OfType, T>::value, typename = detail::enable_if_t<all_integral<OfType, T>::value>>**
**struct detail::value_in_range_of_impl1< OfType, T, NeverOutOfRange, typename >**

Definition at line 4304 of file json.hpp.

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

- include/External/json.hpp

## 9.184 detail::value_in_range_of_impl1< OfType, T, false > Struct Template Reference

**Static Public Member Functions**

- static constexpr bool test (T val)

### 9.184.1 Detailed Description

**template<typename OfType, typename T>**
**struct detail::value_in_range_of_impl1< OfType, T, false >**

Definition at line 4307 of file json.hpp.

### 9.184.2 Member Function Documentation

#### 9.184.2.1 test()

```
template<typename OfType, typename T>
constexpr bool detail::value_in_range_of_impl1< OfType, T, false >::test (
            T val)  [inline], [static], [constexpr]
```

Definition at line 4309 of file json.hpp.

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

- include/External/json.hpp

# 9.185 detail::value_in_range_of_impl1< OfType, T, true > Struct Template Reference

**Static Public Member Functions**

- static constexpr bool test (T)

## 9.185.1 Detailed Description

**template<typename OfType, typename T>**
**struct detail::value_in_range_of_impl1< OfType, T, true >**

Definition at line 4316 of file json.hpp.

## 9.185.2 Member Function Documentation

### 9.185.2.1 test()

```
template<typename OfType, typename T>
constexpr bool detail::value_in_range_of_impl1< OfType, T, true >::test (
            T )  [inline], [static], [constexpr]
```

Definition at line 4318 of file json.hpp.

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

- include/External/json.hpp

# 9.186 detail::value_in_range_of_impl2< OfType, T, OfTypeSigned, TSigned > Struct Template Reference

## 9.186.1 Detailed Description

**template<typename OfType, typename T, bool OfTypeSigned = std::is_signed<OfType>::value, bool TSigned = std::is_signed<T>::value>**
**struct detail::value_in_range_of_impl2< OfType, T, OfTypeSigned, TSigned >**

Definition at line 4258 of file json.hpp.

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

- include/External/json.hpp

# 9.187 detail::value_in_range_of_impl2< OfType, T, false, false > Struct Template Reference

**Static Public Member Functions**

- static constexpr bool test (T val)

## 9.187.1 Detailed Description

**template<typename OfType, typename T>**
**struct detail::value_in_range_of_impl2< OfType, T, false, false >**

Definition at line 4261 of file json.hpp.

## 9.187.2 Member Function Documentation

### 9.187.2.1 test()

```
template<typename OfType, typename T>
constexpr bool detail::value_in_range_of_impl2< OfType, T, false, false >::test (
            T val)  [inline], [static], [constexpr]
```

Definition at line 4263 of file json.hpp.

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

- include/External/json.hpp

# 9.188 detail::value_in_range_of_impl2< OfType, T, false, true > Struct Template Reference

**Static Public Member Functions**

- static constexpr bool test (T val)

## 9.188.1 Detailed Description

**template<typename OfType, typename T>**
**struct detail::value_in_range_of_impl2< OfType, T, false, true >**

Definition at line 4281 of file json.hpp.

### 9.188.2 Member Function Documentation

#### 9.188.2.1 test()

```
template<typename OfType, typename T>
constexpr bool detail::value_in_range_of_impl2< OfType, T, false, true >::test (
            T val)  [inline], [static], [constexpr]
```

Definition at line 4283 of file json.hpp.

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

- include/External/json.hpp

## 9.189 detail::value_in_range_of_impl2< OfType, T, true, false > Struct Template Reference

**Static Public Member Functions**

- static constexpr bool test (T val)

### 9.189.1 Detailed Description

**template<typename OfType, typename T>**
**struct detail::value_in_range_of_impl2< OfType, T, true, false >**

Definition at line 4271 of file json.hpp.

### 9.189.2 Member Function Documentation

#### 9.189.2.1 test()

```
template<typename OfType, typename T>
constexpr bool detail::value_in_range_of_impl2< OfType, T, true, false >::test (
            T val)  [inline], [static], [constexpr]
```

Definition at line 4273 of file json.hpp.

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

- include/External/json.hpp

## 9.190 detail::value_in_range_of_impl2< OfType, T, true, true > Struct Template Reference

**Static Public Member Functions**

- static constexpr bool test (T val)

### 9.190.1 Detailed Description

**template<typename OfType, typename T>**
**struct detail::value_in_range_of_impl2< OfType, T, true, true >**

Definition at line 4291 of file json.hpp.

### 9.190.2 Member Function Documentation

#### 9.190.2.1 test()

```
template<typename OfType, typename T>
constexpr bool detail::value_in_range_of_impl2< OfType, T, true, true >::test (
            T val)  [inline], [static], [constexpr]
```

Definition at line 4293 of file json.hpp.

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

- include/External/json.hpp

## 9.191 detail::wide_string_input_adapter< BaseInputAdapter, WideCharType > Class Template Reference

**Public Types**

- using char_type = char

**Public Member Functions**

- wide_string_input_adapter (BaseInputAdapter base)
- std::char_traits< char >::int_type get_character () noexcept
- template<class T>
  std::size_t get_elements (T *, std::size_t=1)

### 9.191.1 Detailed Description

**template<typename BaseInputAdapter, typename WideCharType>**
**class detail::wide_string_input_adapter< BaseInputAdapter, WideCharType >**

Definition at line 6851 of file json.hpp.

## 9.191.2 Member Typedef Documentation

### 9.191.2.1 char_type

```
template<typename BaseInputAdapter, typename WideCharType>
using detail::wide_string_input_adapter< BaseInputAdapter, WideCharType >::char_type = char
```

Definition at line 6854 of file json.hpp.

## 9.191.3 Constructor & Destructor Documentation

### 9.191.3.1 wide_string_input_adapter()

```
template<typename BaseInputAdapter, typename WideCharType>
detail::wide_string_input_adapter< BaseInputAdapter, WideCharType >::wide_string_input_adapter
(
            BaseInputAdapter base)  [inline]
```

Definition at line 6856 of file json.hpp.

## 9.191.4 Member Function Documentation

### 9.191.4.1 get_character()

```
template<typename BaseInputAdapter, typename WideCharType>
std::char_traits< char >::int_type detail::wide_string_input_adapter< BaseInputAdapter, Wide↩
CharType >::get_character ()  [inline], [noexcept]
```

Definition at line 6859 of file json.hpp.

### 9.191.4.2 get_elements()

```
template<typename BaseInputAdapter, typename WideCharType>
template<class T>
std::size_t detail::wide_string_input_adapter< BaseInputAdapter, WideCharType >::get_elements
(
            T * ,
            std::size_t  = 1)  [inline]
```

Definition at line 6878 of file json.hpp.

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

- include/External/json.hpp

# 9.192 detail::wide_string_input_helper< BaseInputAdapter, T > Struct Template Reference

## 9.192.1 Detailed Description

**template<typename BaseInputAdapter, size_t T>**
**struct detail::wide_string_input_helper< BaseInputAdapter, T >**

Definition at line 6727 of file json.hpp.

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

- include/External/json.hpp

# 9.193 detail::wide_string_input_helper< BaseInputAdapter, 2 > Struct Template Reference

### Static Public Member Functions

- static void fill_buffer (BaseInputAdapter &input, std::array< std::char_traits< char >::int_type, 4 > &utf8_↩ bytes, size_t &utf8_bytes_index, size_t &utf8_bytes_filled)

## 9.193.1 Detailed Description

**template<typename BaseInputAdapter>**
**struct detail::wide_string_input_helper< BaseInputAdapter, 2 >**

Definition at line 6788 of file json.hpp.

## 9.193.2 Member Function Documentation

### 9.193.2.1 fill_buffer()

```
template<typename BaseInputAdapter>
void detail::wide_string_input_helper< BaseInputAdapter, 2 >::fill_buffer (
            BaseInputAdapter & input,
            std::array< std::char_traits< char >::int_type, 4 > & utf8_bytes,
            size_t & utf8_bytes_index,
            size_t & utf8_bytes_filled)  [inline], [static]
```

Definition at line 6791 of file json.hpp.

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

- include/External/json.hpp

# 9.194 detail::wide_string_input_helper< BaseInputAdapter, 4 > Struct Template Reference

**Static Public Member Functions**

- static void fill_buffer (BaseInputAdapter &input, std::array< std::char_traits< char >::int_type, 4 > &utf8_↩ bytes, size_t &utf8_bytes_index, size_t &utf8_bytes_filled)

## 9.194.1 Detailed Description

**template<typename BaseInputAdapter>**
**struct detail::wide_string_input_helper< BaseInputAdapter, 4 >**

Definition at line 6730 of file json.hpp.

## 9.194.2 Member Function Documentation

### 9.194.2.1 fill_buffer()

```
template<typename BaseInputAdapter>
void detail::wide_string_input_helper< BaseInputAdapter, 4 >::fill_buffer (
            BaseInputAdapter & input,
            std::array< std::char_traits< char >::int_type, 4 > & utf8_bytes,
            size_t & utf8_bytes_index,
            size_t & utf8_bytes_filled)  [inline], [static]
```

Definition at line 6733 of file json.hpp.

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

- include/External/json.hpp

# Chapter 10

# File Documentation

## 10.1 BenchmarkRunner.h

```
00001 #ifndef BENCHMARK_RUNNER_H
00002 #define BENCHMARK_RUNNER_H
00003
00004
00005 #include "External/json.hpp"
00006
00007 #include <vector>
00008 #include <string>
00009
00010 #include "Experiment.h"
00011
00012 using json = nlohmann::json;
00013
00014 class BenchmarkRunner {
00015 private:
00016     inline static constexpr std::string fitnessCSV = "fitness.csv";
00017     inline static constexpr std::string timeCSV = "time.csv";
00018
00019     json loadJSON(const std::string& path);
00020     Experiment parseExperiment(const json& j);
00021     std::vector<Experiment> parseExperiments(const json& cfg);
00022     void writeFitnessCSV(const std::vector<Experiment>& experiments, const std::string& filename);
00023     void writeTimeCSV(const std::vector<Experiment>& experiments,const std::string& filename);
00024 public:
00025     void runBenchmarks(const std::string& inputFile, const std::string& benchmarkName);
00026
00027 };
00028
00029 #endif
```

## 10.2 include/Config.h File Reference

Defines the configuration structure for optimization experiments.

```
#include <string>
```

### Classes

- struct ExperimentConfig

  *Container for all parameters required to execute a benchmark run.*

### 10.2.1 Detailed Description

Defines the configuration structure for optimization experiments.

**Author**

Alex Buckley

Definition in file Config.h.

## 10.3 Config.h

Go to the documentation of this file.

```
00001
00007
00008
00009 #ifndef CONFIG_H
00010 #define CONFIG_H
00011
00012 #include <string>
00013
00014
00021 typedef struct {
00022 std::string experimentName;
00023     int problemType;
00024     int dimensions;
00025     double lower;
00026     double upper;
00027     int seed;
00028     std::string optimizer;
00029     int maxIterations;
00030     double neighborDelta;
00031     int numNeighbors;
00032 } ExperimentConfig;
00033
00034
00035
00036 #endif
```

## 10.4 debug.h

```
00001 namespace debug {
00002
00003 constexpr bool enabled =
00004 #ifndef NDEBUG
00005     true;
00006 #else
00007     false;
00008 #endif
00009
00010 template<typename... Args>
00011 inline void log(Args&&... args) {
00012     if constexpr (enabled) {
00013         (std::cerr « ... « args) « '\n';
00014     }
00015 }
00016
00017 } // namespace debug
```

## 10.5 Experiment.h

```
00001 #ifndef EXPERIMENT_H
00002 #define EXPERIMENT_H
00003
00004 #include <vector>
00005 #include <filesystem>
00006
00007 #include "Population.h"
00008 #include "Problem/Problem.h"
00009 #include "ProblemFactory.h"
00010
00011 class Experiment {
00012 private:
00013     const std::string name;
00014     const int problemType;
00015     const int populationSize;
00016     const int dimensions;
00017
00018     std::unique_ptr<Problem> problem;
00019     Population population;
00020     std::vector<double> fitness;
00021     double wallTime;
00022 public:
00023     Experiment(std::string name,
00024                int problemType,
00025                int popSize,
00026                int dims,
00027                unsigned int seed,
00028                int lower,
00029                int upper)
00030         : name(std::move(name)),
00031           problemType(problemType),
00032           populationSize(popSize),
00033           dimensions(dims),
00034           problem(ProblemFactory::create(problemType)),
00035           population(popSize, dims),
00036           wallTime(0.0)
00037     {
00038         population.initialize(lower, upper, seed);
00039     }
00040
00041     void runExperiment();
00042
00043     inline const std::string& getName() const { return name; }
00044     inline const std::vector<double>& getFitness() const { return fitness; }
00045     inline double getWallTime() const {return wallTime; }
00046 };
00047
00048 #endif
```

## 10.6 ExperimentResult.h

```
00001 #ifdef EXPERIMENT_RESULT_H
00002 #define EXPERIMENT_RESULT_H
00003
00004 #include <vector>
00005
00006 class ExperimentResult {
00007 public:
00008     std::vector<double> bestFitnesses; // Best fitness after each iteration
00009     std::vector<std::vector<double» solutions; // Solution vector tested at each iteration
00010     const double time; // Total optimization time
00011
00012     Result(const std::vector<double>& best,
00013            const std::vector<std::vector<double»& sols,
00014            double t)
00015         : bestFitnesses(best), solutions(sols), time(t) {}
00016
00017     // Optional: default destructor
00018     ~Result() = default;
00019
00020 }
00021 #endif
```

## 10.7 json.hpp

```
00001 //     __ _____ _____ _____
```

```
00002 //  __|  |   __|     |   | |  JSON for Modern C++
00003 // |  |  |__   |  |  | | | |  version 3.12.0
00004 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
00005 //
00006 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
00007 // SPDX-License-Identifier: MIT
00008
00009 /****************************************************************************\
00010  * Note on documentation: The source files contain links to the online      *
00011  * documentation of the public API at https://json.nlohmann.me. This URL    *
00012  * contains the most recent documentation and should also be applicable to  *
00013  * previous versions; documentation for deprecated functions is not         *
00014  * removed, but marked deprecated. See "Generate documentation" section in  *
00015  * file docs/README.md.                                                     *
00016 \****************************************************************************/
00017
00018 #ifndef INCLUDE_NLOHMANN_JSON_HPP_
00019 #define INCLUDE_NLOHMANN_JSON_HPP_
00020
00021 #include <algorithm> // all_of, find, for_each
00022 #include <cstddef> // nullptr_t, ptrdiff_t, size_t
00023 #include <functional> // hash, less
00024 #include <initializer_list> // initializer_list
00025 #ifndef JSON_NO_IO
00026     #include <iosfwd> // istream, ostream
00027 #endif  // JSON_NO_IO
00028 #include <iterator> // random_access_iterator_tag
00029 #include <memory> // unique_ptr
00030 #include <string> // string, stoi, to_string
00031 #include <utility> // declval, forward, move, pair, swap
00032 #include <vector> // vector
00033
00034 // #include <nlohmann/adl_serializer.hpp>
00035 //     __ _____ _____ _____
00036 //  __|  |   __|     |   | |  JSON for Modern C++
00037 // |  |  |__   |  |  | | | |  version 3.12.0
00038 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
00039 //
00040 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
00041 // SPDX-License-Identifier: MIT
00042
00043
00044
00045 #include <utility>
00046
00047 // #include <nlohmann/detail/abi_macros.hpp>
00048 //     __ _____ _____ _____
00049 //  __|  |   __|     |   | |  JSON for Modern C++
00050 // |  |  |__   |  |  | | | |  version 3.12.0
00051 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
00052 //
00053 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
00054 // SPDX-License-Identifier: MIT
00055
00056
00057
00058 // This file contains all macro definitions affecting or depending on the ABI
00059
00060 #ifndef JSON_SKIP_LIBRARY_VERSION_CHECK
00061     #if defined(NLOHMANN_JSON_VERSION_MAJOR) && defined(NLOHMANN_JSON_VERSION_MINOR) &&
      defined(NLOHMANN_JSON_VERSION_PATCH)
00062         #if NLOHMANN_JSON_VERSION_MAJOR != 3 || NLOHMANN_JSON_VERSION_MINOR != 12 ||
      NLOHMANN_JSON_VERSION_PATCH != 0
00063             #warning "Already included a different version of the library!"
00064         #endif
00065     #endif
00066 #endif
00067
00068 #define NLOHMANN_JSON_VERSION_MAJOR 3   // NOLINT(modernize-macro-to-enum)
00069 #define NLOHMANN_JSON_VERSION_MINOR 12  // NOLINT(modernize-macro-to-enum)
00070 #define NLOHMANN_JSON_VERSION_PATCH 0   // NOLINT(modernize-macro-to-enum)
00071
00072 #ifndef JSON_DIAGNOSTICS
00073     #define JSON_DIAGNOSTICS 0
00074 #endif
00075
00076 #ifndef JSON_DIAGNOSTIC_POSITIONS
00077     #define JSON_DIAGNOSTIC_POSITIONS 0
00078 #endif
00079
00080 #ifndef JSON_USE_LEGACY_DISCARDED_VALUE_COMPARISON
00081     #define JSON_USE_LEGACY_DISCARDED_VALUE_COMPARISON 0
00082 #endif
00083
00084 #if JSON_DIAGNOSTICS
00085     #define NLOHMANN_JSON_ABI_TAG_DIAGNOSTICS _diag
00086 #else
```

```
00087     #define NLOHMANN_JSON_ABI_TAG_DIAGNOSTICS
00088 #endif
00089
00090 #if JSON_DIAGNOSTIC_POSITIONS
00091     #define NLOHMANN_JSON_ABI_TAG_DIAGNOSTIC_POSITIONS _dp
00092 #else
00093     #define NLOHMANN_JSON_ABI_TAG_DIAGNOSTIC_POSITIONS
00094 #endif
00095
00096 #if JSON_USE_LEGACY_DISCARDED_VALUE_COMPARISON
00097     #define NLOHMANN_JSON_ABI_TAG_LEGACY_DISCARDED_VALUE_COMPARISON _ldvcmp
00098 #else
00099     #define NLOHMANN_JSON_ABI_TAG_LEGACY_DISCARDED_VALUE_COMPARISON
00100 #endif
00101
00102 #ifndef NLOHMANN_JSON_NAMESPACE_NO_VERSION
00103     #define NLOHMANN_JSON_NAMESPACE_NO_VERSION 0
00104 #endif
00105
00106 // Construct the namespace ABI tags component
00107 #define NLOHMANN_JSON_ABI_TAGS_CONCAT_EX(a, b, c) json_abi ## a ## b ## c
00108 #define NLOHMANN_JSON_ABI_TAGS_CONCAT(a, b, c) \
00109     NLOHMANN_JSON_ABI_TAGS_CONCAT_EX(a, b, c)
00110
00111 #define NLOHMANN_JSON_ABI_TAGS                                       \
00112     NLOHMANN_JSON_ABI_TAGS_CONCAT(                                   \
00113             NLOHMANN_JSON_ABI_TAG_DIAGNOSTICS,                       \
00114             NLOHMANN_JSON_ABI_TAG_LEGACY_DISCARDED_VALUE_COMPARISON, \
00115             NLOHMANN_JSON_ABI_TAG_DIAGNOSTIC_POSITIONS)
00116
00117 // Construct the namespace version component
00118 #define NLOHMANN_JSON_NAMESPACE_VERSION_CONCAT_EX(major, minor, patch) \
00119     _v ## major ## _ ## minor ## _ ## patch
00120 #define NLOHMANN_JSON_NAMESPACE_VERSION_CONCAT(major, minor, patch) \
00121     NLOHMANN_JSON_NAMESPACE_VERSION_CONCAT_EX(major, minor, patch)
00122
00123 #if NLOHMANN_JSON_NAMESPACE_NO_VERSION
00124 #define NLOHMANN_JSON_NAMESPACE_VERSION
00125 #else
00126 #define NLOHMANN_JSON_NAMESPACE_VERSION                              \
00127     NLOHMANN_JSON_NAMESPACE_VERSION_CONCAT(NLOHMANN_JSON_VERSION_MAJOR, \
00128                                            NLOHMANN_JSON_VERSION_MINOR, \
00129                                            NLOHMANN_JSON_VERSION_PATCH)
00130 #endif
00131
00132 // Combine namespace components
00133 #define NLOHMANN_JSON_NAMESPACE_CONCAT_EX(a, b) a ## b
00134 #define NLOHMANN_JSON_NAMESPACE_CONCAT(a, b) \
00135     NLOHMANN_JSON_NAMESPACE_CONCAT_EX(a, b)
00136
00137 #ifndef NLOHMANN_JSON_NAMESPACE
00138 #define NLOHMANN_JSON_NAMESPACE               \
00139     nlohmann::NLOHMANN_JSON_NAMESPACE_CONCAT( \
00140             NLOHMANN_JSON_ABI_TAGS,           \
00141             NLOHMANN_JSON_NAMESPACE_VERSION)
00142 #endif
00143
00144 #ifndef NLOHMANN_JSON_NAMESPACE_BEGIN
00145 #define NLOHMANN_JSON_NAMESPACE_BEGIN                \
00146     namespace nlohmann                               \
00147     {                                                \
00148     inline namespace NLOHMANN_JSON_NAMESPACE_CONCAT( \
00149                 NLOHMANN_JSON_ABI_TAGS,              \
00150                 NLOHMANN_JSON_NAMESPACE_VERSION)     \
00151     {
00152 #endif
00153
00154 #ifndef NLOHMANN_JSON_NAMESPACE_END
00155 #define NLOHMANN_JSON_NAMESPACE_END                                     \
00156     }  /* namespace (inline namespace) NOLINT(readability/namespace) */ \
00157     }  // namespace nlohmann
00158 #endif
00159
00160 // #include <nlohmann/detail/conversions/from_json.hpp>
00161 //     __ _____ _____ _____
00162 //  __|  |   __|     |   | |  JSON for Modern C++
00163 // |  |  |__   |  |  | | | |  version 3.12.0
00164 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
00165 //
00166 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
00167 // SPDX-License-Identifier: MIT
00168
00169
00170
00171 #include <algorithm> // transform
00172 #include <array> // array
00173 #include <forward_list> // forward_list
```

```
00174 #include <iterator> // inserter, front_inserter, end
00175 #include <map> // map
00176 #include <string> // string
00177 #include <tuple> // tuple, make_tuple
00178 #include <type_traits> // is_arithmetic, is_same, is_enum, underlying_type, is_convertible
00179 #include <unordered_map> // unordered_map
00180 #include <utility> // pair, declval
00181 #include <valarray> // valarray
00182
00183 // #include <nlohmann/detail/exceptions.hpp>
00184 //     __ _____ _____ _____
00185 //  __|  |   __|     |   | |  JSON for Modern C++
00186 // |  |  |__   |  |  | | | |  version 3.12.0
00187 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
00188 //
00189 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
00190 // SPDX-License-Identifier: MIT
00191
00192
00193
00194 #include <cstddef> // nullptr_t
00195 #include <exception> // exception
00196 #if JSON_DIAGNOSTICS
00197     #include <numeric> // accumulate
00198 #endif
00199 #include <stdexcept> // runtime_error
00200 #include <string> // to_string
00201 #include <vector> // vector
00202
00203 // #include <nlohmann/detail/value_t.hpp>
00204 //     __ _____ _____ _____
00205 //  __|  |   __|     |   | |  JSON for Modern C++
00206 // |  |  |__   |  |  | | | |  version 3.12.0
00207 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
00208 //
00209 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
00210 // SPDX-License-Identifier: MIT
00211
00212
00213
00214 #include <array> // array
00215 #include <cstddef> // size_t
00216 #include <cstdint> // uint8_t
00217 #include <string> // string
00218
00219 // #include <nlohmann/detail/macro_scope.hpp>
00220 //     __ _____ _____ _____
00221 //  __|  |   __|     |   | |  JSON for Modern C++
00222 // |  |  |__   |  |  | | | |  version 3.12.0
00223 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
00224 //
00225 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
00226 // SPDX-License-Identifier: MIT
00227
00228
00229
00230 #include <utility> // declval, pair
00231 // #include <nlohmann/detail/meta/detected.hpp>
00232 //     __ _____ _____ _____
00233 //  __|  |   __|     |   | |  JSON for Modern C++
00234 // |  |  |__   |  |  | | | |  version 3.12.0
00235 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
00236 //
00237 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
00238 // SPDX-License-Identifier: MIT
00239
00240
00241
00242 #include <type_traits>
00243
00244 // #include <nlohmann/detail/meta/void_t.hpp>
00245 //     __ _____ _____ _____
00246 //  __|  |   __|     |   | |  JSON for Modern C++
00247 // |  |  |__   |  |  | | | |  version 3.12.0
00248 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
00249 //
00250 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
00251 // SPDX-License-Identifier: MIT
00252
00253
00254
00255 // #include <nlohmann/detail/abi_macros.hpp>
00256
00257
00258 NLOHMANN_JSON_NAMESPACE_BEGIN
00259 namespace detail
00260 {
```

```
00261
00262 template<typename ...Ts> struct make_void
00263 {
00264     using type = void;
00265 };
00266 template<typename ...Ts> using void_t = typename make_void<Ts...>::type;
00267
00268 }  // namespace detail
00269 NLOHMANN_JSON_NAMESPACE_END
00270
00271
00272 NLOHMANN_JSON_NAMESPACE_BEGIN
00273 namespace detail
00274 {
00275
00276 // https://en.cppreference.com/w/cpp/experimental/is_detected
00277 struct nonesuch
00278 {
00279     nonesuch() = delete;
00280     ~nonesuch() = delete;
00281     nonesuch(nonesuch const&) = delete;
00282     nonesuch(nonesuch const&&) = delete;
00283     void operator=(nonesuch const&) = delete;
00284     void operator=(nonesuch&&) = delete;
00285 };
00286
00287 template<class Default,
00288          class AlwaysVoid,
00289          template<class...> class Op,
00290          class... Args>
00291 struct detector
00292 {
00293     using value_t = std::false_type;
00294     using type = Default;
00295 };
00296
00297 template<class Default, template<class...> class Op, class... Args>
00298 struct detector<Default, void_t<Op<Args...>>, Op, Args...>
00299 {
00300     using value_t = std::true_type;
00301     using type = Op<Args...>;
00302 };
00303
00304 template<template<class...> class Op, class... Args>
00305 using is_detected = typename detector<nonesuch, void, Op, Args...>::value_t;
00306
00307 template<template<class...> class Op, class... Args>
00308 struct is_detected_lazy : is_detected<Op, Args...> { };
00309
00310 template<template<class...> class Op, class... Args>
00311 using detected_t = typename detector<nonesuch, void, Op, Args...>::type;
00312
00313 template<class Default, template<class...> class Op, class... Args>
00314 using detected_or = detector<Default, void, Op, Args...>;
00315
00316 template<class Default, template<class...> class Op, class... Args>
00317 using detected_or_t = typename detected_or<Default, Op, Args...>::type;
00318
00319 template<class Expected, template<class...> class Op, class... Args>
00320 using is_detected_exact = std::is_same<Expected, detected_t<Op, Args...>>;
00321
00322 template<class To, template<class...> class Op, class... Args>
00323 using is_detected_convertible =
00324     std::is_convertible<detected_t<Op, Args...>, To>;
00325
00326 }  // namespace detail
00327 NLOHMANN_JSON_NAMESPACE_END
00328
00329 // #include <nlohmann/thirdparty/hedley/hedley.hpp>
00330
00331
00332 //      __ _____ _____ _____
00333 //   __|  |   __|     |   | |  JSON for Modern C++
00334 // |  |  |__   |  |  | | | |  version 3.12.0
00335 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
00336 //
00337 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
00338 // SPDX-FileCopyrightText: 2016-2021 Evan Nemerson <evan@nemerson.com>
00339 // SPDX-License-Identifier: MIT
00340
00341 /* Hedley - https://nemequ.github.io/hedley
00342  * Created by Evan Nemerson <evan@nemerson.com>
00343  * SPDX-License-Identifier: CC0-1.0
00344  */
00345
00346 #if !defined(JSON_HEDLEY_VERSION) || (JSON_HEDLEY_VERSION < 15)
00347 #if defined(JSON_HEDLEY_VERSION)
```

```
00348     #undef JSON_HEDLEY_VERSION
00349 #endif
00350 #define JSON_HEDLEY_VERSION 15
00351
00352 #if defined(JSON_HEDLEY_STRINGIFY_EX)
00353     #undef JSON_HEDLEY_STRINGIFY_EX
00354 #endif
00355 #define JSON_HEDLEY_STRINGIFY_EX(x) #x
00356
00357 #if defined(JSON_HEDLEY_STRINGIFY)
00358     #undef JSON_HEDLEY_STRINGIFY
00359 #endif
00360 #define JSON_HEDLEY_STRINGIFY(x) JSON_HEDLEY_STRINGIFY_EX(x)
00361
00362 #if defined(JSON_HEDLEY_CONCAT_EX)
00363     #undef JSON_HEDLEY_CONCAT_EX
00364 #endif
00365 #define JSON_HEDLEY_CONCAT_EX(a,b) a##b
00366
00367 #if defined(JSON_HEDLEY_CONCAT)
00368     #undef JSON_HEDLEY_CONCAT
00369 #endif
00370 #define JSON_HEDLEY_CONCAT(a,b) JSON_HEDLEY_CONCAT_EX(a,b)
00371
00372 #if defined(JSON_HEDLEY_CONCAT3_EX)
00373     #undef JSON_HEDLEY_CONCAT3_EX
00374 #endif
00375 #define JSON_HEDLEY_CONCAT3_EX(a,b,c) a##b##c
00376
00377 #if defined(JSON_HEDLEY_CONCAT3)
00378     #undef JSON_HEDLEY_CONCAT3
00379 #endif
00380 #define JSON_HEDLEY_CONCAT3(a,b,c) JSON_HEDLEY_CONCAT3_EX(a,b,c)
00381
00382 #if defined(JSON_HEDLEY_VERSION_ENCODE)
00383     #undef JSON_HEDLEY_VERSION_ENCODE
00384 #endif
00385 #define JSON_HEDLEY_VERSION_ENCODE(major,minor,revision) (((major) * 1000000) + ((minor) * 1000) +
    (revision))
00386
00387 #if defined(JSON_HEDLEY_VERSION_DECODE_MAJOR)
00388     #undef JSON_HEDLEY_VERSION_DECODE_MAJOR
00389 #endif
00390 #define JSON_HEDLEY_VERSION_DECODE_MAJOR(version) ((version) / 1000000)
00391
00392 #if defined(JSON_HEDLEY_VERSION_DECODE_MINOR)
00393     #undef JSON_HEDLEY_VERSION_DECODE_MINOR
00394 #endif
00395 #define JSON_HEDLEY_VERSION_DECODE_MINOR(version) (((version) % 1000000) / 1000)
00396
00397 #if defined(JSON_HEDLEY_VERSION_DECODE_REVISION)
00398     #undef JSON_HEDLEY_VERSION_DECODE_REVISION
00399 #endif
00400 #define JSON_HEDLEY_VERSION_DECODE_REVISION(version) ((version) % 1000)
00401
00402 #if defined(JSON_HEDLEY_GNUC_VERSION)
00403     #undef JSON_HEDLEY_GNUC_VERSION
00404 #endif
00405 #if defined(__GNUC__) && defined(__GNUC_PATCHLEVEL__)
00406     #define JSON_HEDLEY_GNUC_VERSION JSON_HEDLEY_VERSION_ENCODE(__GNUC__, __GNUC_MINOR__,
    __GNUC_PATCHLEVEL__)
00407 #elif defined(__GNUC__)
00408     #define JSON_HEDLEY_GNUC_VERSION JSON_HEDLEY_VERSION_ENCODE(__GNUC__, __GNUC_MINOR__, 0)
00409 #endif
00410
00411 #if defined(JSON_HEDLEY_GNUC_VERSION_CHECK)
00412     #undef JSON_HEDLEY_GNUC_VERSION_CHECK
00413 #endif
00414 #if defined(JSON_HEDLEY_GNUC_VERSION)
00415     #define JSON_HEDLEY_GNUC_VERSION_CHECK(major,minor,patch) (JSON_HEDLEY_GNUC_VERSION >=
    JSON_HEDLEY_VERSION_ENCODE(major, minor, patch))
00416 #else
00417     #define JSON_HEDLEY_GNUC_VERSION_CHECK(major,minor,patch) (0)
00418 #endif
00419
00420 #if defined(JSON_HEDLEY_MSVC_VERSION)
00421     #undef JSON_HEDLEY_MSVC_VERSION
00422 #endif
00423 #if defined(_MSC_FULL_VER) && (_MSC_FULL_VER >= 140000000) && !defined(__ICL)
00424     #define JSON_HEDLEY_MSVC_VERSION JSON_HEDLEY_VERSION_ENCODE(_MSC_FULL_VER / 10000000,
    (_MSC_FULL_VER % 10000000) / 100000, (_MSC_FULL_VER % 100000) / 100)
00425 #elif defined(_MSC_FULL_VER) && !defined(__ICL)
00426     #define JSON_HEDLEY_MSVC_VERSION JSON_HEDLEY_VERSION_ENCODE(_MSC_FULL_VER / 1000000,
    (_MSC_FULL_VER % 1000000) / 10000, (_MSC_FULL_VER % 10000) / 10)
00427 #elif defined(_MSC_VER) && !defined(__ICL)
00428     #define JSON_HEDLEY_MSVC_VERSION JSON_HEDLEY_VERSION_ENCODE(_MSC_VER / 100, _MSC_VER % 100, 0)
00429 #endif
```

```
00430
00431 #if defined(JSON_HEDLEY_MSVC_VERSION_CHECK)
00432     #undef JSON_HEDLEY_MSVC_VERSION_CHECK
00433 #endif
00434 #if !defined(JSON_HEDLEY_MSVC_VERSION)
00435     #define JSON_HEDLEY_MSVC_VERSION_CHECK(major,minor,patch) (0)
00436 #elif defined(_MSC_VER) && (_MSC_VER >= 1400)
00437     #define JSON_HEDLEY_MSVC_VERSION_CHECK(major,minor,patch) (_MSC_FULL_VER >= ((major * 10000000) +
      (minor * 100000) + (patch)))
00438 #elif defined(_MSC_VER) && (_MSC_VER >= 1200)
00439     #define JSON_HEDLEY_MSVC_VERSION_CHECK(major,minor,patch) (_MSC_FULL_VER >= ((major * 1000000) +
      (minor * 10000) + (patch)))
00440 #else
00441     #define JSON_HEDLEY_MSVC_VERSION_CHECK(major,minor,patch) (_MSC_VER >= ((major * 100) + (minor)))
00442 #endif
00443
00444 #if defined(JSON_HEDLEY_INTEL_VERSION)
00445     #undef JSON_HEDLEY_INTEL_VERSION
00446 #endif
00447 #if defined(__INTEL_COMPILER) && defined(__INTEL_COMPILER_UPDATE) && !defined(__ICL)
00448     #define JSON_HEDLEY_INTEL_VERSION JSON_HEDLEY_VERSION_ENCODE(__INTEL_COMPILER / 100,
      __INTEL_COMPILER % 100, __INTEL_COMPILER_UPDATE)
00449 #elif defined(__INTEL_COMPILER) && !defined(__ICL)
00450     #define JSON_HEDLEY_INTEL_VERSION JSON_HEDLEY_VERSION_ENCODE(__INTEL_COMPILER / 100,
      __INTEL_COMPILER % 100, 0)
00451 #endif
00452
00453 #if defined(JSON_HEDLEY_INTEL_VERSION_CHECK)
00454     #undef JSON_HEDLEY_INTEL_VERSION_CHECK
00455 #endif
00456 #if defined(JSON_HEDLEY_INTEL_VERSION)
00457     #define JSON_HEDLEY_INTEL_VERSION_CHECK(major,minor,patch) (JSON_HEDLEY_INTEL_VERSION >=
      JSON_HEDLEY_VERSION_ENCODE(major, minor, patch))
00458 #else
00459     #define JSON_HEDLEY_INTEL_VERSION_CHECK(major,minor,patch) (0)
00460 #endif
00461
00462 #if defined(JSON_HEDLEY_INTEL_CL_VERSION)
00463     #undef JSON_HEDLEY_INTEL_CL_VERSION
00464 #endif
00465 #if defined(__INTEL_COMPILER) && defined(__INTEL_COMPILER_UPDATE) && defined(__ICL)
00466     #define JSON_HEDLEY_INTEL_CL_VERSION JSON_HEDLEY_VERSION_ENCODE(__INTEL_COMPILER,
      __INTEL_COMPILER_UPDATE, 0)
00467 #endif
00468
00469 #if defined(JSON_HEDLEY_INTEL_CL_VERSION_CHECK)
00470     #undef JSON_HEDLEY_INTEL_CL_VERSION_CHECK
00471 #endif
00472 #if defined(JSON_HEDLEY_INTEL_CL_VERSION)
00473     #define JSON_HEDLEY_INTEL_CL_VERSION_CHECK(major,minor,patch) (JSON_HEDLEY_INTEL_CL_VERSION >=
      JSON_HEDLEY_VERSION_ENCODE(major, minor, patch))
00474 #else
00475     #define JSON_HEDLEY_INTEL_CL_VERSION_CHECK(major,minor,patch) (0)
00476 #endif
00477
00478 #if defined(JSON_HEDLEY_PGI_VERSION)
00479     #undef JSON_HEDLEY_PGI_VERSION
00480 #endif
00481 #if defined(__PGI) && defined(__PGIC__) && defined(__PGIC_MINOR__) && defined(__PGIC_PATCHLEVEL__)
00482     #define JSON_HEDLEY_PGI_VERSION JSON_HEDLEY_VERSION_ENCODE(__PGIC__, __PGIC_MINOR__,
      __PGIC_PATCHLEVEL__)
00483 #endif
00484
00485 #if defined(JSON_HEDLEY_PGI_VERSION_CHECK)
00486     #undef JSON_HEDLEY_PGI_VERSION_CHECK
00487 #endif
00488 #if defined(JSON_HEDLEY_PGI_VERSION)
00489     #define JSON_HEDLEY_PGI_VERSION_CHECK(major,minor,patch) (JSON_HEDLEY_PGI_VERSION >=
      JSON_HEDLEY_VERSION_ENCODE(major, minor, patch))
00490 #else
00491     #define JSON_HEDLEY_PGI_VERSION_CHECK(major,minor,patch) (0)
00492 #endif
00493
00494 #if defined(JSON_HEDLEY_SUNPRO_VERSION)
00495     #undef JSON_HEDLEY_SUNPRO_VERSION
00496 #endif
00497 #if defined(__SUNPRO_C) && (__SUNPRO_C > 0x1000)
00498     #define JSON_HEDLEY_SUNPRO_VERSION JSON_HEDLEY_VERSION_ENCODE((((__SUNPRO_C » 16) & 0xf) * 10) +
      ((__SUNPRO_C » 12) & 0xf), (((__SUNPRO_C » 8) & 0xf) * 10) + ((__SUNPRO_C » 4) & 0xf), (__SUNPRO_C &
      0xf) * 10)
00499 #elif defined(__SUNPRO_C)
00500     #define JSON_HEDLEY_SUNPRO_VERSION JSON_HEDLEY_VERSION_ENCODE((__SUNPRO_C » 8) & 0xf, (__SUNPRO_C
      » 4) & 0xf, (__SUNPRO_C) & 0xf)
00501 #elif defined(__SUNPRO_CC) && (__SUNPRO_CC > 0x1000)
00502     #define JSON_HEDLEY_SUNPRO_VERSION JSON_HEDLEY_VERSION_ENCODE((((__SUNPRO_CC » 16) & 0xf) * 10) +
      ((__SUNPRO_CC » 12) & 0xf), (((__SUNPRO_CC » 8) & 0xf) * 10) + ((__SUNPRO_CC » 4) & 0xf), (__SUNPRO_CC
      & 0xf) * 10)
```

```
00503 #elif defined(__SUNPRO_CC)
00504     #define JSON_HEDLEY_SUNPRO_VERSION JSON_HEDLEY_VERSION_ENCODE((__SUNPRO_CC » 8) & 0xf,
      (__SUNPRO_CC » 4) & 0xf, (__SUNPRO_CC) & 0xf)
00505 #endif
00506
00507 #if defined(JSON_HEDLEY_SUNPRO_VERSION_CHECK)
00508     #undef JSON_HEDLEY_SUNPRO_VERSION_CHECK
00509 #endif
00510 #if defined(JSON_HEDLEY_SUNPRO_VERSION)
00511     #define JSON_HEDLEY_SUNPRO_VERSION_CHECK(major,minor,patch) (JSON_HEDLEY_SUNPRO_VERSION >=
      JSON_HEDLEY_VERSION_ENCODE(major, minor, patch))
00512 #else
00513     #define JSON_HEDLEY_SUNPRO_VERSION_CHECK(major,minor,patch) (0)
00514 #endif
00515
00516 #if defined(JSON_HEDLEY_EMSCRIPTEN_VERSION)
00517     #undef JSON_HEDLEY_EMSCRIPTEN_VERSION
00518 #endif
00519 #if defined(__EMSCRIPTEN__)
00520     #define JSON_HEDLEY_EMSCRIPTEN_VERSION JSON_HEDLEY_VERSION_ENCODE(__EMSCRIPTEN_major__,
      __EMSCRIPTEN_minor__, __EMSCRIPTEN_tiny__)
00521 #endif
00522
00523 #if defined(JSON_HEDLEY_EMSCRIPTEN_VERSION_CHECK)
00524     #undef JSON_HEDLEY_EMSCRIPTEN_VERSION_CHECK
00525 #endif
00526 #if defined(JSON_HEDLEY_EMSCRIPTEN_VERSION)
00527     #define JSON_HEDLEY_EMSCRIPTEN_VERSION_CHECK(major,minor,patch) (JSON_HEDLEY_EMSCRIPTEN_VERSION >=
      JSON_HEDLEY_VERSION_ENCODE(major, minor, patch))
00528 #else
00529     #define JSON_HEDLEY_EMSCRIPTEN_VERSION_CHECK(major,minor,patch) (0)
00530 #endif
00531
00532 #if defined(JSON_HEDLEY_ARM_VERSION)
00533     #undef JSON_HEDLEY_ARM_VERSION
00534 #endif
00535 #if defined(__CC_ARM) && defined(__ARMCOMPILER_VERSION)
00536     #define JSON_HEDLEY_ARM_VERSION JSON_HEDLEY_VERSION_ENCODE(__ARMCOMPILER_VERSION / 1000000,
      (__ARMCOMPILER_VERSION % 1000000) / 10000, (__ARMCOMPILER_VERSION % 10000) / 100)
00537 #elif defined(__CC_ARM) && defined(__ARMCC_VERSION)
00538     #define JSON_HEDLEY_ARM_VERSION JSON_HEDLEY_VERSION_ENCODE(__ARMCC_VERSION / 1000000,
      (__ARMCC_VERSION % 1000000) / 10000, (__ARMCC_VERSION % 10000) / 100)
00539 #endif
00540
00541 #if defined(JSON_HEDLEY_ARM_VERSION_CHECK)
00542     #undef JSON_HEDLEY_ARM_VERSION_CHECK
00543 #endif
00544 #if defined(JSON_HEDLEY_ARM_VERSION)
00545     #define JSON_HEDLEY_ARM_VERSION_CHECK(major,minor,patch) (JSON_HEDLEY_ARM_VERSION >=
      JSON_HEDLEY_VERSION_ENCODE(major, minor, patch))
00546 #else
00547     #define JSON_HEDLEY_ARM_VERSION_CHECK(major,minor,patch) (0)
00548 #endif
00549
00550 #if defined(JSON_HEDLEY_IBM_VERSION)
00551     #undef JSON_HEDLEY_IBM_VERSION
00552 #endif
00553 #if defined(__ibmxl__)
00554     #define JSON_HEDLEY_IBM_VERSION JSON_HEDLEY_VERSION_ENCODE(__ibmxl_version__, __ibmxl_release__,
      __ibmxl_modification__)
00555 #elif defined(__xlC__) && defined(__xlC_ver__)
00556     #define JSON_HEDLEY_IBM_VERSION JSON_HEDLEY_VERSION_ENCODE(__xlC__ » 8, __xlC__ & 0xff,
      (__xlC_ver__ » 8) & 0xff)
00557 #elif defined(__xlC__)
00558     #define JSON_HEDLEY_IBM_VERSION JSON_HEDLEY_VERSION_ENCODE(__xlC__ » 8, __xlC__ & 0xff, 0)
00559 #endif
00560
00561 #if defined(JSON_HEDLEY_IBM_VERSION_CHECK)
00562     #undef JSON_HEDLEY_IBM_VERSION_CHECK
00563 #endif
00564 #if defined(JSON_HEDLEY_IBM_VERSION)
00565     #define JSON_HEDLEY_IBM_VERSION_CHECK(major,minor,patch) (JSON_HEDLEY_IBM_VERSION >=
      JSON_HEDLEY_VERSION_ENCODE(major, minor, patch))
00566 #else
00567     #define JSON_HEDLEY_IBM_VERSION_CHECK(major,minor,patch) (0)
00568 #endif
00569
00570 #if defined(JSON_HEDLEY_TI_VERSION)
00571     #undef JSON_HEDLEY_TI_VERSION
00572 #endif
00573 #if \
00574     defined(__TI_COMPILER_VERSION__) && \
00575     ( \
00576       defined(__TMS470__) || defined(__TI_ARM__) || \
00577       defined(__MSP430__) || \
00578       defined(__TMS320C2000__) \
00579     )
```

```
00580 #if (__TI_COMPILER_VERSION__ >= 16000000)
00581     #define JSON_HEDLEY_TI_VERSION JSON_HEDLEY_VERSION_ENCODE(__TI_COMPILER_VERSION__ / 1000000,
      (__TI_COMPILER_VERSION__ % 1000000) / 1000, (__TI_COMPILER_VERSION__ % 1000))
00582 #endif
00583 #endif
00584
00585 #if defined(JSON_HEDLEY_TI_VERSION_CHECK)
00586     #undef JSON_HEDLEY_TI_VERSION_CHECK
00587 #endif
00588 #if defined(JSON_HEDLEY_TI_VERSION)
00589     #define JSON_HEDLEY_TI_VERSION_CHECK(major,minor,patch) (JSON_HEDLEY_TI_VERSION >=
      JSON_HEDLEY_VERSION_ENCODE(major, minor, patch))
00590 #else
00591     #define JSON_HEDLEY_TI_VERSION_CHECK(major,minor,patch) (0)
00592 #endif
00593
00594 #if defined(JSON_HEDLEY_TI_CL2000_VERSION)
00595     #undef JSON_HEDLEY_TI_CL2000_VERSION
00596 #endif
00597 #if defined(__TI_COMPILER_VERSION__) && defined(__TMS320C2000__)
00598     #define JSON_HEDLEY_TI_CL2000_VERSION JSON_HEDLEY_VERSION_ENCODE(__TI_COMPILER_VERSION__ /
      1000000, (__TI_COMPILER_VERSION__ % 1000000) / 1000, (__TI_COMPILER_VERSION__ % 1000))
00599 #endif
00600
00601 #if defined(JSON_HEDLEY_TI_CL2000_VERSION_CHECK)
00602     #undef JSON_HEDLEY_TI_CL2000_VERSION_CHECK
00603 #endif
00604 #if defined(JSON_HEDLEY_TI_CL2000_VERSION)
00605     #define JSON_HEDLEY_TI_CL2000_VERSION_CHECK(major,minor,patch) (JSON_HEDLEY_TI_CL2000_VERSION >=
      JSON_HEDLEY_VERSION_ENCODE(major, minor, patch))
00606 #else
00607     #define JSON_HEDLEY_TI_CL2000_VERSION_CHECK(major,minor,patch) (0)
00608 #endif
00609
00610 #if defined(JSON_HEDLEY_TI_CL430_VERSION)
00611     #undef JSON_HEDLEY_TI_CL430_VERSION
00612 #endif
00613 #if defined(__TI_COMPILER_VERSION__) && defined(__MSP430__)
00614     #define JSON_HEDLEY_TI_CL430_VERSION JSON_HEDLEY_VERSION_ENCODE(__TI_COMPILER_VERSION__ / 1000000,
      (__TI_COMPILER_VERSION__ % 1000000) / 1000, (__TI_COMPILER_VERSION__ % 1000))
00615 #endif
00616
00617 #if defined(JSON_HEDLEY_TI_CL430_VERSION_CHECK)
00618     #undef JSON_HEDLEY_TI_CL430_VERSION_CHECK
00619 #endif
00620 #if defined(JSON_HEDLEY_TI_CL430_VERSION)
00621     #define JSON_HEDLEY_TI_CL430_VERSION_CHECK(major,minor,patch) (JSON_HEDLEY_TI_CL430_VERSION >=
      JSON_HEDLEY_VERSION_ENCODE(major, minor, patch))
00622 #else
00623     #define JSON_HEDLEY_TI_CL430_VERSION_CHECK(major,minor,patch) (0)
00624 #endif
00625
00626 #if defined(JSON_HEDLEY_TI_ARMCL_VERSION)
00627     #undef JSON_HEDLEY_TI_ARMCL_VERSION
00628 #endif
00629 #if defined(__TI_COMPILER_VERSION__) && (defined(__TMS470__) || defined(__TI_ARM__))
00630     #define JSON_HEDLEY_TI_ARMCL_VERSION JSON_HEDLEY_VERSION_ENCODE(__TI_COMPILER_VERSION__ / 1000000,
      (__TI_COMPILER_VERSION__ % 1000000) / 1000, (__TI_COMPILER_VERSION__ % 1000))
00631 #endif
00632
00633 #if defined(JSON_HEDLEY_TI_ARMCL_VERSION_CHECK)
00634     #undef JSON_HEDLEY_TI_ARMCL_VERSION_CHECK
00635 #endif
00636 #if defined(JSON_HEDLEY_TI_ARMCL_VERSION)
00637     #define JSON_HEDLEY_TI_ARMCL_VERSION_CHECK(major,minor,patch) (JSON_HEDLEY_TI_ARMCL_VERSION >=
      JSON_HEDLEY_VERSION_ENCODE(major, minor, patch))
00638 #else
00639     #define JSON_HEDLEY_TI_ARMCL_VERSION_CHECK(major,minor,patch) (0)
00640 #endif
00641
00642 #if defined(JSON_HEDLEY_TI_CL6X_VERSION)
00643     #undef JSON_HEDLEY_TI_CL6X_VERSION
00644 #endif
00645 #if defined(__TI_COMPILER_VERSION__) && defined(__TMS320C6X__)
00646     #define JSON_HEDLEY_TI_CL6X_VERSION JSON_HEDLEY_VERSION_ENCODE(__TI_COMPILER_VERSION__ / 1000000,
      (__TI_COMPILER_VERSION__ % 1000000) / 1000, (__TI_COMPILER_VERSION__ % 1000))
00647 #endif
00648
00649 #if defined(JSON_HEDLEY_TI_CL6X_VERSION_CHECK)
00650     #undef JSON_HEDLEY_TI_CL6X_VERSION_CHECK
00651 #endif
00652 #if defined(JSON_HEDLEY_TI_CL6X_VERSION)
00653     #define JSON_HEDLEY_TI_CL6X_VERSION_CHECK(major,minor,patch) (JSON_HEDLEY_TI_CL6X_VERSION >=
      JSON_HEDLEY_VERSION_ENCODE(major, minor, patch))
00654 #else
00655     #define JSON_HEDLEY_TI_CL6X_VERSION_CHECK(major,minor,patch) (0)
00656 #endif
```

```
00657
00658 #if defined(JSON_HEDLEY_TI_CL7X_VERSION)
00659     #undef JSON_HEDLEY_TI_CL7X_VERSION
00660 #endif
00661 #if defined(__TI_COMPILER_VERSION__) && defined(__C7000__)
00662     #define JSON_HEDLEY_TI_CL7X_VERSION JSON_HEDLEY_VERSION_ENCODE(__TI_COMPILER_VERSION__ / 1000000,
      (__TI_COMPILER_VERSION__ % 1000000) / 1000, (__TI_COMPILER_VERSION__ % 1000))
00663 #endif
00664
00665 #if defined(JSON_HEDLEY_TI_CL7X_VERSION_CHECK)
00666     #undef JSON_HEDLEY_TI_CL7X_VERSION_CHECK
00667 #endif
00668 #if defined(JSON_HEDLEY_TI_CL7X_VERSION)
00669     #define JSON_HEDLEY_TI_CL7X_VERSION_CHECK(major,minor,patch) (JSON_HEDLEY_TI_CL7X_VERSION >=
      JSON_HEDLEY_VERSION_ENCODE(major, minor, patch))
00670 #else
00671     #define JSON_HEDLEY_TI_CL7X_VERSION_CHECK(major,minor,patch) (0)
00672 #endif
00673
00674 #if defined(JSON_HEDLEY_TI_CLPRU_VERSION)
00675     #undef JSON_HEDLEY_TI_CLPRU_VERSION
00676 #endif
00677 #if defined(__TI_COMPILER_VERSION__) && defined(__PRU__)
00678     #define JSON_HEDLEY_TI_CLPRU_VERSION JSON_HEDLEY_VERSION_ENCODE(__TI_COMPILER_VERSION__ / 1000000,
      (__TI_COMPILER_VERSION__ % 1000000) / 1000, (__TI_COMPILER_VERSION__ % 1000))
00679 #endif
00680
00681 #if defined(JSON_HEDLEY_TI_CLPRU_VERSION_CHECK)
00682     #undef JSON_HEDLEY_TI_CLPRU_VERSION_CHECK
00683 #endif
00684 #if defined(JSON_HEDLEY_TI_CLPRU_VERSION)
00685     #define JSON_HEDLEY_TI_CLPRU_VERSION_CHECK(major,minor,patch) (JSON_HEDLEY_TI_CLPRU_VERSION >=
      JSON_HEDLEY_VERSION_ENCODE(major, minor, patch))
00686 #else
00687     #define JSON_HEDLEY_TI_CLPRU_VERSION_CHECK(major,minor,patch) (0)
00688 #endif
00689
00690 #if defined(JSON_HEDLEY_CRAY_VERSION)
00691     #undef JSON_HEDLEY_CRAY_VERSION
00692 #endif
00693 #if defined(_CRAYC)
00694     #if defined(_RELEASE_PATCHLEVEL)
00695         #define JSON_HEDLEY_CRAY_VERSION JSON_HEDLEY_VERSION_ENCODE(_RELEASE_MAJOR, _RELEASE_MINOR,
      _RELEASE_PATCHLEVEL)
00696     #else
00697         #define JSON_HEDLEY_CRAY_VERSION JSON_HEDLEY_VERSION_ENCODE(_RELEASE_MAJOR, _RELEASE_MINOR, 0)
00698     #endif
00699 #endif
00700
00701 #if defined(JSON_HEDLEY_CRAY_VERSION_CHECK)
00702     #undef JSON_HEDLEY_CRAY_VERSION_CHECK
00703 #endif
00704 #if defined(JSON_HEDLEY_CRAY_VERSION)
00705     #define JSON_HEDLEY_CRAY_VERSION_CHECK(major,minor,patch) (JSON_HEDLEY_CRAY_VERSION >=
      JSON_HEDLEY_VERSION_ENCODE(major, minor, patch))
00706 #else
00707     #define JSON_HEDLEY_CRAY_VERSION_CHECK(major,minor,patch) (0)
00708 #endif
00709
00710 #if defined(JSON_HEDLEY_IAR_VERSION)
00711     #undef JSON_HEDLEY_IAR_VERSION
00712 #endif
00713 #if defined(__IAR_SYSTEMS_ICC__)
00714     #if __VER__ > 1000
00715         #define JSON_HEDLEY_IAR_VERSION JSON_HEDLEY_VERSION_ENCODE((__VER__ / 1000000), ((__VER__ /
      1000) % 1000), (__VER__ % 1000))
00716     #else
00717         #define JSON_HEDLEY_IAR_VERSION JSON_HEDLEY_VERSION_ENCODE(__VER__ / 100, __VER__ % 100, 0)
00718     #endif
00719 #endif
00720
00721 #if defined(JSON_HEDLEY_IAR_VERSION_CHECK)
00722     #undef JSON_HEDLEY_IAR_VERSION_CHECK
00723 #endif
00724 #if defined(JSON_HEDLEY_IAR_VERSION)
00725     #define JSON_HEDLEY_IAR_VERSION_CHECK(major,minor,patch) (JSON_HEDLEY_IAR_VERSION >=
      JSON_HEDLEY_VERSION_ENCODE(major, minor, patch))
00726 #else
00727     #define JSON_HEDLEY_IAR_VERSION_CHECK(major,minor,patch) (0)
00728 #endif
00729
00730 #if defined(JSON_HEDLEY_TINYC_VERSION)
00731     #undef JSON_HEDLEY_TINYC_VERSION
00732 #endif
00733 #if defined(__TINYC__)
00734     #define JSON_HEDLEY_TINYC_VERSION JSON_HEDLEY_VERSION_ENCODE(__TINYC__ / 1000, (__TINYC__ / 100) %
      10, __TINYC__ % 100)
```

```
00735 #endif
00736
00737 #if defined(JSON_HEDLEY_TINYC_VERSION_CHECK)
00738     #undef JSON_HEDLEY_TINYC_VERSION_CHECK
00739 #endif
00740 #if defined(JSON_HEDLEY_TINYC_VERSION)
00741     #define JSON_HEDLEY_TINYC_VERSION_CHECK(major,minor,patch) (JSON_HEDLEY_TINYC_VERSION >=
      JSON_HEDLEY_VERSION_ENCODE(major, minor, patch))
00742 #else
00743     #define JSON_HEDLEY_TINYC_VERSION_CHECK(major,minor,patch) (0)
00744 #endif
00745
00746 #if defined(JSON_HEDLEY_DMC_VERSION)
00747     #undef JSON_HEDLEY_DMC_VERSION
00748 #endif
00749 #if defined(__DMC__)
00750     #define JSON_HEDLEY_DMC_VERSION JSON_HEDLEY_VERSION_ENCODE(__DMC__ » 8, (__DMC__ » 4) & 0xf,
      __DMC__ & 0xf)
00751 #endif
00752
00753 #if defined(JSON_HEDLEY_DMC_VERSION_CHECK)
00754     #undef JSON_HEDLEY_DMC_VERSION_CHECK
00755 #endif
00756 #if defined(JSON_HEDLEY_DMC_VERSION)
00757     #define JSON_HEDLEY_DMC_VERSION_CHECK(major,minor,patch) (JSON_HEDLEY_DMC_VERSION >=
      JSON_HEDLEY_VERSION_ENCODE(major, minor, patch))
00758 #else
00759     #define JSON_HEDLEY_DMC_VERSION_CHECK(major,minor,patch) (0)
00760 #endif
00761
00762 #if defined(JSON_HEDLEY_COMPCERT_VERSION)
00763     #undef JSON_HEDLEY_COMPCERT_VERSION
00764 #endif
00765 #if defined(__COMPCERT_VERSION__)
00766     #define JSON_HEDLEY_COMPCERT_VERSION JSON_HEDLEY_VERSION_ENCODE(__COMPCERT_VERSION__ / 10000,
      (__COMPCERT_VERSION__ / 100) % 100, __COMPCERT_VERSION__ % 100)
00767 #endif
00768
00769 #if defined(JSON_HEDLEY_COMPCERT_VERSION_CHECK)
00770     #undef JSON_HEDLEY_COMPCERT_VERSION_CHECK
00771 #endif
00772 #if defined(JSON_HEDLEY_COMPCERT_VERSION)
00773     #define JSON_HEDLEY_COMPCERT_VERSION_CHECK(major,minor,patch) (JSON_HEDLEY_COMPCERT_VERSION >=
      JSON_HEDLEY_VERSION_ENCODE(major, minor, patch))
00774 #else
00775     #define JSON_HEDLEY_COMPCERT_VERSION_CHECK(major,minor,patch) (0)
00776 #endif
00777
00778 #if defined(JSON_HEDLEY_PELLES_VERSION)
00779     #undef JSON_HEDLEY_PELLES_VERSION
00780 #endif
00781 #if defined(__POCC__)
00782     #define JSON_HEDLEY_PELLES_VERSION JSON_HEDLEY_VERSION_ENCODE(__POCC__ / 100, __POCC__ % 100, 0)
00783 #endif
00784
00785 #if defined(JSON_HEDLEY_PELLES_VERSION_CHECK)
00786     #undef JSON_HEDLEY_PELLES_VERSION_CHECK
00787 #endif
00788 #if defined(JSON_HEDLEY_PELLES_VERSION)
00789     #define JSON_HEDLEY_PELLES_VERSION_CHECK(major,minor,patch) (JSON_HEDLEY_PELLES_VERSION >=
      JSON_HEDLEY_VERSION_ENCODE(major, minor, patch))
00790 #else
00791     #define JSON_HEDLEY_PELLES_VERSION_CHECK(major,minor,patch) (0)
00792 #endif
00793
00794 #if defined(JSON_HEDLEY_MCST_LCC_VERSION)
00795     #undef JSON_HEDLEY_MCST_LCC_VERSION
00796 #endif
00797 #if defined(__LCC__) && defined(__LCC_MINOR__)
00798     #define JSON_HEDLEY_MCST_LCC_VERSION JSON_HEDLEY_VERSION_ENCODE(__LCC__ / 100, __LCC__ % 100,
      __LCC_MINOR__)
00799 #endif
00800
00801 #if defined(JSON_HEDLEY_MCST_LCC_VERSION_CHECK)
00802     #undef JSON_HEDLEY_MCST_LCC_VERSION_CHECK
00803 #endif
00804 #if defined(JSON_HEDLEY_MCST_LCC_VERSION)
00805     #define JSON_HEDLEY_MCST_LCC_VERSION_CHECK(major,minor,patch) (JSON_HEDLEY_MCST_LCC_VERSION >=
      JSON_HEDLEY_VERSION_ENCODE(major, minor, patch))
00806 #else
00807     #define JSON_HEDLEY_MCST_LCC_VERSION_CHECK(major,minor,patch) (0)
00808 #endif
00809
00810 #if defined(JSON_HEDLEY_GCC_VERSION)
00811     #undef JSON_HEDLEY_GCC_VERSION
00812 #endif
00813 #if \
```

```
00814     defined(JSON_HEDLEY_GNUC_VERSION) && \
00815     !defined(__clang__) && \
00816     !defined(JSON_HEDLEY_INTEL_VERSION) && \
00817     !defined(JSON_HEDLEY_PGI_VERSION) && \
00818     !defined(JSON_HEDLEY_ARM_VERSION) && \
00819     !defined(JSON_HEDLEY_CRAY_VERSION) && \
00820     !defined(JSON_HEDLEY_TI_VERSION) && \
00821     !defined(JSON_HEDLEY_TI_ARMCL_VERSION) && \
00822     !defined(JSON_HEDLEY_TI_CL430_VERSION) && \
00823     !defined(JSON_HEDLEY_TI_CL2000_VERSION) && \
00824     !defined(JSON_HEDLEY_TI_CL6X_VERSION) && \
00825     !defined(JSON_HEDLEY_TI_CL7X_VERSION) && \
00826     !defined(JSON_HEDLEY_TI_CLPRU_VERSION) && \
00827     !defined(__COMPCERT__) && \
00828     !defined(JSON_HEDLEY_MCST_LCC_VERSION)
00829     #define JSON_HEDLEY_GCC_VERSION JSON_HEDLEY_GNUC_VERSION
00830 #endif
00831
00832 #if defined(JSON_HEDLEY_GCC_VERSION_CHECK)
00833     #undef JSON_HEDLEY_GCC_VERSION_CHECK
00834 #endif
00835 #if defined(JSON_HEDLEY_GCC_VERSION)
00836     #define JSON_HEDLEY_GCC_VERSION_CHECK(major,minor,patch) (JSON_HEDLEY_GCC_VERSION >=
      JSON_HEDLEY_VERSION_ENCODE(major, minor, patch))
00837 #else
00838     #define JSON_HEDLEY_GCC_VERSION_CHECK(major,minor,patch) (0)
00839 #endif
00840
00841 #if defined(JSON_HEDLEY_HAS_ATTRIBUTE)
00842     #undef JSON_HEDLEY_HAS_ATTRIBUTE
00843 #endif
00844 #if \
00845   defined(__has_attribute) && \
00846   ( \
00847     (!defined(JSON_HEDLEY_IAR_VERSION) || JSON_HEDLEY_IAR_VERSION_CHECK(8,5,9)) \
00848   )
00849 #  define JSON_HEDLEY_HAS_ATTRIBUTE(attribute) __has_attribute(attribute)
00850 #else
00851 #  define JSON_HEDLEY_HAS_ATTRIBUTE(attribute) (0)
00852 #endif
00853
00854 #if defined(JSON_HEDLEY_GNUC_HAS_ATTRIBUTE)
00855     #undef JSON_HEDLEY_GNUC_HAS_ATTRIBUTE
00856 #endif
00857 #if defined(__has_attribute)
00858     #define JSON_HEDLEY_GNUC_HAS_ATTRIBUTE(attribute,major,minor,patch)
      JSON_HEDLEY_HAS_ATTRIBUTE(attribute)
00859 #else
00860     #define JSON_HEDLEY_GNUC_HAS_ATTRIBUTE(attribute,major,minor,patch)
      JSON_HEDLEY_GNUC_VERSION_CHECK(major,minor,patch)
00861 #endif
00862
00863 #if defined(JSON_HEDLEY_GCC_HAS_ATTRIBUTE)
00864     #undef JSON_HEDLEY_GCC_HAS_ATTRIBUTE
00865 #endif
00866 #if defined(__has_attribute)
00867     #define JSON_HEDLEY_GCC_HAS_ATTRIBUTE(attribute,major,minor,patch)
      JSON_HEDLEY_HAS_ATTRIBUTE(attribute)
00868 #else
00869     #define JSON_HEDLEY_GCC_HAS_ATTRIBUTE(attribute,major,minor,patch)
      JSON_HEDLEY_GCC_VERSION_CHECK(major,minor,patch)
00870 #endif
00871
00872 #if defined(JSON_HEDLEY_HAS_CPP_ATTRIBUTE)
00873     #undef JSON_HEDLEY_HAS_CPP_ATTRIBUTE
00874 #endif
00875 #if \
00876     defined(__has_cpp_attribute) && \
00877     defined(__cplusplus) && \
00878     (!defined(JSON_HEDLEY_SUNPRO_VERSION) || JSON_HEDLEY_SUNPRO_VERSION_CHECK(5,15,0))
00879     #define JSON_HEDLEY_HAS_CPP_ATTRIBUTE(attribute) __has_cpp_attribute(attribute)
00880 #else
00881     #define JSON_HEDLEY_HAS_CPP_ATTRIBUTE(attribute) (0)
00882 #endif
00883
00884 #if defined(JSON_HEDLEY_HAS_CPP_ATTRIBUTE_NS)
00885     #undef JSON_HEDLEY_HAS_CPP_ATTRIBUTE_NS
00886 #endif
00887 #if !defined(__cplusplus) || !defined(__has_cpp_attribute)
00888     #define JSON_HEDLEY_HAS_CPP_ATTRIBUTE_NS(ns,attribute) (0)
00889 #elif \
00890     !defined(JSON_HEDLEY_PGI_VERSION) && \
00891     !defined(JSON_HEDLEY_IAR_VERSION) && \
00892     (!defined(JSON_HEDLEY_SUNPRO_VERSION) || JSON_HEDLEY_SUNPRO_VERSION_CHECK(5,15,0)) && \
00893     (!defined(JSON_HEDLEY_MSVC_VERSION) || JSON_HEDLEY_MSVC_VERSION_CHECK(19,20,0))
00894     #define JSON_HEDLEY_HAS_CPP_ATTRIBUTE_NS(ns,attribute)
      JSON_HEDLEY_HAS_CPP_ATTRIBUTE(ns::attribute)
```

```
00895 #else
00896     #define JSON_HEDLEY_HAS_CPP_ATTRIBUTE_NS(ns,attribute) (0)
00897 #endif
00898
00899 #if defined(JSON_HEDLEY_GNUC_HAS_CPP_ATTRIBUTE)
00900     #undef JSON_HEDLEY_GNUC_HAS_CPP_ATTRIBUTE
00901 #endif
00902 #if defined(__has_cpp_attribute) && defined(__cplusplus)
00903     #define JSON_HEDLEY_GNUC_HAS_CPP_ATTRIBUTE(attribute,major,minor,patch)
      __has_cpp_attribute(attribute)
00904 #else
00905     #define JSON_HEDLEY_GNUC_HAS_CPP_ATTRIBUTE(attribute,major,minor,patch)
      JSON_HEDLEY_GNUC_VERSION_CHECK(major,minor,patch)
00906 #endif
00907
00908 #if defined(JSON_HEDLEY_GCC_HAS_CPP_ATTRIBUTE)
00909     #undef JSON_HEDLEY_GCC_HAS_CPP_ATTRIBUTE
00910 #endif
00911 #if defined(__has_cpp_attribute) && defined(__cplusplus)
00912     #define JSON_HEDLEY_GCC_HAS_CPP_ATTRIBUTE(attribute,major,minor,patch)
      __has_cpp_attribute(attribute)
00913 #else
00914     #define JSON_HEDLEY_GCC_HAS_CPP_ATTRIBUTE(attribute,major,minor,patch)
      JSON_HEDLEY_GCC_VERSION_CHECK(major,minor,patch)
00915 #endif
00916
00917 #if defined(JSON_HEDLEY_HAS_BUILTIN)
00918     #undef JSON_HEDLEY_HAS_BUILTIN
00919 #endif
00920 #if defined(__has_builtin)
00921     #define JSON_HEDLEY_HAS_BUILTIN(builtin) __has_builtin(builtin)
00922 #else
00923     #define JSON_HEDLEY_HAS_BUILTIN(builtin) (0)
00924 #endif
00925
00926 #if defined(JSON_HEDLEY_GNUC_HAS_BUILTIN)
00927     #undef JSON_HEDLEY_GNUC_HAS_BUILTIN
00928 #endif
00929 #if defined(__has_builtin)
00930     #define JSON_HEDLEY_GNUC_HAS_BUILTIN(builtin,major,minor,patch) __has_builtin(builtin)
00931 #else
00932     #define JSON_HEDLEY_GNUC_HAS_BUILTIN(builtin,major,minor,patch)
      JSON_HEDLEY_GNUC_VERSION_CHECK(major,minor,patch)
00933 #endif
00934
00935 #if defined(JSON_HEDLEY_GCC_HAS_BUILTIN)
00936     #undef JSON_HEDLEY_GCC_HAS_BUILTIN
00937 #endif
00938 #if defined(__has_builtin)
00939     #define JSON_HEDLEY_GCC_HAS_BUILTIN(builtin,major,minor,patch) __has_builtin(builtin)
00940 #else
00941     #define JSON_HEDLEY_GCC_HAS_BUILTIN(builtin,major,minor,patch)
      JSON_HEDLEY_GCC_VERSION_CHECK(major,minor,patch)
00942 #endif
00943
00944 #if defined(JSON_HEDLEY_HAS_FEATURE)
00945     #undef JSON_HEDLEY_HAS_FEATURE
00946 #endif
00947 #if defined(__has_feature)
00948     #define JSON_HEDLEY_HAS_FEATURE(feature) __has_feature(feature)
00949 #else
00950     #define JSON_HEDLEY_HAS_FEATURE(feature) (0)
00951 #endif
00952
00953 #if defined(JSON_HEDLEY_GNUC_HAS_FEATURE)
00954     #undef JSON_HEDLEY_GNUC_HAS_FEATURE
00955 #endif
00956 #if defined(__has_feature)
00957     #define JSON_HEDLEY_GNUC_HAS_FEATURE(feature,major,minor,patch) __has_feature(feature)
00958 #else
00959     #define JSON_HEDLEY_GNUC_HAS_FEATURE(feature,major,minor,patch)
      JSON_HEDLEY_GNUC_VERSION_CHECK(major,minor,patch)
00960 #endif
00961
00962 #if defined(JSON_HEDLEY_GCC_HAS_FEATURE)
00963     #undef JSON_HEDLEY_GCC_HAS_FEATURE
00964 #endif
00965 #if defined(__has_feature)
00966     #define JSON_HEDLEY_GCC_HAS_FEATURE(feature,major,minor,patch) __has_feature(feature)
00967 #else
00968     #define JSON_HEDLEY_GCC_HAS_FEATURE(feature,major,minor,patch)
      JSON_HEDLEY_GCC_VERSION_CHECK(major,minor,patch)
00969 #endif
00970
00971 #if defined(JSON_HEDLEY_HAS_EXTENSION)
00972     #undef JSON_HEDLEY_HAS_EXTENSION
00973 #endif
```

```
00974 #if defined(__has_extension)
00975     #define JSON_HEDLEY_HAS_EXTENSION(extension) __has_extension(extension)
00976 #else
00977     #define JSON_HEDLEY_HAS_EXTENSION(extension) (0)
00978 #endif
00979
00980 #if defined(JSON_HEDLEY_GNUC_HAS_EXTENSION)
00981     #undef JSON_HEDLEY_GNUC_HAS_EXTENSION
00982 #endif
00983 #if defined(__has_extension)
00984     #define JSON_HEDLEY_GNUC_HAS_EXTENSION(extension,major,minor,patch) __has_extension(extension)
00985 #else
00986     #define JSON_HEDLEY_GNUC_HAS_EXTENSION(extension,major,minor,patch)
      JSON_HEDLEY_GNUC_VERSION_CHECK(major,minor,patch)
00987 #endif
00988
00989 #if defined(JSON_HEDLEY_GCC_HAS_EXTENSION)
00990     #undef JSON_HEDLEY_GCC_HAS_EXTENSION
00991 #endif
00992 #if defined(__has_extension)
00993     #define JSON_HEDLEY_GCC_HAS_EXTENSION(extension,major,minor,patch) __has_extension(extension)
00994 #else
00995     #define JSON_HEDLEY_GCC_HAS_EXTENSION(extension,major,minor,patch)
      JSON_HEDLEY_GCC_VERSION_CHECK(major,minor,patch)
00996 #endif
00997
00998 #if defined(JSON_HEDLEY_HAS_DECLSPEC_ATTRIBUTE)
00999     #undef JSON_HEDLEY_HAS_DECLSPEC_ATTRIBUTE
01000 #endif
01001 #if defined(__has_declspec_attribute)
01002     #define JSON_HEDLEY_HAS_DECLSPEC_ATTRIBUTE(attribute) __has_declspec_attribute(attribute)
01003 #else
01004     #define JSON_HEDLEY_HAS_DECLSPEC_ATTRIBUTE(attribute) (0)
01005 #endif
01006
01007 #if defined(JSON_HEDLEY_GNUC_HAS_DECLSPEC_ATTRIBUTE)
01008     #undef JSON_HEDLEY_GNUC_HAS_DECLSPEC_ATTRIBUTE
01009 #endif
01010 #if defined(__has_declspec_attribute)
01011     #define JSON_HEDLEY_GNUC_HAS_DECLSPEC_ATTRIBUTE(attribute,major,minor,patch)
      __has_declspec_attribute(attribute)
01012 #else
01013     #define JSON_HEDLEY_GNUC_HAS_DECLSPEC_ATTRIBUTE(attribute,major,minor,patch)
      JSON_HEDLEY_GNUC_VERSION_CHECK(major,minor,patch)
01014 #endif
01015
01016 #if defined(JSON_HEDLEY_GCC_HAS_DECLSPEC_ATTRIBUTE)
01017     #undef JSON_HEDLEY_GCC_HAS_DECLSPEC_ATTRIBUTE
01018 #endif
01019 #if defined(__has_declspec_attribute)
01020     #define JSON_HEDLEY_GCC_HAS_DECLSPEC_ATTRIBUTE(attribute,major,minor,patch)
      __has_declspec_attribute(attribute)
01021 #else
01022     #define JSON_HEDLEY_GCC_HAS_DECLSPEC_ATTRIBUTE(attribute,major,minor,patch)
      JSON_HEDLEY_GCC_VERSION_CHECK(major,minor,patch)
01023 #endif
01024
01025 #if defined(JSON_HEDLEY_HAS_WARNING)
01026     #undef JSON_HEDLEY_HAS_WARNING
01027 #endif
01028 #if defined(__has_warning)
01029     #define JSON_HEDLEY_HAS_WARNING(warning) __has_warning(warning)
01030 #else
01031     #define JSON_HEDLEY_HAS_WARNING(warning) (0)
01032 #endif
01033
01034 #if defined(JSON_HEDLEY_GNUC_HAS_WARNING)
01035     #undef JSON_HEDLEY_GNUC_HAS_WARNING
01036 #endif
01037 #if defined(__has_warning)
01038     #define JSON_HEDLEY_GNUC_HAS_WARNING(warning,major,minor,patch) __has_warning(warning)
01039 #else
01040     #define JSON_HEDLEY_GNUC_HAS_WARNING(warning,major,minor,patch)
      JSON_HEDLEY_GNUC_VERSION_CHECK(major,minor,patch)
01041 #endif
01042
01043 #if defined(JSON_HEDLEY_GCC_HAS_WARNING)
01044     #undef JSON_HEDLEY_GCC_HAS_WARNING
01045 #endif
01046 #if defined(__has_warning)
01047     #define JSON_HEDLEY_GCC_HAS_WARNING(warning,major,minor,patch) __has_warning(warning)
01048 #else
01049     #define JSON_HEDLEY_GCC_HAS_WARNING(warning,major,minor,patch)
      JSON_HEDLEY_GCC_VERSION_CHECK(major,minor,patch)
01050 #endif
01051
01052 #if \
```

```
01053     (defined(__STDC_VERSION__) && (__STDC_VERSION__ >= 199901L)) || \
01054     defined(__clang__) || \
01055     JSON_HEDLEY_GCC_VERSION_CHECK(3,0,0) || \
01056     JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0) || \
01057     JSON_HEDLEY_IAR_VERSION_CHECK(8,0,0) || \
01058     JSON_HEDLEY_PGI_VERSION_CHECK(18,4,0) || \
01059     JSON_HEDLEY_ARM_VERSION_CHECK(4,1,0) || \
01060     JSON_HEDLEY_TI_VERSION_CHECK(15,12,0) || \
01061     JSON_HEDLEY_TI_ARMCL_VERSION_CHECK(4,7,0) || \
01062     JSON_HEDLEY_TI_CL430_VERSION_CHECK(2,0,1) || \
01063     JSON_HEDLEY_TI_CL2000_VERSION_CHECK(6,1,0) || \
01064     JSON_HEDLEY_TI_CL6X_VERSION_CHECK(7,0,0) || \
01065     JSON_HEDLEY_TI_CL7X_VERSION_CHECK(1,2,0) || \
01066     JSON_HEDLEY_TI_CLPRU_VERSION_CHECK(2,1,0) || \
01067     JSON_HEDLEY_CRAY_VERSION_CHECK(5,0,0) || \
01068     JSON_HEDLEY_TINYC_VERSION_CHECK(0,9,17) || \
01069     JSON_HEDLEY_SUNPRO_VERSION_CHECK(8,0,0) || \
01070     (JSON_HEDLEY_IBM_VERSION_CHECK(10,1,0) && defined(__C99_PRAGMA_OPERATOR))
01071     #define JSON_HEDLEY_PRAGMA(value) _Pragma(#value)
01072 #elif JSON_HEDLEY_MSVC_VERSION_CHECK(15,0,0)
01073     #define JSON_HEDLEY_PRAGMA(value) __pragma(value)
01074 #else
01075     #define JSON_HEDLEY_PRAGMA(value)
01076 #endif
01077
01078 #if defined(JSON_HEDLEY_DIAGNOSTIC_PUSH)
01079     #undef JSON_HEDLEY_DIAGNOSTIC_PUSH
01080 #endif
01081 #if defined(JSON_HEDLEY_DIAGNOSTIC_POP)
01082     #undef JSON_HEDLEY_DIAGNOSTIC_POP
01083 #endif
01084 #if defined(__clang__)
01085     #define JSON_HEDLEY_DIAGNOSTIC_PUSH _Pragma("clang diagnostic push")
01086     #define JSON_HEDLEY_DIAGNOSTIC_POP _Pragma("clang diagnostic pop")
01087 #elif JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0)
01088     #define JSON_HEDLEY_DIAGNOSTIC_PUSH _Pragma("warning(push)")
01089     #define JSON_HEDLEY_DIAGNOSTIC_POP _Pragma("warning(pop)")
01090 #elif JSON_HEDLEY_GCC_VERSION_CHECK(4,6,0)
01091     #define JSON_HEDLEY_DIAGNOSTIC_PUSH _Pragma("GCC diagnostic push")
01092     #define JSON_HEDLEY_DIAGNOSTIC_POP _Pragma("GCC diagnostic pop")
01093 #elif \
01094     JSON_HEDLEY_MSVC_VERSION_CHECK(15,0,0) || \
01095     JSON_HEDLEY_INTEL_CL_VERSION_CHECK(2021,1,0)
01096     #define JSON_HEDLEY_DIAGNOSTIC_PUSH __pragma(warning(push))
01097     #define JSON_HEDLEY_DIAGNOSTIC_POP __pragma(warning(pop))
01098 #elif JSON_HEDLEY_ARM_VERSION_CHECK(5,6,0)
01099     #define JSON_HEDLEY_DIAGNOSTIC_PUSH _Pragma("push")
01100     #define JSON_HEDLEY_DIAGNOSTIC_POP _Pragma("pop")
01101 #elif \
01102     JSON_HEDLEY_TI_VERSION_CHECK(15,12,0) || \
01103     JSON_HEDLEY_TI_ARMCL_VERSION_CHECK(5,2,0) || \
01104     JSON_HEDLEY_TI_CL430_VERSION_CHECK(4,4,0) || \
01105     JSON_HEDLEY_TI_CL6X_VERSION_CHECK(8,1,0) || \
01106     JSON_HEDLEY_TI_CL7X_VERSION_CHECK(1,2,0) || \
01107     JSON_HEDLEY_TI_CLPRU_VERSION_CHECK(2,1,0)
01108     #define JSON_HEDLEY_DIAGNOSTIC_PUSH _Pragma("diag_push")
01109     #define JSON_HEDLEY_DIAGNOSTIC_POP _Pragma("diag_pop")
01110 #elif JSON_HEDLEY_PELLES_VERSION_CHECK(2,90,0)
01111     #define JSON_HEDLEY_DIAGNOSTIC_PUSH _Pragma("warning(push)")
01112     #define JSON_HEDLEY_DIAGNOSTIC_POP _Pragma("warning(pop)")
01113 #else
01114     #define JSON_HEDLEY_DIAGNOSTIC_PUSH
01115     #define JSON_HEDLEY_DIAGNOSTIC_POP
01116 #endif
01117
01118 /* JSON_HEDLEY_DIAGNOSTIC_DISABLE_CPP98_COMPAT_WRAP_ is for
01119    HEDLEY INTERNAL USE ONLY.  API subject to change without notice. */
01120 #if defined(JSON_HEDLEY_DIAGNOSTIC_DISABLE_CPP98_COMPAT_WRAP_)
01121     #undef JSON_HEDLEY_DIAGNOSTIC_DISABLE_CPP98_COMPAT_WRAP_
01122 #endif
01123 #if defined(__cplusplus)
01124 #  if JSON_HEDLEY_HAS_WARNING("-Wc++98-compat")
01125 #    if JSON_HEDLEY_HAS_WARNING("-Wc++17-extensions")
01126 #      if JSON_HEDLEY_HAS_WARNING("-Wc++1z-extensions")
01127 #        define JSON_HEDLEY_DIAGNOSTIC_DISABLE_CPP98_COMPAT_WRAP_(xpr) \
01128     JSON_HEDLEY_DIAGNOSTIC_PUSH \
01129     _Pragma("clang diagnostic ignored \"-Wc++98-compat\"") \
01130     _Pragma("clang diagnostic ignored \"-Wc++17-extensions\"") \
01131     _Pragma("clang diagnostic ignored \"-Wc++1z-extensions\"") \
01132     xpr \
01133     JSON_HEDLEY_DIAGNOSTIC_POP
01134 #      else
01135 #        define JSON_HEDLEY_DIAGNOSTIC_DISABLE_CPP98_COMPAT_WRAP_(xpr) \
01136     JSON_HEDLEY_DIAGNOSTIC_PUSH \
01137     _Pragma("clang diagnostic ignored \"-Wc++98-compat\"") \
01138     _Pragma("clang diagnostic ignored \"-Wc++17-extensions\"") \
01139     xpr \
```

```
01140     JSON_HEDLEY_DIAGNOSTIC_POP
01141 #       endif
01142 #     else
01143 #       define JSON_HEDLEY_DIAGNOSTIC_DISABLE_CPP98_COMPAT_WRAP_(xpr) \
01144     JSON_HEDLEY_DIAGNOSTIC_PUSH \
01145     _Pragma("clang diagnostic ignored \"-Wc++98-compat\"") \
01146     xpr \
01147     JSON_HEDLEY_DIAGNOSTIC_POP
01148 #     endif
01149 #   endif
01150 #endif
01151 #if !defined(JSON_HEDLEY_DIAGNOSTIC_DISABLE_CPP98_COMPAT_WRAP_)
01152     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_CPP98_COMPAT_WRAP_(x) x
01153 #endif
01154
01155 #if defined(JSON_HEDLEY_CONST_CAST)
01156     #undef JSON_HEDLEY_CONST_CAST
01157 #endif
01158 #if defined(__cplusplus)
01159 #  define JSON_HEDLEY_CONST_CAST(T, expr) (const_cast<T>(expr))
01160 #elif \
01161   JSON_HEDLEY_HAS_WARNING("-Wcast-qual") || \
01162   JSON_HEDLEY_GCC_VERSION_CHECK(4,6,0) || \
01163   JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0)
01164 #  define JSON_HEDLEY_CONST_CAST(T, expr) (__extension__ ({ \
01165         JSON_HEDLEY_DIAGNOSTIC_PUSH \
01166         JSON_HEDLEY_DIAGNOSTIC_DISABLE_CAST_QUAL \
01167         ((T) (expr)); \
01168         JSON_HEDLEY_DIAGNOSTIC_POP \
01169     }))
01170 #else
01171 #  define JSON_HEDLEY_CONST_CAST(T, expr) ((T) (expr))
01172 #endif
01173
01174 #if defined(JSON_HEDLEY_REINTERPRET_CAST)
01175     #undef JSON_HEDLEY_REINTERPRET_CAST
01176 #endif
01177 #if defined(__cplusplus)
01178     #define JSON_HEDLEY_REINTERPRET_CAST(T, expr) (reinterpret_cast<T>(expr))
01179 #else
01180     #define JSON_HEDLEY_REINTERPRET_CAST(T, expr) ((T) (expr))
01181 #endif
01182
01183 #if defined(JSON_HEDLEY_STATIC_CAST)
01184     #undef JSON_HEDLEY_STATIC_CAST
01185 #endif
01186 #if defined(__cplusplus)
01187     #define JSON_HEDLEY_STATIC_CAST(T, expr) (static_cast<T>(expr))
01188 #else
01189     #define JSON_HEDLEY_STATIC_CAST(T, expr) ((T) (expr))
01190 #endif
01191
01192 #if defined(JSON_HEDLEY_CPP_CAST)
01193     #undef JSON_HEDLEY_CPP_CAST
01194 #endif
01195 #if defined(__cplusplus)
01196 #  if JSON_HEDLEY_HAS_WARNING("-Wold-style-cast")
01197 #    define JSON_HEDLEY_CPP_CAST(T, expr) \
01198     JSON_HEDLEY_DIAGNOSTIC_PUSH \
01199     _Pragma("clang diagnostic ignored \"-Wold-style-cast\"") \
01200     ((T) (expr)) \
01201     JSON_HEDLEY_DIAGNOSTIC_POP
01202 #  elif JSON_HEDLEY_IAR_VERSION_CHECK(8,3,0)
01203 #    define JSON_HEDLEY_CPP_CAST(T, expr) \
01204     JSON_HEDLEY_DIAGNOSTIC_PUSH \
01205     _Pragma("diag_suppress=Pe137") \
01206     JSON_HEDLEY_DIAGNOSTIC_POP
01207 #  else
01208 #    define JSON_HEDLEY_CPP_CAST(T, expr) ((T) (expr))
01209 #  endif
01210 #else
01211 #  define JSON_HEDLEY_CPP_CAST(T, expr) (expr)
01212 #endif
01213
01214 #if defined(JSON_HEDLEY_DIAGNOSTIC_DISABLE_DEPRECATED)
01215     #undef JSON_HEDLEY_DIAGNOSTIC_DISABLE_DEPRECATED
01216 #endif
01217 #if JSON_HEDLEY_HAS_WARNING("-Wdeprecated-declarations")
01218     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_DEPRECATED _Pragma("clang diagnostic ignored
    \"-Wdeprecated-declarations\"")
01219 #elif JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0)
01220     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_DEPRECATED _Pragma("warning(disable:1478 1786)")
01221 #elif JSON_HEDLEY_INTEL_CL_VERSION_CHECK(2021,1,0)
01222     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_DEPRECATED __pragma(warning(disable:1478 1786))
01223 #elif JSON_HEDLEY_PGI_VERSION_CHECK(20,7,0)
01224     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_DEPRECATED _Pragma("diag_suppress 1215,1216,1444,1445")
01225 #elif JSON_HEDLEY_PGI_VERSION_CHECK(17,10,0)
```

```
01226     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_DEPRECATED _Pragma("diag_suppress 1215,1444")
01227 #elif JSON_HEDLEY_GCC_VERSION_CHECK(4,3,0)
01228     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_DEPRECATED _Pragma("GCC diagnostic ignored
      \"-Wdeprecated-declarations\"")
01229 #elif JSON_HEDLEY_MSVC_VERSION_CHECK(15,0,0)
01230     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_DEPRECATED __pragma(warning(disable:4996))
01231 #elif JSON_HEDLEY_MCST_LCC_VERSION_CHECK(1,25,10)
01232     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_DEPRECATED _Pragma("diag_suppress 1215,1444")
01233 #elif \
01234     JSON_HEDLEY_TI_VERSION_CHECK(15,12,0) || \
01235     (JSON_HEDLEY_TI_ARMCL_VERSION_CHECK(4,8,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01236     JSON_HEDLEY_TI_ARMCL_VERSION_CHECK(5,2,0) || \
01237     (JSON_HEDLEY_TI_CL2000_VERSION_CHECK(6,0,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01238     JSON_HEDLEY_TI_CL2000_VERSION_CHECK(6,4,0) || \
01239     (JSON_HEDLEY_TI_CL430_VERSION_CHECK(4,0,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01240     JSON_HEDLEY_TI_CL430_VERSION_CHECK(4,3,0) || \
01241     (JSON_HEDLEY_TI_CL6X_VERSION_CHECK(7,2,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01242     JSON_HEDLEY_TI_CL6X_VERSION_CHECK(7,5,0) || \
01243     JSON_HEDLEY_TI_CL7X_VERSION_CHECK(1,2,0) || \
01244     JSON_HEDLEY_TI_CLPRU_VERSION_CHECK(2,1,0)
01245     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_DEPRECATED _Pragma("diag_suppress 1291,1718")
01246 #elif JSON_HEDLEY_SUNPRO_VERSION_CHECK(5,13,0) && !defined(__cplusplus)
01247     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_DEPRECATED
      _Pragma("error_messages(off,E_DEPRECATED_ATT,E_DEPRECATED_ATT_MESS)")
01248 #elif JSON_HEDLEY_SUNPRO_VERSION_CHECK(5,13,0) && defined(__cplusplus)
01249     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_DEPRECATED
      _Pragma("error_messages(off,symdeprecated,symdeprecated2)")
01250 #elif JSON_HEDLEY_IAR_VERSION_CHECK(8,0,0)
01251     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_DEPRECATED _Pragma("diag_suppress=Pe1444,Pe1215")
01252 #elif JSON_HEDLEY_PELLES_VERSION_CHECK(2,90,0)
01253     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_DEPRECATED _Pragma("warn(disable:2241)")
01254 #else
01255     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_DEPRECATED
01256 #endif
01257
01258 #if defined(JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_PRAGMAS)
01259     #undef JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_PRAGMAS
01260 #endif
01261 #if JSON_HEDLEY_HAS_WARNING("-Wunknown-pragmas")
01262     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_PRAGMAS _Pragma("clang diagnostic ignored
      \"-Wunknown-pragmas\"")
01263 #elif JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0)
01264     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_PRAGMAS _Pragma("warning(disable:161)")
01265 #elif JSON_HEDLEY_INTEL_CL_VERSION_CHECK(2021,1,0)
01266     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_PRAGMAS __pragma(warning(disable:161))
01267 #elif JSON_HEDLEY_PGI_VERSION_CHECK(17,10,0)
01268     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_PRAGMAS _Pragma("diag_suppress 1675")
01269 #elif JSON_HEDLEY_GCC_VERSION_CHECK(4,3,0)
01270     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_PRAGMAS _Pragma("GCC diagnostic ignored
      \"-Wunknown-pragmas\"")
01271 #elif JSON_HEDLEY_MSVC_VERSION_CHECK(15,0,0)
01272     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_PRAGMAS __pragma(warning(disable:4068))
01273 #elif \
01274     JSON_HEDLEY_TI_VERSION_CHECK(16,9,0) || \
01275     JSON_HEDLEY_TI_CL6X_VERSION_CHECK(8,0,0) || \
01276     JSON_HEDLEY_TI_CL7X_VERSION_CHECK(1,2,0) || \
01277     JSON_HEDLEY_TI_CLPRU_VERSION_CHECK(2,3,0)
01278     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_PRAGMAS _Pragma("diag_suppress 163")
01279 #elif JSON_HEDLEY_TI_CL6X_VERSION_CHECK(8,0,0)
01280     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_PRAGMAS _Pragma("diag_suppress 163")
01281 #elif JSON_HEDLEY_IAR_VERSION_CHECK(8,0,0)
01282     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_PRAGMAS _Pragma("diag_suppress=Pe161")
01283 #elif JSON_HEDLEY_MCST_LCC_VERSION_CHECK(1,25,10)
01284     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_PRAGMAS _Pragma("diag_suppress 161")
01285 #else
01286     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_PRAGMAS
01287 #endif
01288
01289 #if defined(JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_CPP_ATTRIBUTES)
01290     #undef JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_CPP_ATTRIBUTES
01291 #endif
01292 #if JSON_HEDLEY_HAS_WARNING("-Wunknown-attributes")
01293     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_CPP_ATTRIBUTES _Pragma("clang diagnostic ignored
      \"-Wunknown-attributes\"")
01294 #elif JSON_HEDLEY_GCC_VERSION_CHECK(4,6,0)
01295     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_CPP_ATTRIBUTES _Pragma("GCC diagnostic ignored
      \"-Wdeprecated-declarations\"")
01296 #elif JSON_HEDLEY_INTEL_VERSION_CHECK(17,0,0)
01297     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_CPP_ATTRIBUTES _Pragma("warning(disable:1292)")
01298 #elif JSON_HEDLEY_INTEL_CL_VERSION_CHECK(2021,1,0)
01299     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_CPP_ATTRIBUTES __pragma(warning(disable:1292))
01300 #elif JSON_HEDLEY_MSVC_VERSION_CHECK(19,0,0)
01301     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_CPP_ATTRIBUTES __pragma(warning(disable:5030))
01302 #elif JSON_HEDLEY_PGI_VERSION_CHECK(20,7,0)
01303     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_CPP_ATTRIBUTES _Pragma("diag_suppress 1097,1098")
01304 #elif JSON_HEDLEY_PGI_VERSION_CHECK(17,10,0)
01305     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_CPP_ATTRIBUTES _Pragma("diag_suppress 1097")
```

```
01306 #elif JSON_HEDLEY_SUNPRO_VERSION_CHECK(5,14,0) && defined(__cplusplus)
01307     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_CPP_ATTRIBUTES
      _Pragma("error_messages(off,attrskipunsup)")
01308 #elif \
01309     JSON_HEDLEY_TI_VERSION_CHECK(18,1,0) || \
01310     JSON_HEDLEY_TI_CL6X_VERSION_CHECK(8,3,0) || \
01311     JSON_HEDLEY_TI_CL7X_VERSION_CHECK(1,2,0)
01312     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_CPP_ATTRIBUTES _Pragma("diag_suppress 1173")
01313 #elif JSON_HEDLEY_IAR_VERSION_CHECK(8,0,0)
01314     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_CPP_ATTRIBUTES _Pragma("diag_suppress=Pe1097")
01315 #elif JSON_HEDLEY_MCST_LCC_VERSION_CHECK(1,25,10)
01316     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_CPP_ATTRIBUTES _Pragma("diag_suppress 1097")
01317 #else
01318     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_CPP_ATTRIBUTES
01319 #endif
01320
01321 #if defined(JSON_HEDLEY_DIAGNOSTIC_DISABLE_CAST_QUAL)
01322     #undef JSON_HEDLEY_DIAGNOSTIC_DISABLE_CAST_QUAL
01323 #endif
01324 #if JSON_HEDLEY_HAS_WARNING("-Wcast-qual")
01325     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_CAST_QUAL _Pragma("clang diagnostic ignored
      \"-Wcast-qual\"")
01326 #elif JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0)
01327     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_CAST_QUAL _Pragma("warning(disable:2203 2331)")
01328 #elif JSON_HEDLEY_GCC_VERSION_CHECK(3,0,0)
01329     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_CAST_QUAL _Pragma("GCC diagnostic ignored \"-Wcast-qual\"")
01330 #else
01331     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_CAST_QUAL
01332 #endif
01333
01334 #if defined(JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNUSED_FUNCTION)
01335     #undef JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNUSED_FUNCTION
01336 #endif
01337 #if JSON_HEDLEY_HAS_WARNING("-Wunused-function")
01338     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNUSED_FUNCTION _Pragma("clang diagnostic ignored
      \"-Wunused-function\"")
01339 #elif JSON_HEDLEY_GCC_VERSION_CHECK(3,4,0)
01340     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNUSED_FUNCTION _Pragma("GCC diagnostic ignored
      \"-Wunused-function\"")
01341 #elif JSON_HEDLEY_MSVC_VERSION_CHECK(1,0,0)
01342     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNUSED_FUNCTION __pragma(warning(disable:4505))
01343 #elif JSON_HEDLEY_MCST_LCC_VERSION_CHECK(1,25,10)
01344     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNUSED_FUNCTION _Pragma("diag_suppress 3142")
01345 #else
01346     #define JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNUSED_FUNCTION
01347 #endif
01348
01349 #if defined(JSON_HEDLEY_DEPRECATED)
01350     #undef JSON_HEDLEY_DEPRECATED
01351 #endif
01352 #if defined(JSON_HEDLEY_DEPRECATED_FOR)
01353     #undef JSON_HEDLEY_DEPRECATED_FOR
01354 #endif
01355 #if \
01356     JSON_HEDLEY_MSVC_VERSION_CHECK(14,0,0) || \
01357     JSON_HEDLEY_INTEL_CL_VERSION_CHECK(2021,1,0)
01358     #define JSON_HEDLEY_DEPRECATED(since) __declspec(deprecated("Since " # since))
01359     #define JSON_HEDLEY_DEPRECATED_FOR(since, replacement) __declspec(deprecated("Since " #since ";
      use " #replacement))
01360 #elif \
01361     (JSON_HEDLEY_HAS_EXTENSION(attribute_deprecated_with_message) &&
      !defined(JSON_HEDLEY_IAR_VERSION)) || \
01362     JSON_HEDLEY_GCC_VERSION_CHECK(4,5,0) || \
01363     JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0) || \
01364     JSON_HEDLEY_ARM_VERSION_CHECK(5,6,0) || \
01365     JSON_HEDLEY_SUNPRO_VERSION_CHECK(5,13,0) || \
01366     JSON_HEDLEY_PGI_VERSION_CHECK(17,10,0) || \
01367     JSON_HEDLEY_TI_VERSION_CHECK(18,1,0) || \
01368     JSON_HEDLEY_TI_ARMCL_VERSION_CHECK(18,1,0) || \
01369     JSON_HEDLEY_TI_CL6X_VERSION_CHECK(8,3,0) || \
01370     JSON_HEDLEY_TI_CL7X_VERSION_CHECK(1,2,0) || \
01371     JSON_HEDLEY_TI_CLPRU_VERSION_CHECK(2,3,0) || \
01372     JSON_HEDLEY_MCST_LCC_VERSION_CHECK(1,25,10)
01373     #define JSON_HEDLEY_DEPRECATED(since) __attribute__((__deprecated__("Since " #since)))
01374     #define JSON_HEDLEY_DEPRECATED_FOR(since, replacement) __attribute__((__deprecated__("Since "
      #since "; use " #replacement)))
01375 #elif defined(__cplusplus) && (__cplusplus >= 201402L)
01376     #define JSON_HEDLEY_DEPRECATED(since)
      JSON_HEDLEY_DIAGNOSTIC_DISABLE_CPP98_COMPAT_WRAP_([[deprecated("Since " #since)]])
01377     #define JSON_HEDLEY_DEPRECATED_FOR(since, replacement)
      JSON_HEDLEY_DIAGNOSTIC_DISABLE_CPP98_COMPAT_WRAP_([[deprecated("Since " #since "; use "
      #replacement)]])
01378 #elif \
01379     JSON_HEDLEY_HAS_ATTRIBUTE(deprecated) || \
01380     JSON_HEDLEY_GCC_VERSION_CHECK(3,1,0) || \
01381     JSON_HEDLEY_ARM_VERSION_CHECK(4,1,0) || \
01382     JSON_HEDLEY_TI_VERSION_CHECK(15,12,0) || \
```

```
01383     (JSON_HEDLEY_TI_ARMCL_VERSION_CHECK(4,8,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01384     JSON_HEDLEY_TI_ARMCL_VERSION_CHECK(5,2,0) || \
01385     (JSON_HEDLEY_TI_CL2000_VERSION_CHECK(6,0,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01386     JSON_HEDLEY_TI_CL2000_VERSION_CHECK(6,4,0) || \
01387     (JSON_HEDLEY_TI_CL430_VERSION_CHECK(4,0,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01388     JSON_HEDLEY_TI_CL430_VERSION_CHECK(4,3,0) || \
01389     (JSON_HEDLEY_TI_CL6X_VERSION_CHECK(7,2,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01390     JSON_HEDLEY_TI_CL6X_VERSION_CHECK(7,5,0) || \
01391     JSON_HEDLEY_TI_CL7X_VERSION_CHECK(1,2,0) || \
01392     JSON_HEDLEY_TI_CLPRU_VERSION_CHECK(2,1,0) || \
01393     JSON_HEDLEY_MCST_LCC_VERSION_CHECK(1,25,10) || \
01394     JSON_HEDLEY_IAR_VERSION_CHECK(8,10,0)
01395     #define JSON_HEDLEY_DEPRECATED(since) __attribute__((__deprecated__))
01396     #define JSON_HEDLEY_DEPRECATED_FOR(since, replacement) __attribute__((__deprecated__))
01397 #elif \
01398     JSON_HEDLEY_MSVC_VERSION_CHECK(13,10,0) || \
01399     JSON_HEDLEY_PELLES_VERSION_CHECK(6,50,0) || \
01400     JSON_HEDLEY_INTEL_CL_VERSION_CHECK(2021,1,0)
01401     #define JSON_HEDLEY_DEPRECATED(since) __declspec(deprecated)
01402     #define JSON_HEDLEY_DEPRECATED_FOR(since, replacement) __declspec(deprecated)
01403 #elif JSON_HEDLEY_IAR_VERSION_CHECK(8,0,0)
01404     #define JSON_HEDLEY_DEPRECATED(since) _Pragma("deprecated")
01405     #define JSON_HEDLEY_DEPRECATED_FOR(since, replacement) _Pragma("deprecated")
01406 #else
01407     #define JSON_HEDLEY_DEPRECATED(since)
01408     #define JSON_HEDLEY_DEPRECATED_FOR(since, replacement)
01409 #endif
01410
01411 #if defined(JSON_HEDLEY_UNAVAILABLE)
01412     #undef JSON_HEDLEY_UNAVAILABLE
01413 #endif
01414 #if \
01415     JSON_HEDLEY_HAS_ATTRIBUTE(warning) || \
01416     JSON_HEDLEY_GCC_VERSION_CHECK(4,3,0) || \
01417     JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0) || \
01418     JSON_HEDLEY_MCST_LCC_VERSION_CHECK(1,25,10)
01419     #define JSON_HEDLEY_UNAVAILABLE(available_since) __attribute__((__warning__("Not available until "
      #available_since)))
01420 #else
01421     #define JSON_HEDLEY_UNAVAILABLE(available_since)
01422 #endif
01423
01424 #if defined(JSON_HEDLEY_WARN_UNUSED_RESULT)
01425     #undef JSON_HEDLEY_WARN_UNUSED_RESULT
01426 #endif
01427 #if defined(JSON_HEDLEY_WARN_UNUSED_RESULT_MSG)
01428     #undef JSON_HEDLEY_WARN_UNUSED_RESULT_MSG
01429 #endif
01430 #if \
01431     JSON_HEDLEY_HAS_ATTRIBUTE(warn_unused_result) || \
01432     JSON_HEDLEY_GCC_VERSION_CHECK(3,4,0) || \
01433     JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0) || \
01434     JSON_HEDLEY_TI_VERSION_CHECK(15,12,0) || \
01435     (JSON_HEDLEY_TI_ARMCL_VERSION_CHECK(4,8,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01436     JSON_HEDLEY_TI_ARMCL_VERSION_CHECK(5,2,0) || \
01437     (JSON_HEDLEY_TI_CL2000_VERSION_CHECK(6,0,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01438     JSON_HEDLEY_TI_CL2000_VERSION_CHECK(6,4,0) || \
01439     (JSON_HEDLEY_TI_CL430_VERSION_CHECK(4,0,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01440     JSON_HEDLEY_TI_CL430_VERSION_CHECK(4,3,0) || \
01441     (JSON_HEDLEY_TI_CL6X_VERSION_CHECK(7,2,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01442     JSON_HEDLEY_TI_CL6X_VERSION_CHECK(7,5,0) || \
01443     JSON_HEDLEY_TI_CL7X_VERSION_CHECK(1,2,0) || \
01444     JSON_HEDLEY_TI_CLPRU_VERSION_CHECK(2,1,0) || \
01445     (JSON_HEDLEY_SUNPRO_VERSION_CHECK(5,15,0) && defined(__cplusplus)) || \
01446     JSON_HEDLEY_PGI_VERSION_CHECK(17,10,0) || \
01447     JSON_HEDLEY_MCST_LCC_VERSION_CHECK(1,25,10)
01448     #define JSON_HEDLEY_WARN_UNUSED_RESULT __attribute__((__warn_unused_result__))
01449     #define JSON_HEDLEY_WARN_UNUSED_RESULT_MSG(msg) __attribute__((__warn_unused_result__))
01450 #elif (JSON_HEDLEY_HAS_CPP_ATTRIBUTE(nodiscard) >= 201907L)
01451     #define JSON_HEDLEY_WARN_UNUSED_RESULT
      JSON_HEDLEY_DIAGNOSTIC_DISABLE_CPP98_COMPAT_WRAP_([[nodiscard]])
01452     #define JSON_HEDLEY_WARN_UNUSED_RESULT_MSG(msg)
      JSON_HEDLEY_DIAGNOSTIC_DISABLE_CPP98_COMPAT_WRAP_([[nodiscard(msg)]])
01453 #elif JSON_HEDLEY_HAS_CPP_ATTRIBUTE(nodiscard)
01454     #define JSON_HEDLEY_WARN_UNUSED_RESULT
      JSON_HEDLEY_DIAGNOSTIC_DISABLE_CPP98_COMPAT_WRAP_([[nodiscard]])
01455     #define JSON_HEDLEY_WARN_UNUSED_RESULT_MSG(msg)
      JSON_HEDLEY_DIAGNOSTIC_DISABLE_CPP98_COMPAT_WRAP_([[nodiscard]])
01456 #elif defined(_Check_return_) /* SAL */
01457     #define JSON_HEDLEY_WARN_UNUSED_RESULT _Check_return_
01458     #define JSON_HEDLEY_WARN_UNUSED_RESULT_MSG(msg) _Check_return_
01459 #else
01460     #define JSON_HEDLEY_WARN_UNUSED_RESULT
01461     #define JSON_HEDLEY_WARN_UNUSED_RESULT_MSG(msg)
01462 #endif
01463
01464 #if defined(JSON_HEDLEY_SENTINEL)
```

```
01465     #undef JSON_HEDLEY_SENTINEL
01466 #endif
01467 #if \
01468     JSON_HEDLEY_HAS_ATTRIBUTE(sentinel) || \
01469     JSON_HEDLEY_GCC_VERSION_CHECK(4,0,0) || \
01470     JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0) || \
01471     JSON_HEDLEY_ARM_VERSION_CHECK(5,4,0) || \
01472     JSON_HEDLEY_MCST_LCC_VERSION_CHECK(1,25,10)
01473     #define JSON_HEDLEY_SENTINEL(position) __attribute__((__sentinel__(position)))
01474 #else
01475     #define JSON_HEDLEY_SENTINEL(position)
01476 #endif
01477
01478 #if defined(JSON_HEDLEY_NO_RETURN)
01479     #undef JSON_HEDLEY_NO_RETURN
01480 #endif
01481 #if JSON_HEDLEY_IAR_VERSION_CHECK(8,0,0)
01482     #define JSON_HEDLEY_NO_RETURN __noreturn
01483 #elif \
01484     JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0) || \
01485     JSON_HEDLEY_MCST_LCC_VERSION_CHECK(1,25,10)
01486     #define JSON_HEDLEY_NO_RETURN __attribute__((__noreturn__))
01487 #elif defined(__STDC_VERSION__) && __STDC_VERSION__ >= 201112L
01488     #define JSON_HEDLEY_NO_RETURN _Noreturn
01489 #elif defined(__cplusplus) && (__cplusplus >= 201103L)
01490     #define JSON_HEDLEY_NO_RETURN JSON_HEDLEY_DIAGNOSTIC_DISABLE_CPP98_COMPAT_WRAP_([[noreturn]])
01491 #elif \
01492     JSON_HEDLEY_HAS_ATTRIBUTE(noreturn) || \
01493     JSON_HEDLEY_GCC_VERSION_CHECK(3,2,0) || \
01494     JSON_HEDLEY_SUNPRO_VERSION_CHECK(5,11,0) || \
01495     JSON_HEDLEY_ARM_VERSION_CHECK(4,1,0) || \
01496     JSON_HEDLEY_IBM_VERSION_CHECK(10,1,0) || \
01497     JSON_HEDLEY_TI_VERSION_CHECK(15,12,0) || \
01498     (JSON_HEDLEY_TI_ARMCL_VERSION_CHECK(4,8,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01499     JSON_HEDLEY_TI_ARMCL_VERSION_CHECK(5,2,0) || \
01500     (JSON_HEDLEY_TI_CL2000_VERSION_CHECK(6,0,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01501     JSON_HEDLEY_TI_CL2000_VERSION_CHECK(6,4,0) || \
01502     (JSON_HEDLEY_TI_CL430_VERSION_CHECK(4,0,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01503     JSON_HEDLEY_TI_CL430_VERSION_CHECK(4,3,0) || \
01504     (JSON_HEDLEY_TI_CL6X_VERSION_CHECK(7,2,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01505     JSON_HEDLEY_TI_CL6X_VERSION_CHECK(7,5,0) || \
01506     JSON_HEDLEY_TI_CL7X_VERSION_CHECK(1,2,0) || \
01507     JSON_HEDLEY_TI_CLPRU_VERSION_CHECK(2,1,0) || \
01508     JSON_HEDLEY_IAR_VERSION_CHECK(8,10,0)
01509     #define JSON_HEDLEY_NO_RETURN __attribute__((__noreturn__))
01510 #elif JSON_HEDLEY_SUNPRO_VERSION_CHECK(5,10,0)
01511     #define JSON_HEDLEY_NO_RETURN _Pragma("does_not_return")
01512 #elif \
01513     JSON_HEDLEY_MSVC_VERSION_CHECK(13,10,0) || \
01514     JSON_HEDLEY_INTEL_CL_VERSION_CHECK(2021,1,0)
01515     #define JSON_HEDLEY_NO_RETURN __declspec(noreturn)
01516 #elif JSON_HEDLEY_TI_CL6X_VERSION_CHECK(6,0,0) && defined(__cplusplus)
01517     #define JSON_HEDLEY_NO_RETURN _Pragma("FUNC_NEVER_RETURNS;")
01518 #elif JSON_HEDLEY_COMPCERT_VERSION_CHECK(3,2,0)
01519     #define JSON_HEDLEY_NO_RETURN __attribute((noreturn))
01520 #elif JSON_HEDLEY_PELLES_VERSION_CHECK(9,0,0)
01521     #define JSON_HEDLEY_NO_RETURN __declspec(noreturn)
01522 #else
01523     #define JSON_HEDLEY_NO_RETURN
01524 #endif
01525
01526 #if defined(JSON_HEDLEY_NO_ESCAPE)
01527     #undef JSON_HEDLEY_NO_ESCAPE
01528 #endif
01529 #if JSON_HEDLEY_HAS_ATTRIBUTE(noescape)
01530     #define JSON_HEDLEY_NO_ESCAPE __attribute__((__noescape__))
01531 #else
01532     #define JSON_HEDLEY_NO_ESCAPE
01533 #endif
01534
01535 #if defined(JSON_HEDLEY_UNREACHABLE)
01536     #undef JSON_HEDLEY_UNREACHABLE
01537 #endif
01538 #if defined(JSON_HEDLEY_UNREACHABLE_RETURN)
01539     #undef JSON_HEDLEY_UNREACHABLE_RETURN
01540 #endif
01541 #if defined(JSON_HEDLEY_ASSUME)
01542     #undef JSON_HEDLEY_ASSUME
01543 #endif
01544 #if \
01545     JSON_HEDLEY_MSVC_VERSION_CHECK(13,10,0) || \
01546     JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0) || \
01547     JSON_HEDLEY_INTEL_CL_VERSION_CHECK(2021,1,0)
01548     #define JSON_HEDLEY_ASSUME(expr) __assume(expr)
01549 #elif JSON_HEDLEY_HAS_BUILTIN(__builtin_assume)
01550     #define JSON_HEDLEY_ASSUME(expr) __builtin_assume(expr)
01551 #elif \
```

```
01552     JSON_HEDLEY_TI_CL2000_VERSION_CHECK(6,2,0) || \
01553     JSON_HEDLEY_TI_CL6X_VERSION_CHECK(4,0,0)
01554     #if defined(__cplusplus)
01555         #define JSON_HEDLEY_ASSUME(expr) std::_nassert(expr)
01556     #else
01557         #define JSON_HEDLEY_ASSUME(expr) _nassert(expr)
01558     #endif
01559 #endif
01560 #if \
01561     (JSON_HEDLEY_HAS_BUILTIN(__builtin_unreachable) && (!defined(JSON_HEDLEY_ARM_VERSION))) || \
01562     JSON_HEDLEY_GCC_VERSION_CHECK(4,5,0) || \
01563     JSON_HEDLEY_PGI_VERSION_CHECK(18,10,0) || \
01564     JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0) || \
01565     JSON_HEDLEY_IBM_VERSION_CHECK(13,1,5) || \
01566     JSON_HEDLEY_CRAY_VERSION_CHECK(10,0,0) || \
01567     JSON_HEDLEY_MCST_LCC_VERSION_CHECK(1,25,10)
01568     #define JSON_HEDLEY_UNREACHABLE() __builtin_unreachable()
01569 #elif defined(JSON_HEDLEY_ASSUME)
01570     #define JSON_HEDLEY_UNREACHABLE() JSON_HEDLEY_ASSUME(0)
01571 #endif
01572 #if !defined(JSON_HEDLEY_ASSUME)
01573     #if defined(JSON_HEDLEY_UNREACHABLE)
01574         #define JSON_HEDLEY_ASSUME(expr) JSON_HEDLEY_STATIC_CAST(void, ((expr) ? 1 :
    (JSON_HEDLEY_UNREACHABLE(), 1)))
01575     #else
01576         #define JSON_HEDLEY_ASSUME(expr) JSON_HEDLEY_STATIC_CAST(void, expr)
01577     #endif
01578 #endif
01579 #if defined(JSON_HEDLEY_UNREACHABLE)
01580     #if  \
01581         JSON_HEDLEY_TI_CL2000_VERSION_CHECK(6,2,0) || \
01582         JSON_HEDLEY_TI_CL6X_VERSION_CHECK(4,0,0)
01583         #define JSON_HEDLEY_UNREACHABLE_RETURN(value) return (JSON_HEDLEY_STATIC_CAST(void,
    JSON_HEDLEY_ASSUME(0)), (value))
01584     #else
01585         #define JSON_HEDLEY_UNREACHABLE_RETURN(value) JSON_HEDLEY_UNREACHABLE()
01586     #endif
01587 #else
01588     #define JSON_HEDLEY_UNREACHABLE_RETURN(value) return (value)
01589 #endif
01590 #if !defined(JSON_HEDLEY_UNREACHABLE)
01591     #define JSON_HEDLEY_UNREACHABLE() JSON_HEDLEY_ASSUME(0)
01592 #endif
01593
01594 JSON_HEDLEY_DIAGNOSTIC_PUSH
01595 #if JSON_HEDLEY_HAS_WARNING("-Wpedantic")
01596     #pragma clang diagnostic ignored "-Wpedantic"
01597 #endif
01598 #if JSON_HEDLEY_HAS_WARNING("-Wc++98-compat-pedantic") && defined(__cplusplus)
01599     #pragma clang diagnostic ignored "-Wc++98-compat-pedantic"
01600 #endif
01601 #if JSON_HEDLEY_GCC_HAS_WARNING("-Wvariadic-macros",4,0,0)
01602     #if defined(__clang__)
01603         #pragma clang diagnostic ignored "-Wvariadic-macros"
01604     #elif defined(JSON_HEDLEY_GCC_VERSION)
01605         #pragma GCC diagnostic ignored "-Wvariadic-macros"
01606     #endif
01607 #endif
01608 #if defined(JSON_HEDLEY_NON_NULL)
01609     #undef JSON_HEDLEY_NON_NULL
01610 #endif
01611 #if \
01612     JSON_HEDLEY_HAS_ATTRIBUTE(nonnull) || \
01613     JSON_HEDLEY_GCC_VERSION_CHECK(3,3,0) || \
01614     JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0) || \
01615     JSON_HEDLEY_ARM_VERSION_CHECK(4,1,0)
01616     #define JSON_HEDLEY_NON_NULL(...) __attribute__((__nonnull__(__VA_ARGS__)))
01617 #else
01618     #define JSON_HEDLEY_NON_NULL(...)
01619 #endif
01620 JSON_HEDLEY_DIAGNOSTIC_POP
01621
01622 #if defined(JSON_HEDLEY_PRINTF_FORMAT)
01623     #undef JSON_HEDLEY_PRINTF_FORMAT
01624 #endif
01625 #if defined(__MINGW32__) && JSON_HEDLEY_GCC_HAS_ATTRIBUTE(format,4,4,0) &&
    !defined(__USE_MINGW_ANSI_STDIO)
01626     #define JSON_HEDLEY_PRINTF_FORMAT(string_idx,first_to_check) __attribute__((__format__(ms_printf,
    string_idx, first_to_check)))
01627 #elif defined(__MINGW32__) && JSON_HEDLEY_GCC_HAS_ATTRIBUTE(format,4,4,0) &&
    defined(__USE_MINGW_ANSI_STDIO)
01628     #define JSON_HEDLEY_PRINTF_FORMAT(string_idx,first_to_check) __attribute__((__format__(gnu_printf,
    string_idx, first_to_check)))
01629 #elif \
01630     JSON_HEDLEY_HAS_ATTRIBUTE(format) || \
01631     JSON_HEDLEY_GCC_VERSION_CHECK(3,1,0) || \
01632     JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0) || \
```

```
01633     JSON_HEDLEY_ARM_VERSION_CHECK(5,6,0) || \
01634     JSON_HEDLEY_IBM_VERSION_CHECK(10,1,0) || \
01635     JSON_HEDLEY_TI_VERSION_CHECK(15,12,0) || \
01636     (JSON_HEDLEY_TI_ARMCL_VERSION_CHECK(4,8,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01637     JSON_HEDLEY_TI_ARMCL_VERSION_CHECK(5,2,0) || \
01638     (JSON_HEDLEY_TI_CL2000_VERSION_CHECK(6,0,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01639     JSON_HEDLEY_TI_CL2000_VERSION_CHECK(6,4,0) || \
01640     (JSON_HEDLEY_TI_CL430_VERSION_CHECK(4,0,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01641     JSON_HEDLEY_TI_CL430_VERSION_CHECK(4,3,0) || \
01642     (JSON_HEDLEY_TI_CL6X_VERSION_CHECK(7,2,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01643     JSON_HEDLEY_TI_CL6X_VERSION_CHECK(7,5,0) || \
01644     JSON_HEDLEY_TI_CL7X_VERSION_CHECK(1,2,0) || \
01645     JSON_HEDLEY_TI_CLPRU_VERSION_CHECK(2,1,0) || \
01646     JSON_HEDLEY_MCST_LCC_VERSION_CHECK(1,25,10)
01647     #define JSON_HEDLEY_PRINTF_FORMAT(string_idx,first_to_check) __attribute__((__format__(__printf__,
      string_idx, first_to_check)))
01648 #elif JSON_HEDLEY_PELLES_VERSION_CHECK(6,0,0)
01649     #define JSON_HEDLEY_PRINTF_FORMAT(string_idx,first_to_check)
      __declspec(vaformat(printf,string_idx,first_to_check))
01650 #else
01651     #define JSON_HEDLEY_PRINTF_FORMAT(string_idx,first_to_check)
01652 #endif
01653
01654 #if defined(JSON_HEDLEY_CONSTEXPR)
01655     #undef JSON_HEDLEY_CONSTEXPR
01656 #endif
01657 #if defined(__cplusplus)
01658     #if __cplusplus >= 201103L
01659         #define JSON_HEDLEY_CONSTEXPR JSON_HEDLEY_DIAGNOSTIC_DISABLE_CPP98_COMPAT_WRAP_(constexpr)
01660     #endif
01661 #endif
01662 #if !defined(JSON_HEDLEY_CONSTEXPR)
01663     #define JSON_HEDLEY_CONSTEXPR
01664 #endif
01665
01666 #if defined(JSON_HEDLEY_PREDICT)
01667     #undef JSON_HEDLEY_PREDICT
01668 #endif
01669 #if defined(JSON_HEDLEY_LIKELY)
01670     #undef JSON_HEDLEY_LIKELY
01671 #endif
01672 #if defined(JSON_HEDLEY_UNLIKELY)
01673     #undef JSON_HEDLEY_UNLIKELY
01674 #endif
01675 #if defined(JSON_HEDLEY_UNPREDICTABLE)
01676     #undef JSON_HEDLEY_UNPREDICTABLE
01677 #endif
01678 #if JSON_HEDLEY_HAS_BUILTIN(__builtin_unpredictable)
01679     #define JSON_HEDLEY_UNPREDICTABLE(expr) __builtin_unpredictable((expr))
01680 #endif
01681 #if \
01682   (JSON_HEDLEY_HAS_BUILTIN(__builtin_expect_with_probability) && !defined(JSON_HEDLEY_PGI_VERSION)) ||
      \
01683   JSON_HEDLEY_GCC_VERSION_CHECK(9,0,0) || \
01684   JSON_HEDLEY_MCST_LCC_VERSION_CHECK(1,25,10)
01685 #  define JSON_HEDLEY_PREDICT(expr, value, probability) __builtin_expect_with_probability(  (expr),
      (value), (probability))
01686 #  define JSON_HEDLEY_PREDICT_TRUE(expr, probability)   __builtin_expect_with_probability(!!(expr),
      1   , (probability))
01687 #  define JSON_HEDLEY_PREDICT_FALSE(expr, probability)  __builtin_expect_with_probability(!!(expr),
      0   , (probability))
01688 #  define JSON_HEDLEY_LIKELY(expr)                      __builtin_expect                 (!!(expr),
      1                   )
01689 #  define JSON_HEDLEY_UNLIKELY(expr)                    __builtin_expect                 (!!(expr),
      0                   )
01690 #elif \
01691   (JSON_HEDLEY_HAS_BUILTIN(__builtin_expect) && !defined(JSON_HEDLEY_INTEL_CL_VERSION)) || \
01692   JSON_HEDLEY_GCC_VERSION_CHECK(3,0,0) || \
01693   JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0) || \
01694   (JSON_HEDLEY_SUNPRO_VERSION_CHECK(5,15,0) && defined(__cplusplus)) || \
01695   JSON_HEDLEY_ARM_VERSION_CHECK(4,1,0) || \
01696   JSON_HEDLEY_IBM_VERSION_CHECK(10,1,0) || \
01697   JSON_HEDLEY_TI_VERSION_CHECK(15,12,0) || \
01698   JSON_HEDLEY_TI_ARMCL_VERSION_CHECK(4,7,0) || \
01699   JSON_HEDLEY_TI_CL430_VERSION_CHECK(3,1,0) || \
01700   JSON_HEDLEY_TI_CL2000_VERSION_CHECK(6,1,0) || \
01701   JSON_HEDLEY_TI_CL6X_VERSION_CHECK(6,1,0) || \
01702   JSON_HEDLEY_TI_CL7X_VERSION_CHECK(1,2,0) || \
01703   JSON_HEDLEY_TI_CLPRU_VERSION_CHECK(2,1,0) || \
01704   JSON_HEDLEY_TINYC_VERSION_CHECK(0,9,27) || \
01705   JSON_HEDLEY_CRAY_VERSION_CHECK(8,1,0) || \
01706   JSON_HEDLEY_MCST_LCC_VERSION_CHECK(1,25,10)
01707 #  define JSON_HEDLEY_PREDICT(expr, expected, probability) \
01708     (((probability) >= 0.9) ? __builtin_expect((expr), (expected)) : (JSON_HEDLEY_STATIC_CAST(void,
      expected), (expr)))
01709 #  define JSON_HEDLEY_PREDICT_TRUE(expr, probability) \
01710     (__extension__ ({ \
```

```
01711         double hedley_probability_ = (probability); \
01712         ((hedley_probability_ >= 0.9) ? __builtin_expect(!!(expr), 1) : ((hedley_probability_ <= 0.1)
    ? __builtin_expect(!!(expr), 0) : !!(expr))); \
01713     }))
01714 #  define JSON_HEDLEY_PREDICT_FALSE(expr, probability) \
01715     (__extension__ ({ \
01716         double hedley_probability_ = (probability); \
01717         ((hedley_probability_ >= 0.9) ? __builtin_expect(!!(expr), 0) : ((hedley_probability_ <= 0.1)
    ? __builtin_expect(!!(expr), 1) : !!(expr))); \
01718     }))
01719 #  define JSON_HEDLEY_LIKELY(expr)   __builtin_expect(!!(expr), 1)
01720 #  define JSON_HEDLEY_UNLIKELY(expr) __builtin_expect(!!(expr), 0)
01721 #else
01722 #  define JSON_HEDLEY_PREDICT(expr, expected, probability) (JSON_HEDLEY_STATIC_CAST(void, expected),
    (expr))
01723 #  define JSON_HEDLEY_PREDICT_TRUE(expr, probability) (!!(expr))
01724 #  define JSON_HEDLEY_PREDICT_FALSE(expr, probability) (!!(expr))
01725 #  define JSON_HEDLEY_LIKELY(expr) (!!(expr))
01726 #  define JSON_HEDLEY_UNLIKELY(expr) (!!(expr))
01727 #endif
01728 #if !defined(JSON_HEDLEY_UNPREDICTABLE)
01729     #define JSON_HEDLEY_UNPREDICTABLE(expr) JSON_HEDLEY_PREDICT(expr, 1, 0.5)
01730 #endif
01731
01732 #if defined(JSON_HEDLEY_MALLOC)
01733     #undef JSON_HEDLEY_MALLOC
01734 #endif
01735 #if \
01736     JSON_HEDLEY_HAS_ATTRIBUTE(malloc) || \
01737     JSON_HEDLEY_GCC_VERSION_CHECK(3,1,0) || \
01738     JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0) || \
01739     JSON_HEDLEY_SUNPRO_VERSION_CHECK(5,11,0) || \
01740     JSON_HEDLEY_ARM_VERSION_CHECK(4,1,0) || \
01741     JSON_HEDLEY_IBM_VERSION_CHECK(12,1,0) || \
01742     JSON_HEDLEY_TI_VERSION_CHECK(15,12,0) || \
01743     (JSON_HEDLEY_TI_ARMCL_VERSION_CHECK(4,8,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01744     JSON_HEDLEY_TI_ARMCL_VERSION_CHECK(5,2,0) || \
01745     (JSON_HEDLEY_TI_CL2000_VERSION_CHECK(6,0,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01746     JSON_HEDLEY_TI_CL2000_VERSION_CHECK(6,4,0) || \
01747     (JSON_HEDLEY_TI_CL430_VERSION_CHECK(4,0,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01748     JSON_HEDLEY_TI_CL430_VERSION_CHECK(4,3,0) || \
01749     (JSON_HEDLEY_TI_CL6X_VERSION_CHECK(7,2,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01750     JSON_HEDLEY_TI_CL6X_VERSION_CHECK(7,5,0) || \
01751     JSON_HEDLEY_TI_CL7X_VERSION_CHECK(1,2,0) || \
01752     JSON_HEDLEY_TI_CLPRU_VERSION_CHECK(2,1,0) || \
01753     JSON_HEDLEY_MCST_LCC_VERSION_CHECK(1,25,10)
01754     #define JSON_HEDLEY_MALLOC __attribute__((__malloc__))
01755 #elif JSON_HEDLEY_SUNPRO_VERSION_CHECK(5,10,0)
01756     #define JSON_HEDLEY_MALLOC _Pragma("returns_new_memory")
01757 #elif \
01758     JSON_HEDLEY_MSVC_VERSION_CHECK(14,0,0) || \
01759     JSON_HEDLEY_INTEL_CL_VERSION_CHECK(2021,1,0)
01760     #define JSON_HEDLEY_MALLOC __declspec(restrict)
01761 #else
01762     #define JSON_HEDLEY_MALLOC
01763 #endif
01764
01765 #if defined(JSON_HEDLEY_PURE)
01766     #undef JSON_HEDLEY_PURE
01767 #endif
01768 #if \
01769   JSON_HEDLEY_HAS_ATTRIBUTE(pure) || \
01770   JSON_HEDLEY_GCC_VERSION_CHECK(2,96,0) || \
01771   JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0) || \
01772   JSON_HEDLEY_SUNPRO_VERSION_CHECK(5,11,0) || \
01773   JSON_HEDLEY_ARM_VERSION_CHECK(4,1,0) || \
01774   JSON_HEDLEY_IBM_VERSION_CHECK(10,1,0) || \
01775   JSON_HEDLEY_TI_VERSION_CHECK(15,12,0) || \
01776   (JSON_HEDLEY_TI_ARMCL_VERSION_CHECK(4,8,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01777   JSON_HEDLEY_TI_ARMCL_VERSION_CHECK(5,2,0) || \
01778   (JSON_HEDLEY_TI_CL2000_VERSION_CHECK(6,0,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01779   JSON_HEDLEY_TI_CL2000_VERSION_CHECK(6,4,0) || \
01780   (JSON_HEDLEY_TI_CL430_VERSION_CHECK(4,0,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01781   JSON_HEDLEY_TI_CL430_VERSION_CHECK(4,3,0) || \
01782   (JSON_HEDLEY_TI_CL6X_VERSION_CHECK(7,2,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01783   JSON_HEDLEY_TI_CL6X_VERSION_CHECK(7,5,0) || \
01784   JSON_HEDLEY_TI_CL7X_VERSION_CHECK(1,2,0) || \
01785   JSON_HEDLEY_TI_CLPRU_VERSION_CHECK(2,1,0) || \
01786   JSON_HEDLEY_PGI_VERSION_CHECK(17,10,0) || \
01787   JSON_HEDLEY_MCST_LCC_VERSION_CHECK(1,25,10)
01788 #  define JSON_HEDLEY_PURE __attribute__((__pure__))
01789 #elif JSON_HEDLEY_SUNPRO_VERSION_CHECK(5,10,0)
01790 #  define JSON_HEDLEY_PURE _Pragma("does_not_write_global_data")
01791 #elif defined(__cplusplus) && \
01792     ( \
01793       JSON_HEDLEY_TI_CL430_VERSION_CHECK(2,0,1) || \
01794       JSON_HEDLEY_TI_CL6X_VERSION_CHECK(4,0,0) || \
```

```
01795       JSON_HEDLEY_TI_CL7X_VERSION_CHECK(1,2,0) \
01796     )
01797 #  define JSON_HEDLEY_PURE _Pragma("FUNC_IS_PURE;")
01798 #else
01799 #  define JSON_HEDLEY_PURE
01800 #endif
01801
01802 #if defined(JSON_HEDLEY_CONST)
01803     #undef JSON_HEDLEY_CONST
01804 #endif
01805 #if \
01806     JSON_HEDLEY_HAS_ATTRIBUTE(const) || \
01807     JSON_HEDLEY_GCC_VERSION_CHECK(2,5,0) || \
01808     JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0) || \
01809     JSON_HEDLEY_SUNPRO_VERSION_CHECK(5,11,0) || \
01810     JSON_HEDLEY_ARM_VERSION_CHECK(4,1,0) || \
01811     JSON_HEDLEY_IBM_VERSION_CHECK(10,1,0) || \
01812     JSON_HEDLEY_TI_VERSION_CHECK(15,12,0) || \
01813     (JSON_HEDLEY_TI_ARMCL_VERSION_CHECK(4,8,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01814     JSON_HEDLEY_TI_ARMCL_VERSION_CHECK(5,2,0) || \
01815     (JSON_HEDLEY_TI_CL2000_VERSION_CHECK(6,0,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01816     JSON_HEDLEY_TI_CL2000_VERSION_CHECK(6,4,0) || \
01817     (JSON_HEDLEY_TI_CL430_VERSION_CHECK(4,0,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01818     JSON_HEDLEY_TI_CL430_VERSION_CHECK(4,3,0) || \
01819     (JSON_HEDLEY_TI_CL6X_VERSION_CHECK(7,2,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01820     JSON_HEDLEY_TI_CL6X_VERSION_CHECK(7,5,0) || \
01821     JSON_HEDLEY_TI_CL7X_VERSION_CHECK(1,2,0) || \
01822     JSON_HEDLEY_TI_CLPRU_VERSION_CHECK(2,1,0) || \
01823     JSON_HEDLEY_PGI_VERSION_CHECK(17,10,0) || \
01824     JSON_HEDLEY_MCST_LCC_VERSION_CHECK(1,25,10)
01825     #define JSON_HEDLEY_CONST __attribute__((__const__))
01826 #elif \
01827     JSON_HEDLEY_SUNPRO_VERSION_CHECK(5,10,0)
01828     #define JSON_HEDLEY_CONST _Pragma("no_side_effect")
01829 #else
01830     #define JSON_HEDLEY_CONST JSON_HEDLEY_PURE
01831 #endif
01832
01833 #if defined(JSON_HEDLEY_RESTRICT)
01834     #undef JSON_HEDLEY_RESTRICT
01835 #endif
01836 #if defined(__STDC_VERSION__) && (__STDC_VERSION__ >= 199901L) && !defined(__cplusplus)
01837     #define JSON_HEDLEY_RESTRICT restrict
01838 #elif \
01839     JSON_HEDLEY_GCC_VERSION_CHECK(3,1,0) || \
01840     JSON_HEDLEY_MSVC_VERSION_CHECK(14,0,0) || \
01841     JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0) || \
01842     JSON_HEDLEY_INTEL_CL_VERSION_CHECK(2021,1,0) || \
01843     JSON_HEDLEY_ARM_VERSION_CHECK(4,1,0) || \
01844     JSON_HEDLEY_IBM_VERSION_CHECK(10,1,0) || \
01845     JSON_HEDLEY_PGI_VERSION_CHECK(17,10,0) || \
01846     JSON_HEDLEY_TI_CL430_VERSION_CHECK(4,3,0) || \
01847     JSON_HEDLEY_TI_CL2000_VERSION_CHECK(6,2,4) || \
01848     JSON_HEDLEY_TI_CL6X_VERSION_CHECK(8,1,0) || \
01849     JSON_HEDLEY_TI_CL7X_VERSION_CHECK(1,2,0) || \
01850     (JSON_HEDLEY_SUNPRO_VERSION_CHECK(5,14,0) && defined(__cplusplus)) || \
01851     JSON_HEDLEY_IAR_VERSION_CHECK(8,0,0) || \
01852     defined(__clang__) || \
01853     JSON_HEDLEY_MCST_LCC_VERSION_CHECK(1,25,10)
01854     #define JSON_HEDLEY_RESTRICT __restrict
01855 #elif JSON_HEDLEY_SUNPRO_VERSION_CHECK(5,3,0) && !defined(__cplusplus)
01856     #define JSON_HEDLEY_RESTRICT _Restrict
01857 #else
01858     #define JSON_HEDLEY_RESTRICT
01859 #endif
01860
01861 #if defined(JSON_HEDLEY_INLINE)
01862     #undef JSON_HEDLEY_INLINE
01863 #endif
01864 #if \
01865     (defined(__STDC_VERSION__) && (__STDC_VERSION__ >= 199901L)) || \
01866     (defined(__cplusplus) && (__cplusplus >= 199711L))
01867     #define JSON_HEDLEY_INLINE inline
01868 #elif \
01869     defined(JSON_HEDLEY_GCC_VERSION) || \
01870     JSON_HEDLEY_ARM_VERSION_CHECK(6,2,0)
01871     #define JSON_HEDLEY_INLINE __inline__
01872 #elif \
01873     JSON_HEDLEY_MSVC_VERSION_CHECK(12,0,0) || \
01874     JSON_HEDLEY_INTEL_CL_VERSION_CHECK(2021,1,0) || \
01875     JSON_HEDLEY_ARM_VERSION_CHECK(4,1,0) || \
01876     JSON_HEDLEY_TI_ARMCL_VERSION_CHECK(5,1,0) || \
01877     JSON_HEDLEY_TI_CL430_VERSION_CHECK(3,1,0) || \
01878     JSON_HEDLEY_TI_CL2000_VERSION_CHECK(6,2,0) || \
01879     JSON_HEDLEY_TI_CL6X_VERSION_CHECK(8,0,0) || \
01880     JSON_HEDLEY_TI_CL7X_VERSION_CHECK(1,2,0) || \
01881     JSON_HEDLEY_TI_CLPRU_VERSION_CHECK(2,1,0) || \
```

```
01882     JSON_HEDLEY_MCST_LCC_VERSION_CHECK(1,25,10)
01883     #define JSON_HEDLEY_INLINE __inline
01884 #else
01885     #define JSON_HEDLEY_INLINE
01886 #endif
01887
01888 #if defined(JSON_HEDLEY_ALWAYS_INLINE)
01889     #undef JSON_HEDLEY_ALWAYS_INLINE
01890 #endif
01891 #if \
01892   JSON_HEDLEY_HAS_ATTRIBUTE(always_inline) || \
01893   JSON_HEDLEY_GCC_VERSION_CHECK(4,0,0) || \
01894   JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0) || \
01895   JSON_HEDLEY_SUNPRO_VERSION_CHECK(5,11,0) || \
01896   JSON_HEDLEY_ARM_VERSION_CHECK(4,1,0) || \
01897   JSON_HEDLEY_IBM_VERSION_CHECK(10,1,0) || \
01898   JSON_HEDLEY_TI_VERSION_CHECK(15,12,0) || \
01899   (JSON_HEDLEY_TI_ARMCL_VERSION_CHECK(4,8,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01900   JSON_HEDLEY_TI_ARMCL_VERSION_CHECK(5,2,0) || \
01901   (JSON_HEDLEY_TI_CL2000_VERSION_CHECK(6,0,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01902   JSON_HEDLEY_TI_CL2000_VERSION_CHECK(6,4,0) || \
01903   (JSON_HEDLEY_TI_CL430_VERSION_CHECK(4,0,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01904   JSON_HEDLEY_TI_CL430_VERSION_CHECK(4,3,0) || \
01905   (JSON_HEDLEY_TI_CL6X_VERSION_CHECK(7,2,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01906   JSON_HEDLEY_TI_CL6X_VERSION_CHECK(7,5,0) || \
01907   JSON_HEDLEY_TI_CL7X_VERSION_CHECK(1,2,0) || \
01908   JSON_HEDLEY_TI_CLPRU_VERSION_CHECK(2,1,0) || \
01909   JSON_HEDLEY_MCST_LCC_VERSION_CHECK(1,25,10) || \
01910   JSON_HEDLEY_IAR_VERSION_CHECK(8,10,0)
01911 #  define JSON_HEDLEY_ALWAYS_INLINE __attribute__((__always_inline__)) JSON_HEDLEY_INLINE
01912 #elif \
01913   JSON_HEDLEY_MSVC_VERSION_CHECK(12,0,0) || \
01914   JSON_HEDLEY_INTEL_CL_VERSION_CHECK(2021,1,0)
01915 #  define JSON_HEDLEY_ALWAYS_INLINE __forceinline
01916 #elif defined(__cplusplus) && \
01917     ( \
01918       JSON_HEDLEY_TI_ARMCL_VERSION_CHECK(5,2,0) || \
01919       JSON_HEDLEY_TI_CL430_VERSION_CHECK(4,3,0) || \
01920       JSON_HEDLEY_TI_CL2000_VERSION_CHECK(6,4,0) || \
01921       JSON_HEDLEY_TI_CL6X_VERSION_CHECK(6,1,0) || \
01922       JSON_HEDLEY_TI_CL7X_VERSION_CHECK(1,2,0) || \
01923       JSON_HEDLEY_TI_CLPRU_VERSION_CHECK(2,1,0) \
01924     )
01925 #  define JSON_HEDLEY_ALWAYS_INLINE _Pragma("FUNC_ALWAYS_INLINE;")
01926 #elif JSON_HEDLEY_IAR_VERSION_CHECK(8,0,0)
01927 #  define JSON_HEDLEY_ALWAYS_INLINE _Pragma("inline=forced")
01928 #else
01929 #  define JSON_HEDLEY_ALWAYS_INLINE JSON_HEDLEY_INLINE
01930 #endif
01931
01932 #if defined(JSON_HEDLEY_NEVER_INLINE)
01933     #undef JSON_HEDLEY_NEVER_INLINE
01934 #endif
01935 #if \
01936     JSON_HEDLEY_HAS_ATTRIBUTE(noinline) || \
01937     JSON_HEDLEY_GCC_VERSION_CHECK(4,0,0) || \
01938     JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0) || \
01939     JSON_HEDLEY_SUNPRO_VERSION_CHECK(5,11,0) || \
01940     JSON_HEDLEY_ARM_VERSION_CHECK(4,1,0) || \
01941     JSON_HEDLEY_IBM_VERSION_CHECK(10,1,0) || \
01942     JSON_HEDLEY_TI_VERSION_CHECK(15,12,0) || \
01943     (JSON_HEDLEY_TI_ARMCL_VERSION_CHECK(4,8,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01944     JSON_HEDLEY_TI_ARMCL_VERSION_CHECK(5,2,0) || \
01945     (JSON_HEDLEY_TI_CL2000_VERSION_CHECK(6,0,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01946     JSON_HEDLEY_TI_CL2000_VERSION_CHECK(6,4,0) || \
01947     (JSON_HEDLEY_TI_CL430_VERSION_CHECK(4,0,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01948     JSON_HEDLEY_TI_CL430_VERSION_CHECK(4,3,0) || \
01949     (JSON_HEDLEY_TI_CL6X_VERSION_CHECK(7,2,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01950     JSON_HEDLEY_TI_CL6X_VERSION_CHECK(7,5,0) || \
01951     JSON_HEDLEY_TI_CL7X_VERSION_CHECK(1,2,0) || \
01952     JSON_HEDLEY_TI_CLPRU_VERSION_CHECK(2,1,0) || \
01953     JSON_HEDLEY_MCST_LCC_VERSION_CHECK(1,25,10) || \
01954     JSON_HEDLEY_IAR_VERSION_CHECK(8,10,0)
01955     #define JSON_HEDLEY_NEVER_INLINE __attribute__((__noinline__))
01956 #elif \
01957     JSON_HEDLEY_MSVC_VERSION_CHECK(13,10,0) || \
01958     JSON_HEDLEY_INTEL_CL_VERSION_CHECK(2021,1,0)
01959     #define JSON_HEDLEY_NEVER_INLINE __declspec(noinline)
01960 #elif JSON_HEDLEY_PGI_VERSION_CHECK(10,2,0)
01961     #define JSON_HEDLEY_NEVER_INLINE _Pragma("noinline")
01962 #elif JSON_HEDLEY_TI_CL6X_VERSION_CHECK(6,0,0) && defined(__cplusplus)
01963     #define JSON_HEDLEY_NEVER_INLINE _Pragma("FUNC_CANNOT_INLINE;")
01964 #elif JSON_HEDLEY_IAR_VERSION_CHECK(8,0,0)
01965     #define JSON_HEDLEY_NEVER_INLINE _Pragma("inline=never")
01966 #elif JSON_HEDLEY_COMPCERT_VERSION_CHECK(3,2,0)
01967     #define JSON_HEDLEY_NEVER_INLINE __attribute((noinline))
01968 #elif JSON_HEDLEY_PELLES_VERSION_CHECK(9,0,0)
```

```
01969     #define JSON_HEDLEY_NEVER_INLINE __declspec(noinline)
01970 #else
01971     #define JSON_HEDLEY_NEVER_INLINE
01972 #endif
01973
01974 #if defined(JSON_HEDLEY_PRIVATE)
01975     #undef JSON_HEDLEY_PRIVATE
01976 #endif
01977 #if defined(JSON_HEDLEY_PUBLIC)
01978     #undef JSON_HEDLEY_PUBLIC
01979 #endif
01980 #if defined(JSON_HEDLEY_IMPORT)
01981     #undef JSON_HEDLEY_IMPORT
01982 #endif
01983 #if defined(_WIN32) || defined(__CYGWIN__)
01984 #  define JSON_HEDLEY_PRIVATE
01985 #  define JSON_HEDLEY_PUBLIC   __declspec(dllexport)
01986 #  define JSON_HEDLEY_IMPORT   __declspec(dllimport)
01987 #else
01988 #  if \
01989     JSON_HEDLEY_HAS_ATTRIBUTE(visibility) || \
01990     JSON_HEDLEY_GCC_VERSION_CHECK(3,3,0) || \
01991     JSON_HEDLEY_SUNPRO_VERSION_CHECK(5,11,0) || \
01992     JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0) || \
01993     JSON_HEDLEY_ARM_VERSION_CHECK(4,1,0) || \
01994     JSON_HEDLEY_IBM_VERSION_CHECK(13,1,0) || \
01995     ( \
01996       defined(__TI_EABI__) && \
01997       ( \
01998         (JSON_HEDLEY_TI_CL6X_VERSION_CHECK(7,2,0) && defined(__TI_GNU_ATTRIBUTE_SUPPORT__)) || \
01999         JSON_HEDLEY_TI_CL6X_VERSION_CHECK(7,5,0) \
02000       ) \
02001     ) || \
02002     JSON_HEDLEY_MCST_LCC_VERSION_CHECK(1,25,10)
02003 #    define JSON_HEDLEY_PRIVATE __attribute__((__visibility__("hidden")))
02004 #    define JSON_HEDLEY_PUBLIC  __attribute__((__visibility__("default")))
02005 #  else
02006 #    define JSON_HEDLEY_PRIVATE
02007 #    define JSON_HEDLEY_PUBLIC
02008 #  endif
02009 #  define JSON_HEDLEY_IMPORT    extern
02010 #endif
02011
02012 #if defined(JSON_HEDLEY_NO_THROW)
02013     #undef JSON_HEDLEY_NO_THROW
02014 #endif
02015 #if \
02016     JSON_HEDLEY_HAS_ATTRIBUTE(nothrow) || \
02017     JSON_HEDLEY_GCC_VERSION_CHECK(3,3,0) || \
02018     JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0) || \
02019     JSON_HEDLEY_MCST_LCC_VERSION_CHECK(1,25,10)
02020     #define JSON_HEDLEY_NO_THROW __attribute__((__nothrow__))
02021 #elif \
02022     JSON_HEDLEY_MSVC_VERSION_CHECK(13,1,0) || \
02023     JSON_HEDLEY_INTEL_CL_VERSION_CHECK(2021,1,0) || \
02024     JSON_HEDLEY_ARM_VERSION_CHECK(4,1,0)
02025     #define JSON_HEDLEY_NO_THROW __declspec(nothrow)
02026 #else
02027     #define JSON_HEDLEY_NO_THROW
02028 #endif
02029
02030 #if defined(JSON_HEDLEY_FALL_THROUGH)
02031     #undef JSON_HEDLEY_FALL_THROUGH
02032 #endif
02033 #if \
02034     JSON_HEDLEY_HAS_ATTRIBUTE(fallthrough) || \
02035     JSON_HEDLEY_GCC_VERSION_CHECK(7,0,0) || \
02036     JSON_HEDLEY_MCST_LCC_VERSION_CHECK(1,25,10)
02037     #define JSON_HEDLEY_FALL_THROUGH __attribute__((__fallthrough__))
02038 #elif JSON_HEDLEY_HAS_CPP_ATTRIBUTE_NS(clang,fallthrough)
02039     #define JSON_HEDLEY_FALL_THROUGH
      JSON_HEDLEY_DIAGNOSTIC_DISABLE_CPP98_COMPAT_WRAP_([[clang::fallthrough]])
02040 #elif JSON_HEDLEY_HAS_CPP_ATTRIBUTE(fallthrough)
02041     #define JSON_HEDLEY_FALL_THROUGH
      JSON_HEDLEY_DIAGNOSTIC_DISABLE_CPP98_COMPAT_WRAP_([[fallthrough]])
02042 #elif defined(__fallthrough) /* SAL */
02043     #define JSON_HEDLEY_FALL_THROUGH __fallthrough
02044 #else
02045     #define JSON_HEDLEY_FALL_THROUGH
02046 #endif
02047
02048 #if defined(JSON_HEDLEY_RETURNS_NON_NULL)
02049     #undef JSON_HEDLEY_RETURNS_NON_NULL
02050 #endif
02051 #if \
02052     JSON_HEDLEY_HAS_ATTRIBUTE(returns_nonnull) || \
02053     JSON_HEDLEY_GCC_VERSION_CHECK(4,9,0) || \
```

```
02054     JSON_HEDLEY_MCST_LCC_VERSION_CHECK(1,25,10)
02055     #define JSON_HEDLEY_RETURNS_NON_NULL __attribute__((__returns_nonnull__))
02056 #elif defined(_Ret_notnull_) /* SAL */
02057     #define JSON_HEDLEY_RETURNS_NON_NULL _Ret_notnull_
02058 #else
02059     #define JSON_HEDLEY_RETURNS_NON_NULL
02060 #endif
02061
02062 #if defined(JSON_HEDLEY_ARRAY_PARAM)
02063     #undef JSON_HEDLEY_ARRAY_PARAM
02064 #endif
02065 #if \
02066     defined(__STDC_VERSION__) && (__STDC_VERSION__ >= 199901L) && \
02067     !defined(__STDC_NO_VLA__) && \
02068     !defined(__cplusplus) && \
02069     !defined(JSON_HEDLEY_PGI_VERSION) && \
02070     !defined(JSON_HEDLEY_TINYC_VERSION)
02071     #define JSON_HEDLEY_ARRAY_PARAM(name) (name)
02072 #else
02073     #define JSON_HEDLEY_ARRAY_PARAM(name)
02074 #endif
02075
02076 #if defined(JSON_HEDLEY_IS_CONSTANT)
02077     #undef JSON_HEDLEY_IS_CONSTANT
02078 #endif
02079 #if defined(JSON_HEDLEY_REQUIRE_CONSTEXPR)
02080     #undef JSON_HEDLEY_REQUIRE_CONSTEXPR
02081 #endif
02082 /* JSON_HEDLEY_IS_CONSTEXPR_ is for
02083    HEDLEY INTERNAL USE ONLY.  API subject to change without notice. */
02084 #if defined(JSON_HEDLEY_IS_CONSTEXPR_)
02085     #undef JSON_HEDLEY_IS_CONSTEXPR_
02086 #endif
02087 #if \
02088     JSON_HEDLEY_HAS_BUILTIN(__builtin_constant_p) || \
02089     JSON_HEDLEY_GCC_VERSION_CHECK(3,4,0) || \
02090     JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0) || \
02091     JSON_HEDLEY_TINYC_VERSION_CHECK(0,9,19) || \
02092     JSON_HEDLEY_ARM_VERSION_CHECK(4,1,0) || \
02093     JSON_HEDLEY_IBM_VERSION_CHECK(13,1,0) || \
02094     JSON_HEDLEY_TI_CL6X_VERSION_CHECK(6,1,0) || \
02095     (JSON_HEDLEY_SUNPRO_VERSION_CHECK(5,10,0) && !defined(__cplusplus)) || \
02096     JSON_HEDLEY_CRAY_VERSION_CHECK(8,1,0) || \
02097     JSON_HEDLEY_MCST_LCC_VERSION_CHECK(1,25,10)
02098     #define JSON_HEDLEY_IS_CONSTANT(expr) __builtin_constant_p(expr)
02099 #endif
02100 #if !defined(__cplusplus)
02101 #  if \
02102       JSON_HEDLEY_HAS_BUILTIN(__builtin_types_compatible_p) || \
02103       JSON_HEDLEY_GCC_VERSION_CHECK(3,4,0) || \
02104       JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0) || \
02105       JSON_HEDLEY_IBM_VERSION_CHECK(13,1,0) || \
02106       JSON_HEDLEY_CRAY_VERSION_CHECK(8,1,0) || \
02107       JSON_HEDLEY_ARM_VERSION_CHECK(5,4,0) || \
02108       JSON_HEDLEY_TINYC_VERSION_CHECK(0,9,24)
02109 #if defined(__INTPTR_TYPE__)
02110     #define JSON_HEDLEY_IS_CONSTEXPR_(expr) __builtin_types_compatible_p(__typeof__((1 ? (void*)
      ((__INTPTR_TYPE__) ((expr) * 0)) : (int*) 0)), int*)
02111 #else
02112     #include <stdint.h>
02113     #define JSON_HEDLEY_IS_CONSTEXPR_(expr) __builtin_types_compatible_p(__typeof__((1 ? (void*)
      ((intptr_t) ((expr) * 0)) : (int*) 0)), int*)
02114 #endif
02115 #  elif \
02116       ( \
02117          defined(__STDC_VERSION__) && (__STDC_VERSION__ >= 201112L) && \
02118          !defined(JSON_HEDLEY_SUNPRO_VERSION) && \
02119          !defined(JSON_HEDLEY_PGI_VERSION) && \
02120          !defined(JSON_HEDLEY_IAR_VERSION)) || \
02121       (JSON_HEDLEY_HAS_EXTENSION(c_generic_selections) && !defined(JSON_HEDLEY_IAR_VERSION)) || \
02122       JSON_HEDLEY_GCC_VERSION_CHECK(4,9,0) || \
02123       JSON_HEDLEY_INTEL_VERSION_CHECK(17,0,0) || \
02124       JSON_HEDLEY_IBM_VERSION_CHECK(12,1,0) || \
02125       JSON_HEDLEY_ARM_VERSION_CHECK(5,3,0)
02126 #if defined(__INTPTR_TYPE__)
02127     #define JSON_HEDLEY_IS_CONSTEXPR_(expr) _Generic((1 ? (void*) ((__INTPTR_TYPE__) ((expr) * 0)) :
      (int*) 0), int*: 1, void*: 0)
02128 #else
02129     #include <stdint.h>
02130     #define JSON_HEDLEY_IS_CONSTEXPR_(expr) _Generic((1 ? (void*) ((intptr_t) * 0) : (int*) 0), int*:
      1, void*: 0)
02131 #endif
02132 #  elif \
02133       defined(JSON_HEDLEY_GCC_VERSION) || \
02134       defined(JSON_HEDLEY_INTEL_VERSION) || \
02135       defined(JSON_HEDLEY_TINYC_VERSION) || \
02136       defined(JSON_HEDLEY_TI_ARMCL_VERSION) || \
```

```
02137        JSON_HEDLEY_TI_CL430_VERSION_CHECK(18,12,0) || \
02138        defined(JSON_HEDLEY_TI_CL2000_VERSION) || \
02139        defined(JSON_HEDLEY_TI_CL6X_VERSION) || \
02140        defined(JSON_HEDLEY_TI_CL7X_VERSION) || \
02141        defined(JSON_HEDLEY_TI_CLPRU_VERSION) || \
02142        defined(__clang__)
02143 #    define JSON_HEDLEY_IS_CONSTEXPR_(expr) ( \
02144        sizeof(void) != \
02145        sizeof(*( \
02146                  1 ? \
02147                  ((void*) ((expr) * 0L) ) : \
02148 ((struct { char v[sizeof(void) * 2]; } *) 1) \
02149                ) \
02150              ) \
02151                                            )
02152 #  endif
02153 #endif
02154 #if defined(JSON_HEDLEY_IS_CONSTEXPR_)
02155     #if !defined(JSON_HEDLEY_IS_CONSTANT)
02156         #define JSON_HEDLEY_IS_CONSTANT(expr) JSON_HEDLEY_IS_CONSTEXPR_(expr)
02157     #endif
02158     #define JSON_HEDLEY_REQUIRE_CONSTEXPR(expr) (JSON_HEDLEY_IS_CONSTEXPR_(expr) ? (expr) : (-1))
02159 #else
02160     #if !defined(JSON_HEDLEY_IS_CONSTANT)
02161         #define JSON_HEDLEY_IS_CONSTANT(expr) (0)
02162     #endif
02163     #define JSON_HEDLEY_REQUIRE_CONSTEXPR(expr) (expr)
02164 #endif
02165
02166 #if defined(JSON_HEDLEY_BEGIN_C_DECLS)
02167     #undef JSON_HEDLEY_BEGIN_C_DECLS
02168 #endif
02169 #if defined(JSON_HEDLEY_END_C_DECLS)
02170     #undef JSON_HEDLEY_END_C_DECLS
02171 #endif
02172 #if defined(JSON_HEDLEY_C_DECL)
02173     #undef JSON_HEDLEY_C_DECL
02174 #endif
02175 #if defined(__cplusplus)
02176     #define JSON_HEDLEY_BEGIN_C_DECLS extern "C" {
02177     #define JSON_HEDLEY_END_C_DECLS }
02178     #define JSON_HEDLEY_C_DECL extern "C"
02179 #else
02180     #define JSON_HEDLEY_BEGIN_C_DECLS
02181     #define JSON_HEDLEY_END_C_DECLS
02182     #define JSON_HEDLEY_C_DECL
02183 #endif
02184
02185 #if defined(JSON_HEDLEY_STATIC_ASSERT)
02186     #undef JSON_HEDLEY_STATIC_ASSERT
02187 #endif
02188 #if \
02189   !defined(__cplusplus) && ( \
02190       (defined(__STDC_VERSION__) && (__STDC_VERSION__ >= 201112L)) || \
02191       (JSON_HEDLEY_HAS_FEATURE(c_static_assert) && !defined(JSON_HEDLEY_INTEL_CL_VERSION)) || \
02192       JSON_HEDLEY_GCC_VERSION_CHECK(6,0,0) || \
02193       JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0) || \
02194       defined(_Static_assert) \
02195     )
02196 #  define JSON_HEDLEY_STATIC_ASSERT(expr, message) _Static_assert(expr, message)
02197 #elif \
02198   (defined(__cplusplus) && (__cplusplus >= 201103L)) || \
02199   JSON_HEDLEY_MSVC_VERSION_CHECK(16,0,0) || \
02200   JSON_HEDLEY_INTEL_CL_VERSION_CHECK(2021,1,0)
02201 #  define JSON_HEDLEY_STATIC_ASSERT(expr, message)
      JSON_HEDLEY_DIAGNOSTIC_DISABLE_CPP98_COMPAT_WRAP_(static_assert(expr, message))
02202 #else
02203 #  define JSON_HEDLEY_STATIC_ASSERT(expr, message)
02204 #endif
02205
02206 #if defined(JSON_HEDLEY_NULL)
02207     #undef JSON_HEDLEY_NULL
02208 #endif
02209 #if defined(__cplusplus)
02210     #if __cplusplus >= 201103L
02211         #define JSON_HEDLEY_NULL JSON_HEDLEY_DIAGNOSTIC_DISABLE_CPP98_COMPAT_WRAP_(nullptr)
02212     #elif defined(NULL)
02213         #define JSON_HEDLEY_NULL NULL
02214     #else
02215         #define JSON_HEDLEY_NULL JSON_HEDLEY_STATIC_CAST(void*, 0)
02216     #endif
02217 #elif defined(NULL)
02218     #define JSON_HEDLEY_NULL NULL
02219 #else
02220     #define JSON_HEDLEY_NULL ((void*) 0)
02221 #endif
02222
```

```
02223 #if defined(JSON_HEDLEY_MESSAGE)
02224     #undef JSON_HEDLEY_MESSAGE
02225 #endif
02226 #if JSON_HEDLEY_HAS_WARNING("-Wunknown-pragmas")
02227 #  define JSON_HEDLEY_MESSAGE(msg) \
02228     JSON_HEDLEY_DIAGNOSTIC_PUSH \
02229     JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_PRAGMAS \
02230     JSON_HEDLEY_PRAGMA(message msg) \
02231     JSON_HEDLEY_DIAGNOSTIC_POP
02232 #elif \
02233   JSON_HEDLEY_GCC_VERSION_CHECK(4,4,0) || \
02234   JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0)
02235 #  define JSON_HEDLEY_MESSAGE(msg) JSON_HEDLEY_PRAGMA(message msg)
02236 #elif JSON_HEDLEY_CRAY_VERSION_CHECK(5,0,0)
02237 #  define JSON_HEDLEY_MESSAGE(msg) JSON_HEDLEY_PRAGMA(_CRI message msg)
02238 #elif JSON_HEDLEY_IAR_VERSION_CHECK(8,0,0)
02239 #  define JSON_HEDLEY_MESSAGE(msg) JSON_HEDLEY_PRAGMA(message(msg))
02240 #elif JSON_HEDLEY_PELLES_VERSION_CHECK(2,0,0)
02241 #  define JSON_HEDLEY_MESSAGE(msg) JSON_HEDLEY_PRAGMA(message(msg))
02242 #else
02243 #  define JSON_HEDLEY_MESSAGE(msg)
02244 #endif
02245
02246 #if defined(JSON_HEDLEY_WARNING)
02247     #undef JSON_HEDLEY_WARNING
02248 #endif
02249 #if JSON_HEDLEY_HAS_WARNING("-Wunknown-pragmas")
02250 #  define JSON_HEDLEY_WARNING(msg) \
02251     JSON_HEDLEY_DIAGNOSTIC_PUSH \
02252     JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_PRAGMAS \
02253     JSON_HEDLEY_PRAGMA(clang warning msg) \
02254     JSON_HEDLEY_DIAGNOSTIC_POP
02255 #elif \
02256   JSON_HEDLEY_GCC_VERSION_CHECK(4,8,0) || \
02257   JSON_HEDLEY_PGI_VERSION_CHECK(18,4,0) || \
02258   JSON_HEDLEY_INTEL_VERSION_CHECK(13,0,0)
02259 #  define JSON_HEDLEY_WARNING(msg) JSON_HEDLEY_PRAGMA(GCC warning msg)
02260 #elif \
02261   JSON_HEDLEY_MSVC_VERSION_CHECK(15,0,0) || \
02262   JSON_HEDLEY_INTEL_CL_VERSION_CHECK(2021,1,0)
02263 #  define JSON_HEDLEY_WARNING(msg) JSON_HEDLEY_PRAGMA(message(msg))
02264 #else
02265 #  define JSON_HEDLEY_WARNING(msg) JSON_HEDLEY_MESSAGE(msg)
02266 #endif
02267
02268 #if defined(JSON_HEDLEY_REQUIRE)
02269     #undef JSON_HEDLEY_REQUIRE
02270 #endif
02271 #if defined(JSON_HEDLEY_REQUIRE_MSG)
02272     #undef JSON_HEDLEY_REQUIRE_MSG
02273 #endif
02274 #if JSON_HEDLEY_HAS_ATTRIBUTE(diagnose_if)
02275 #  if JSON_HEDLEY_HAS_WARNING("-Wgcc-compat")
02276 #    define JSON_HEDLEY_REQUIRE(expr) \
02277     JSON_HEDLEY_DIAGNOSTIC_PUSH \
02278     _Pragma("clang diagnostic ignored \"-Wgcc-compat\"") \
02279     __attribute__((diagnose_if(!(expr), #expr, "error"))) \
02280     JSON_HEDLEY_DIAGNOSTIC_POP
02281 #    define JSON_HEDLEY_REQUIRE_MSG(expr,msg) \
02282     JSON_HEDLEY_DIAGNOSTIC_PUSH \
02283     _Pragma("clang diagnostic ignored \"-Wgcc-compat\"") \
02284     __attribute__((diagnose_if(!(expr), msg, "error"))) \
02285     JSON_HEDLEY_DIAGNOSTIC_POP
02286 #  else
02287 #    define JSON_HEDLEY_REQUIRE(expr) __attribute__((diagnose_if(!(expr), #expr, "error")))
02288 #    define JSON_HEDLEY_REQUIRE_MSG(expr,msg) __attribute__((diagnose_if(!(expr), msg, "error")))
02289 #  endif
02290 #else
02291 #  define JSON_HEDLEY_REQUIRE(expr)
02292 #  define JSON_HEDLEY_REQUIRE_MSG(expr,msg)
02293 #endif
02294
02295 #if defined(JSON_HEDLEY_FLAGS)
02296     #undef JSON_HEDLEY_FLAGS
02297 #endif
02298 #if JSON_HEDLEY_HAS_ATTRIBUTE(flag_enum) && (!defined(__cplusplus) ||
      JSON_HEDLEY_HAS_WARNING("-Wbitfield-enum-conversion"))
02299     #define JSON_HEDLEY_FLAGS __attribute__((__flag_enum__))
02300 #else
02301     #define JSON_HEDLEY_FLAGS
02302 #endif
02303
02304 #if defined(JSON_HEDLEY_FLAGS_CAST)
02305     #undef JSON_HEDLEY_FLAGS_CAST
02306 #endif
02307 #if JSON_HEDLEY_INTEL_VERSION_CHECK(19,0,0)
02308 #  define JSON_HEDLEY_FLAGS_CAST(T, expr) (__extension__ ({ \
```

```
02309         JSON_HEDLEY_DIAGNOSTIC_PUSH \
02310         _Pragma("warning(disable:188)") \
02311         ((T) (expr)); \
02312         JSON_HEDLEY_DIAGNOSTIC_POP \
02313     }))
02314 #else
02315 #  define JSON_HEDLEY_FLAGS_CAST(T, expr) JSON_HEDLEY_STATIC_CAST(T, expr)
02316 #endif
02317
02318 #if defined(JSON_HEDLEY_EMPTY_BASES)
02319     #undef JSON_HEDLEY_EMPTY_BASES
02320 #endif
02321 #if \
02322     (JSON_HEDLEY_MSVC_VERSION_CHECK(19,0,23918) && !JSON_HEDLEY_MSVC_VERSION_CHECK(20,0,0)) || \
02323     JSON_HEDLEY_INTEL_CL_VERSION_CHECK(2021,1,0)
02324     #define JSON_HEDLEY_EMPTY_BASES __declspec(empty_bases)
02325 #else
02326     #define JSON_HEDLEY_EMPTY_BASES
02327 #endif
02328
02329 /* Remaining macros are deprecated. */
02330
02331 #if defined(JSON_HEDLEY_GCC_NOT_CLANG_VERSION_CHECK)
02332     #undef JSON_HEDLEY_GCC_NOT_CLANG_VERSION_CHECK
02333 #endif
02334 #if defined(__clang__)
02335     #define JSON_HEDLEY_GCC_NOT_CLANG_VERSION_CHECK(major,minor,patch) (0)
02336 #else
02337     #define JSON_HEDLEY_GCC_NOT_CLANG_VERSION_CHECK(major,minor,patch)
      JSON_HEDLEY_GCC_VERSION_CHECK(major,minor,patch)
02338 #endif
02339
02340 #if defined(JSON_HEDLEY_CLANG_HAS_ATTRIBUTE)
02341     #undef JSON_HEDLEY_CLANG_HAS_ATTRIBUTE
02342 #endif
02343 #define JSON_HEDLEY_CLANG_HAS_ATTRIBUTE(attribute) JSON_HEDLEY_HAS_ATTRIBUTE(attribute)
02344
02345 #if defined(JSON_HEDLEY_CLANG_HAS_CPP_ATTRIBUTE)
02346     #undef JSON_HEDLEY_CLANG_HAS_CPP_ATTRIBUTE
02347 #endif
02348 #define JSON_HEDLEY_CLANG_HAS_CPP_ATTRIBUTE(attribute) JSON_HEDLEY_HAS_CPP_ATTRIBUTE(attribute)
02349
02350 #if defined(JSON_HEDLEY_CLANG_HAS_BUILTIN)
02351     #undef JSON_HEDLEY_CLANG_HAS_BUILTIN
02352 #endif
02353 #define JSON_HEDLEY_CLANG_HAS_BUILTIN(builtin) JSON_HEDLEY_HAS_BUILTIN(builtin)
02354
02355 #if defined(JSON_HEDLEY_CLANG_HAS_FEATURE)
02356     #undef JSON_HEDLEY_CLANG_HAS_FEATURE
02357 #endif
02358 #define JSON_HEDLEY_CLANG_HAS_FEATURE(feature) JSON_HEDLEY_HAS_FEATURE(feature)
02359
02360 #if defined(JSON_HEDLEY_CLANG_HAS_EXTENSION)
02361     #undef JSON_HEDLEY_CLANG_HAS_EXTENSION
02362 #endif
02363 #define JSON_HEDLEY_CLANG_HAS_EXTENSION(extension) JSON_HEDLEY_HAS_EXTENSION(extension)
02364
02365 #if defined(JSON_HEDLEY_CLANG_HAS_DECLSPEC_DECLSPEC_ATTRIBUTE)
02366     #undef JSON_HEDLEY_CLANG_HAS_DECLSPEC_DECLSPEC_ATTRIBUTE
02367 #endif
02368 #define JSON_HEDLEY_CLANG_HAS_DECLSPEC_ATTRIBUTE(attribute)
      JSON_HEDLEY_HAS_DECLSPEC_ATTRIBUTE(attribute)
02369
02370 #if defined(JSON_HEDLEY_CLANG_HAS_WARNING)
02371     #undef JSON_HEDLEY_CLANG_HAS_WARNING
02372 #endif
02373 #define JSON_HEDLEY_CLANG_HAS_WARNING(warning) JSON_HEDLEY_HAS_WARNING(warning)
02374
02375 #endif /* !defined(JSON_HEDLEY_VERSION) || (JSON_HEDLEY_VERSION < X) */
02376
02377
02378 // This file contains all internal macro definitions (except those affecting ABI)
02379 // You MUST include macro_unscope.hpp at the end of json.hpp to undef all of them
02380
02381 // #include <nlohmann/detail/abi_macros.hpp>
02382
02383
02384 // exclude unsupported compilers
02385 #if !defined(JSON_SKIP_UNSUPPORTED_COMPILER_CHECK)
02386     #if defined(__clang__)
02387         #if (__clang_major__ * 10000 + __clang_minor__ * 100 + __clang_patchlevel__) < 30400
02388             #error "unsupported Clang version - see
      https://github.com/nlohmann/json#supported-compilers"
02389         #endif
02390     #elif defined(__GNUC__) && !(defined(__ICC) || defined(__INTEL_COMPILER))
02391         #if (__GNUC__ * 10000 + __GNUC_MINOR__ * 100 + __GNUC_PATCHLEVEL__) < 40800
02392             #error "unsupported GCC version - see
```

```
    https://github.com/nlohmann/json#supported-compilers"
02393         #endif
02394     #endif
02395 #endif
02396
02397 // C++ language standard detection
02398 // if the user manually specified the used C++ version, this is skipped
02399 #if !defined(JSON_HAS_CPP_26) && !defined(JSON_HAS_CPP_23) && !defined(JSON_HAS_CPP_20) &&
    !defined(JSON_HAS_CPP_17) && !defined(JSON_HAS_CPP_14) && !defined(JSON_HAS_CPP_11)
02400     #if (defined(__cplusplus) && __cplusplus > 202302L) || (defined(_MSVC_LANG) && _MSVC_LANG >
    202302L)
02401         #define JSON_HAS_CPP_26
02402         #define JSON_HAS_CPP_23
02403         #define JSON_HAS_CPP_20
02404         #define JSON_HAS_CPP_17
02405         #define JSON_HAS_CPP_14
02406     #elif (defined(__cplusplus) && __cplusplus > 202002L) || (defined(_MSVC_LANG) && _MSVC_LANG >
    202002L)
02407         #define JSON_HAS_CPP_23
02408         #define JSON_HAS_CPP_20
02409         #define JSON_HAS_CPP_17
02410         #define JSON_HAS_CPP_14
02411     #elif (defined(__cplusplus) && __cplusplus > 201703L) || (defined(_MSVC_LANG) && _MSVC_LANG >
    201703L)
02412         #define JSON_HAS_CPP_20
02413         #define JSON_HAS_CPP_17
02414         #define JSON_HAS_CPP_14
02415     #elif (defined(__cplusplus) && __cplusplus > 201402L) || (defined(_HAS_CXX17) && _HAS_CXX17 == 1)
    // fix for issue #464
02416         #define JSON_HAS_CPP_17
02417         #define JSON_HAS_CPP_14
02418     #elif (defined(__cplusplus) && __cplusplus > 201103L) || (defined(_HAS_CXX14) && _HAS_CXX14 == 1)
02419         #define JSON_HAS_CPP_14
02420     #endif
02421     // the cpp 11 flag is always specified because it is the minimal required version
02422     #define JSON_HAS_CPP_11
02423 #endif
02424
02425 #ifdef __has_include
02426     #if __has_include(<version>)
02427         #include <version>
02428     #endif
02429 #endif
02430
02431 #if !defined(JSON_HAS_FILESYSTEM) && !defined(JSON_HAS_EXPERIMENTAL_FILESYSTEM)
02432     #ifdef JSON_HAS_CPP_17
02433         #if defined(__cpp_lib_filesystem)
02434             #define JSON_HAS_FILESYSTEM 1
02435         #elif defined(__cpp_lib_experimental_filesystem)
02436             #define JSON_HAS_EXPERIMENTAL_FILESYSTEM 1
02437         #elif !defined(__has_include)
02438             #define JSON_HAS_EXPERIMENTAL_FILESYSTEM 1
02439         #elif __has_include(<filesystem>)
02440             #define JSON_HAS_FILESYSTEM 1
02441         #elif __has_include(<experimental/filesystem>)
02442             #define JSON_HAS_EXPERIMENTAL_FILESYSTEM 1
02443         #endif
02444
02445         // std::filesystem does not work on MinGW GCC 8: https://sourceforge.net/p/mingw-w64/bugs/737/
02446         #if defined(__MINGW32__) && defined(__GNUC__) && __GNUC__ == 8
02447             #undef JSON_HAS_FILESYSTEM
02448             #undef JSON_HAS_EXPERIMENTAL_FILESYSTEM
02449         #endif
02450
02451         // no filesystem support before GCC 8: https://en.cppreference.com/w/cpp/compiler_support
02452         #if defined(__GNUC__) && !defined(__clang__) && __GNUC__ < 8
02453             #undef JSON_HAS_FILESYSTEM
02454             #undef JSON_HAS_EXPERIMENTAL_FILESYSTEM
02455         #endif
02456
02457         // no filesystem support before Clang 7: https://en.cppreference.com/w/cpp/compiler_support
02458         #if defined(__clang_major__) && __clang_major__ < 7
02459             #undef JSON_HAS_FILESYSTEM
02460             #undef JSON_HAS_EXPERIMENTAL_FILESYSTEM
02461         #endif
02462
02463         // no filesystem support before MSVC 19.14: https://en.cppreference.com/w/cpp/compiler_support
02464         #if defined(_MSC_VER) && _MSC_VER < 1914
02465             #undef JSON_HAS_FILESYSTEM
02466             #undef JSON_HAS_EXPERIMENTAL_FILESYSTEM
02467         #endif
02468
02469         // no filesystem support before iOS 13
02470         #if defined(__IPHONE_OS_VERSION_MIN_REQUIRED) && __IPHONE_OS_VERSION_MIN_REQUIRED < 130000
02471             #undef JSON_HAS_FILESYSTEM
02472             #undef JSON_HAS_EXPERIMENTAL_FILESYSTEM
02473         #endif
```

```
02474
02475         // no filesystem support before macOS Catalina
02476         #if defined(__MAC_OS_X_VERSION_MIN_REQUIRED) && __MAC_OS_X_VERSION_MIN_REQUIRED < 101500
02477             #undef JSON_HAS_FILESYSTEM
02478             #undef JSON_HAS_EXPERIMENTAL_FILESYSTEM
02479         #endif
02480     #endif
02481 #endif
02482
02483 #ifndef JSON_HAS_EXPERIMENTAL_FILESYSTEM
02484     #define JSON_HAS_EXPERIMENTAL_FILESYSTEM 0
02485 #endif
02486
02487 #ifndef JSON_HAS_FILESYSTEM
02488     #define JSON_HAS_FILESYSTEM 0
02489 #endif
02490
02491 #ifndef JSON_HAS_THREE_WAY_COMPARISON
02492     #if defined(__cpp_impl_three_way_comparison) && __cpp_impl_three_way_comparison >= 201907L \
02493         && defined(__cpp_lib_three_way_comparison) && __cpp_lib_three_way_comparison >= 201907L
02494         #define JSON_HAS_THREE_WAY_COMPARISON 1
02495     #else
02496         #define JSON_HAS_THREE_WAY_COMPARISON 0
02497     #endif
02498 #endif
02499
02500 #ifndef JSON_HAS_RANGES
02501     // ranges header shipping in GCC 11.1.0 (released 2021-04-27) has a syntax error
02502     #if defined(__GLIBCXX__) && __GLIBCXX__ == 20210427
02503         #define JSON_HAS_RANGES 0
02504     #elif defined(__cpp_lib_ranges)
02505         #define JSON_HAS_RANGES 1
02506     #else
02507         #define JSON_HAS_RANGES 0
02508     #endif
02509 #endif
02510
02511 #ifndef JSON_HAS_STATIC_RTTI
02512     #if !defined(_HAS_STATIC_RTTI) || _HAS_STATIC_RTTI != 0
02513         #define JSON_HAS_STATIC_RTTI 1
02514     #else
02515         #define JSON_HAS_STATIC_RTTI 0
02516     #endif
02517 #endif
02518
02519 #ifdef JSON_HAS_CPP_17
02520     #define JSON_INLINE_VARIABLE inline
02521 #else
02522     #define JSON_INLINE_VARIABLE
02523 #endif
02524
02525 #if JSON_HEDLEY_HAS_ATTRIBUTE(no_unique_address)
02526     #define JSON_NO_UNIQUE_ADDRESS [[no_unique_address]]
02527 #else
02528     #define JSON_NO_UNIQUE_ADDRESS
02529 #endif
02530
02531 // disable documentation warnings on clang
02532 #if defined(__clang__)
02533     #pragma clang diagnostic push
02534     #pragma clang diagnostic ignored "-Wdocumentation"
02535     #pragma clang diagnostic ignored "-Wdocumentation-unknown-command"
02536 #endif
02537
02538 // allow disabling exceptions
02539 #if (defined(__cpp_exceptions) || defined(__EXCEPTIONS) || defined(_CPPUNWIND)) &&
    !defined(JSON_NOEXCEPTION)
02540     #define JSON_THROW(exception) throw exception
02541     #define JSON_TRY try
02542     #define JSON_CATCH(exception) catch(exception)
02543     #define JSON_INTERNAL_CATCH(exception) catch(exception)
02544 #else
02545     #include <cstdlib>
02546     #define JSON_THROW(exception) std::abort()
02547     #define JSON_TRY if(true)
02548     #define JSON_CATCH(exception) if(false)
02549     #define JSON_INTERNAL_CATCH(exception) if(false)
02550 #endif
02551
02552 // override exception macros
02553 #if defined(JSON_THROW_USER)
02554     #undef JSON_THROW
02555     #define JSON_THROW JSON_THROW_USER
02556 #endif
02557 #if defined(JSON_TRY_USER)
02558     #undef JSON_TRY
02559     #define JSON_TRY JSON_TRY_USER
```

```
02560 #endif
02561 #if defined(JSON_CATCH_USER)
02562     #undef JSON_CATCH
02563     #define JSON_CATCH JSON_CATCH_USER
02564     #undef JSON_INTERNAL_CATCH
02565     #define JSON_INTERNAL_CATCH JSON_CATCH_USER
02566 #endif
02567 #if defined(JSON_INTERNAL_CATCH_USER)
02568     #undef JSON_INTERNAL_CATCH
02569     #define JSON_INTERNAL_CATCH JSON_INTERNAL_CATCH_USER
02570 #endif
02571
02572 // allow overriding assert
02573 #if !defined(JSON_ASSERT)
02574     #include <cassert> // assert
02575     #define JSON_ASSERT(x) assert(x)
02576 #endif
02577
02578 // allow accessing some private functions (needed by the test suite)
02579 #if defined(JSON_TESTS_PRIVATE)
02580     #define JSON_PRIVATE_UNLESS_TESTED public
02581 #else
02582     #define JSON_PRIVATE_UNLESS_TESTED private
02583 #endif
02584
02590 #define NLOHMANN_JSON_SERIALIZE_ENUM(ENUM_TYPE, ...)                                    \
02591     template<typename BasicJsonType>                                                    \
02592     inline void to_json(BasicJsonType& j, const ENUM_TYPE& e)                           \
02593     {                                                                                   \
02594         /* NOLINTNEXTLINE(modernize-type-traits) we use C++11 */                        \
02595         static_assert(std::is_enum<ENUM_TYPE>::value, #ENUM_TYPE " must be an enum!");  \
02596         /* NOLINTNEXTLINE(modernize-avoid-c-arrays) we don't want to depend on <array> */ \
02597         static const std::pair<ENUM_TYPE, BasicJsonType> m[] = __VA_ARGS__;             \
02598         auto it = std::find_if(std::begin(m), std::end(m),                              \
02599                                [e](const std::pair<ENUM_TYPE, BasicJsonType>& ej_pair) -> bool \
02600         {                                                                               \
02601             return ej_pair.first == e;                                                  \
02602         });                                                                             \
02603         j = ((it != std::end(m)) ? it : std::begin(m))->second;                         \
02604     }                                                                                   \
02605     template<typename BasicJsonType>                                                    \
02606     inline void from_json(const BasicJsonType& j, ENUM_TYPE& e)                         \
02607     {                                                                                   \
02608         /* NOLINTNEXTLINE(modernize-type-traits) we use C++11 */                        \
02609         static_assert(std::is_enum<ENUM_TYPE>::value, #ENUM_TYPE " must be an enum!");  \
02610         /* NOLINTNEXTLINE(modernize-avoid-c-arrays) we don't want to depend on <array> */ \
02611         static const std::pair<ENUM_TYPE, BasicJsonType> m[] = __VA_ARGS__;             \
02612         auto it = std::find_if(std::begin(m), std::end(m),                              \
02613                                [&j](const std::pair<ENUM_TYPE, BasicJsonType>& ej_pair) -> bool \
02614         {                                                                               \
02615             return ej_pair.second == j;                                                 \
02616         });                                                                             \
02617         e = ((it != std::end(m)) ? it : std::begin(m))->first;                          \
02618     }
02619
02620 // Ugly macros to avoid uglier copy-paste when specializing basic_json. They
02621 // may be removed in the future once the class is split.
02622
02623 #define NLOHMANN_BASIC_JSON_TPL_DECLARATION                                             \
02624     template<template<typename, typename, typename...> class ObjectType,                \
02625              template<typename, typename...> class ArrayType,                           \
02626              class StringType, class BooleanType, class NumberIntegerType,              \
02627              class NumberUnsignedType, class NumberFloatType,                           \
02628              template<typename> class AllocatorType,                                    \
02629              template<typename, typename = void> class JSONSerializer,                  \
02630              class BinaryType,                                                          \
02631              class CustomBaseClass>
02632
02633 #define NLOHMANN_BASIC_JSON_TPL                                                         \
02634     basic_json<ObjectType, ArrayType, StringType, BooleanType,                          \
02635     NumberIntegerType, NumberUnsignedType, NumberFloatType,                             \
02636     AllocatorType, JSONSerializer, BinaryType, CustomBaseClass>
02637
02638 // Macros to simplify conversion from/to types
02639
02640 #define NLOHMANN_JSON_EXPAND( x ) x
02641 #define NLOHMANN_JSON_GET_MACRO(_1, _2, _3, _4, _5, _6, _7, _8, _9, _10, _11, _12, _13, _14, _15, _16,
      _17, _18, _19, _20, _21, _22, _23, _24, _25, _26, _27, _28, _29, _30, _31, _32, _33, _34, _35, _36,
      _37, _38, _39, _40, _41, _42, _43, _44, _45, _46, _47, _48, _49, _50, _51, _52, _53, _54, _55, _56,
      _57, _58, _59, _60, _61, _62, _63, _64, NAME,...) NAME
02642 #define NLOHMANN_JSON_PASTE(...) NLOHMANN_JSON_EXPAND(NLOHMANN_JSON_GET_MACRO(__VA_ARGS__, \
02643         NLOHMANN_JSON_PASTE64, \
02644         NLOHMANN_JSON_PASTE63, \
02645         NLOHMANN_JSON_PASTE62, \
02646         NLOHMANN_JSON_PASTE61, \
02647         NLOHMANN_JSON_PASTE60, \
02648         NLOHMANN_JSON_PASTE59, \
```

```
02649         NLOHMANN_JSON_PASTE58, \
02650         NLOHMANN_JSON_PASTE57, \
02651         NLOHMANN_JSON_PASTE56, \
02652         NLOHMANN_JSON_PASTE55, \
02653         NLOHMANN_JSON_PASTE54, \
02654         NLOHMANN_JSON_PASTE53, \
02655         NLOHMANN_JSON_PASTE52, \
02656         NLOHMANN_JSON_PASTE51, \
02657         NLOHMANN_JSON_PASTE50, \
02658         NLOHMANN_JSON_PASTE49, \
02659         NLOHMANN_JSON_PASTE48, \
02660         NLOHMANN_JSON_PASTE47, \
02661         NLOHMANN_JSON_PASTE46, \
02662         NLOHMANN_JSON_PASTE45, \
02663         NLOHMANN_JSON_PASTE44, \
02664         NLOHMANN_JSON_PASTE43, \
02665         NLOHMANN_JSON_PASTE42, \
02666         NLOHMANN_JSON_PASTE41, \
02667         NLOHMANN_JSON_PASTE40, \
02668         NLOHMANN_JSON_PASTE39, \
02669         NLOHMANN_JSON_PASTE38, \
02670         NLOHMANN_JSON_PASTE37, \
02671         NLOHMANN_JSON_PASTE36, \
02672         NLOHMANN_JSON_PASTE35, \
02673         NLOHMANN_JSON_PASTE34, \
02674         NLOHMANN_JSON_PASTE33, \
02675         NLOHMANN_JSON_PASTE32, \
02676         NLOHMANN_JSON_PASTE31, \
02677         NLOHMANN_JSON_PASTE30, \
02678         NLOHMANN_JSON_PASTE29, \
02679         NLOHMANN_JSON_PASTE28, \
02680         NLOHMANN_JSON_PASTE27, \
02681         NLOHMANN_JSON_PASTE26, \
02682         NLOHMANN_JSON_PASTE25, \
02683         NLOHMANN_JSON_PASTE24, \
02684         NLOHMANN_JSON_PASTE23, \
02685         NLOHMANN_JSON_PASTE22, \
02686         NLOHMANN_JSON_PASTE21, \
02687         NLOHMANN_JSON_PASTE20, \
02688         NLOHMANN_JSON_PASTE19, \
02689         NLOHMANN_JSON_PASTE18, \
02690         NLOHMANN_JSON_PASTE17, \
02691         NLOHMANN_JSON_PASTE16, \
02692         NLOHMANN_JSON_PASTE15, \
02693         NLOHMANN_JSON_PASTE14, \
02694         NLOHMANN_JSON_PASTE13, \
02695         NLOHMANN_JSON_PASTE12, \
02696         NLOHMANN_JSON_PASTE11, \
02697         NLOHMANN_JSON_PASTE10, \
02698         NLOHMANN_JSON_PASTE9, \
02699         NLOHMANN_JSON_PASTE8, \
02700         NLOHMANN_JSON_PASTE7, \
02701         NLOHMANN_JSON_PASTE6, \
02702         NLOHMANN_JSON_PASTE5, \
02703         NLOHMANN_JSON_PASTE4, \
02704         NLOHMANN_JSON_PASTE3, \
02705         NLOHMANN_JSON_PASTE2, \
02706         NLOHMANN_JSON_PASTE1)(__VA_ARGS__))
02707 #define NLOHMANN_JSON_PASTE2(func, v1) func(v1)
02708 #define NLOHMANN_JSON_PASTE3(func, v1, v2) NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE2(func,
      v2)
02709 #define NLOHMANN_JSON_PASTE4(func, v1, v2, v3) NLOHMANN_JSON_PASTE2(func, v1)
      NLOHMANN_JSON_PASTE3(func, v2, v3)
02710 #define NLOHMANN_JSON_PASTE5(func, v1, v2, v3, v4) NLOHMANN_JSON_PASTE2(func, v1)
      NLOHMANN_JSON_PASTE4(func, v2, v3, v4)
02711 #define NLOHMANN_JSON_PASTE6(func, v1, v2, v3, v4, v5) NLOHMANN_JSON_PASTE2(func, v1)
      NLOHMANN_JSON_PASTE5(func, v2, v3, v4, v5)
02712 #define NLOHMANN_JSON_PASTE7(func, v1, v2, v3, v4, v5, v6) NLOHMANN_JSON_PASTE2(func, v1)
      NLOHMANN_JSON_PASTE6(func, v2, v3, v4, v5, v6)
02713 #define NLOHMANN_JSON_PASTE8(func, v1, v2, v3, v4, v5, v6, v7) NLOHMANN_JSON_PASTE2(func, v1)
      NLOHMANN_JSON_PASTE7(func, v2, v3, v4, v5, v6, v7)
02714 #define NLOHMANN_JSON_PASTE9(func, v1, v2, v3, v4, v5, v6, v7, v8) NLOHMANN_JSON_PASTE2(func, v1)
      NLOHMANN_JSON_PASTE8(func, v2, v3, v4, v5, v6, v7, v8)
02715 #define NLOHMANN_JSON_PASTE10(func, v1, v2, v3, v4, v5, v6, v7, v8, v9) NLOHMANN_JSON_PASTE2(func, v1)
      NLOHMANN_JSON_PASTE9(func, v2, v3, v4, v5, v6, v7, v8, v9)
02716 #define NLOHMANN_JSON_PASTE11(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10)
      NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE10(func, v2, v3, v4, v5, v6, v7, v8, v9, v10)
02717 #define NLOHMANN_JSON_PASTE12(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11)
      NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE11(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11)
02718 #define NLOHMANN_JSON_PASTE13(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12)
      NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE12(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11,
      v12)
02719 #define NLOHMANN_JSON_PASTE14(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13)
      NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE13(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11,
      v12, v13)
02720 #define NLOHMANN_JSON_PASTE15(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14)
      NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE14(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11,
```

```
      v12, v13, v14)
02721 #define NLOHMANN_JSON_PASTE16(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15)
      NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE15(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11,
      v12, v13, v14, v15)
02722 #define NLOHMANN_JSON_PASTE17(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
      v16) NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE16(func, v2, v3, v4, v5, v6, v7, v8, v9, v10,
      v11, v12, v13, v14, v15, v16)
02723 #define NLOHMANN_JSON_PASTE18(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
      v16, v17) NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE17(func, v2, v3, v4, v5, v6, v7, v8, v9,
      v10, v11, v12, v13, v14, v15, v16, v17)
02724 #define NLOHMANN_JSON_PASTE19(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
      v16, v17, v18) NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE18(func, v2, v3, v4, v5, v6, v7, v8,
      v9, v10, v11, v12, v13, v14, v15, v16, v17, v18)
02725 #define NLOHMANN_JSON_PASTE20(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
      v16, v17, v18, v19) NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE19(func, v2, v3, v4, v5, v6, v7,
      v8, v9, v10, v11, v12, v13, v14, v15, v16, v17, v18, v19)
02726 #define NLOHMANN_JSON_PASTE21(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
      v16, v17, v18, v19, v20) NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE20(func, v2, v3, v4, v5,
      v6, v7, v8, v9, v10, v11, v12, v13, v14, v15, v16, v17, v18, v19, v20)
02727 #define NLOHMANN_JSON_PASTE22(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
      v16, v17, v18, v19, v20, v21) NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE21(func, v2, v3, v4,
      v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15, v16, v17, v18, v19, v20, v21)
02728 #define NLOHMANN_JSON_PASTE23(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
      v16, v17, v18, v19, v20, v21, v22) NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE22(func, v2, v3,
      v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22)
02729 #define NLOHMANN_JSON_PASTE24(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
      v16, v17, v18, v19, v20, v21, v22, v23) NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE23(func, v2,
      v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22, v23)
02730 #define NLOHMANN_JSON_PASTE25(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
      v16, v17, v18, v19, v20, v21, v22, v23, v24) NLOHMANN_JSON_PASTE2(func, v1)
      NLOHMANN_JSON_PASTE24(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15, v16, v17,
      v18, v19, v20, v21, v22, v23, v24)
02731 #define NLOHMANN_JSON_PASTE26(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
      v16, v17, v18, v19, v20, v21, v22, v23, v24, v25) NLOHMANN_JSON_PASTE2(func, v1)
      NLOHMANN_JSON_PASTE25(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15, v16, v17,
      v18, v19, v20, v21, v22, v23, v24, v25)
02732 #define NLOHMANN_JSON_PASTE27(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
      v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26) NLOHMANN_JSON_PASTE2(func, v1)
      NLOHMANN_JSON_PASTE26(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15, v16, v17,
      v18, v19, v20, v21, v22, v23, v24, v25, v26)
02733 #define NLOHMANN_JSON_PASTE28(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
      v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27) NLOHMANN_JSON_PASTE2(func, v1)
      NLOHMANN_JSON_PASTE27(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15, v16, v17,
      v18, v19, v20, v21, v22, v23, v24, v25, v26, v27)
02734 #define NLOHMANN_JSON_PASTE29(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
      v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28) NLOHMANN_JSON_PASTE2(func, v1)
      NLOHMANN_JSON_PASTE28(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15, v16, v17,
      v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28)
02735 #define NLOHMANN_JSON_PASTE30(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
      v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29) NLOHMANN_JSON_PASTE2(func, v1)
      NLOHMANN_JSON_PASTE29(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15, v16, v17,
      v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29)
02736 #define NLOHMANN_JSON_PASTE31(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
      v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30) NLOHMANN_JSON_PASTE2(func,
      v1) NLOHMANN_JSON_PASTE30(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15, v16,
      v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30)
02737 #define NLOHMANN_JSON_PASTE32(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
      v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31)
      NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE31(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11,
      v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31)
02738 #define NLOHMANN_JSON_PASTE33(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
      v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32)
      NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE32(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11,
      v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31,
      v32)
02739 #define NLOHMANN_JSON_PASTE34(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
      v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33)
      NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE33(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11,
      v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31,
      v32, v33)
02740 #define NLOHMANN_JSON_PASTE35(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
      v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34)
      NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE34(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11,
      v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31,
      v32, v33, v34)
02741 #define NLOHMANN_JSON_PASTE36(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
      v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35)
      NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE35(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11,
      v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31,
      v32, v33, v34, v35)
02742 #define NLOHMANN_JSON_PASTE37(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
      v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35,
      v36) NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE36(func, v2, v3, v4, v5, v6, v7, v8, v9, v10,
      v11, v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30,
      v31, v32, v33, v34, v35, v36)
02743 #define NLOHMANN_JSON_PASTE38(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
      v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35,
```

```
    v36, v37) NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE37(func, v2, v3, v4, v5, v6, v7, v8, v9,
    v10, v11, v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29,
    v30, v31, v32, v33, v34, v35, v36, v37)
02744 #define NLOHMANN_JSON_PASTE39(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
    v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35,
    v36, v37, v38) NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE38(func, v2, v3, v4, v5, v6, v7, v8,
    v9, v10, v11, v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28,
    v29, v30, v31, v32, v33, v34, v35, v36, v37, v38)
02745 #define NLOHMANN_JSON_PASTE40(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
    v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35,
    v36, v37, v38, v39) NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE39(func, v2, v3, v4, v5, v6, v7,
    v8, v9, v10, v11, v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28,
    v29, v30, v31, v32, v33, v34, v35, v36, v37, v38, v39)
02746 #define NLOHMANN_JSON_PASTE41(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
    v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35,
    v36, v37, v38, v39, v40) NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE40(func, v2, v3, v4, v5,
    v6, v7, v8, v9, v10, v11, v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26,
    v27, v28, v29, v30, v31, v32, v33, v34, v35, v36, v37, v38, v39, v40)
02747 #define NLOHMANN_JSON_PASTE42(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
    v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35,
    v36, v37, v38, v39, v40, v41) NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE41(func, v2, v3, v4,
    v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22, v23, v24, v25,
    v26, v27, v28, v29, v30, v31, v32, v33, v34, v35, v36, v37, v38, v39, v40, v41)
02748 #define NLOHMANN_JSON_PASTE43(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
    v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35,
    v36, v37, v38, v39, v40, v41, v42) NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE42(func, v2, v3,
    v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22, v23, v24,
    v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35, v36, v37, v38, v39, v40, v41, v42)
02749 #define NLOHMANN_JSON_PASTE44(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
    v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35,
    v36, v37, v38, v39, v40, v41, v42, v43) NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE43(func, v2,
    v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22, v23, v24,
    v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35, v36, v37, v38, v39, v40, v41, v42, v43)
02750 #define NLOHMANN_JSON_PASTE45(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
    v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35,
    v36, v37, v38, v39, v40, v41, v42, v43, v44) NLOHMANN_JSON_PASTE2(func, v1)
    NLOHMANN_JSON_PASTE44(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15, v16, v17,
    v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35, v36, v37,
    v38, v39, v40, v41, v42, v43, v44)
02751 #define NLOHMANN_JSON_PASTE46(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
    v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35,
    v36, v37, v38, v39, v40, v41, v42, v43, v44, v45) NLOHMANN_JSON_PASTE2(func, v1)
    NLOHMANN_JSON_PASTE45(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15, v16, v17,
    v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35, v36, v37,
    v38, v39, v40, v41, v42, v43, v44, v45)
02752 #define NLOHMANN_JSON_PASTE47(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
    v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35,
    v36, v37, v38, v39, v40, v41, v42, v43, v44, v45, v46) NLOHMANN_JSON_PASTE2(func, v1)
    NLOHMANN_JSON_PASTE46(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15, v16, v17,
    v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35, v36, v37,
    v38, v39, v40, v41, v42, v43, v44, v45, v46)
02753 #define NLOHMANN_JSON_PASTE48(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
    v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35,
    v36, v37, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47) NLOHMANN_JSON_PASTE2(func, v1)
    NLOHMANN_JSON_PASTE47(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15, v16, v17,
    v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35, v36, v37,
    v38, v39, v40, v41, v42, v43, v44, v45, v46, v47)
02754 #define NLOHMANN_JSON_PASTE49(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
    v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35,
    v36, v37, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47, v48) NLOHMANN_JSON_PASTE2(func, v1)
    NLOHMANN_JSON_PASTE48(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15, v16, v17,
    v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35, v36, v37,
    v38, v39, v40, v41, v42, v43, v44, v45, v46, v47, v48)
02755 #define NLOHMANN_JSON_PASTE50(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
    v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35,
    v36, v37, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47, v48, v49) NLOHMANN_JSON_PASTE2(func, v1)
    NLOHMANN_JSON_PASTE49(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15, v16, v17,
    v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35, v36, v37,
    v38, v39, v40, v41, v42, v43, v44, v45, v46, v47, v48, v49)
02756 #define NLOHMANN_JSON_PASTE51(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
    v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35,
    v36, v37, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47, v48, v49, v50) NLOHMANN_JSON_PASTE2(func,
    v1) NLOHMANN_JSON_PASTE50(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15, v16,
    v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35, v36,
    v37, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47, v48, v49, v50)
02757 #define NLOHMANN_JSON_PASTE52(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
    v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35,
    v36, v37, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47, v48, v49, v50, v51)
    NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE51(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11,
    v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31,
    v32, v33, v34, v35, v36, v37, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47, v48, v49, v50, v51)
02758 #define NLOHMANN_JSON_PASTE53(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
    v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35,
    v36, v37, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47, v48, v49, v50, v51, v52)
    NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE52(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11,
    v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31,
    v32, v33, v34, v35, v36, v37, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47, v48, v49, v50, v51,
```

```
    v52)
02759 #define NLOHMANN_JSON_PASTE54(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
    v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35,
    v36, v37, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47, v48, v49, v50, v51, v52, v53)
    NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE53(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11,
    v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31,
    v32, v33, v34, v35, v36, v37, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47, v48, v49, v50, v51,
    v52, v53)
02760 #define NLOHMANN_JSON_PASTE55(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
    v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35,
    v36, v37, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47, v48, v49, v50, v51, v52, v53, v54)
    NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE54(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11,
    v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31,
    v32, v33, v34, v35, v36, v37, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47, v48, v49, v50, v51,
    v52, v53, v54)
02761 #define NLOHMANN_JSON_PASTE56(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
    v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35,
    v36, v37, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47, v48, v49, v50, v51, v52, v53, v54, v55)
    NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE55(func, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11,
    v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31,
    v32, v33, v34, v35, v36, v37, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47, v48, v49, v50, v51,
    v52, v53, v54, v55)
02762 #define NLOHMANN_JSON_PASTE57(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
    v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35,
    v36, v37, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47, v48, v49, v50, v51, v52, v53, v54, v55,
    v56) NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE56(func, v2, v3, v4, v5, v6, v7, v8, v9, v10,
    v11, v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30,
    v31, v32, v33, v34, v35, v36, v37, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47, v48, v49, v50,
    v51, v52, v53, v54, v55, v56)
02763 #define NLOHMANN_JSON_PASTE58(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
    v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35,
    v36, v37, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47, v48, v49, v50, v51, v52, v53, v54, v55,
    v56, v57) NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE57(func, v2, v3, v4, v5, v6, v7, v8, v9,
    v10, v11, v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29,
    v30, v31, v32, v33, v34, v35, v36, v37, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47, v48, v49,
    v50, v51, v52, v53, v54, v55, v56, v57)
02764 #define NLOHMANN_JSON_PASTE59(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
    v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35,
    v36, v37, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47, v48, v49, v50, v51, v52, v53, v54, v55,
    v56, v57, v58) NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE58(func, v2, v3, v4, v5, v6, v7, v8,
    v9, v10, v11, v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28,
    v29, v30, v31, v32, v33, v34, v35, v36, v37, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47, v48,
    v49, v50, v51, v52, v53, v54, v55, v56, v57, v58)
02765 #define NLOHMANN_JSON_PASTE60(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
    v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35,
    v36, v37, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47, v48, v49, v50, v51, v52, v53, v54, v55,
    v56, v57, v58, v59) NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE59(func, v2, v3, v4, v5, v6, v7,
    v8, v9, v10, v11, v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28,
    v29, v30, v31, v32, v33, v34, v35, v36, v37, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47, v48,
    v49, v50, v51, v52, v53, v54, v55, v56, v57, v58, v59)
02766 #define NLOHMANN_JSON_PASTE61(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
    v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35,
    v36, v37, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47, v48, v49, v50, v51, v52, v53, v54, v55,
    v56, v57, v58, v59, v60) NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE60(func, v2, v3, v4, v5,
    v6, v7, v8, v9, v10, v11, v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26,
    v27, v28, v29, v30, v31, v32, v33, v34, v35, v36, v37, v38, v39, v40, v41, v42, v43, v44, v45, v46,
    v47, v48, v49, v50, v51, v52, v53, v54, v55, v56, v57, v58, v59, v60)
02767 #define NLOHMANN_JSON_PASTE62(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
    v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35,
    v36, v37, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47, v48, v49, v50, v51, v52, v53, v54, v55,
    v56, v57, v58, v59, v60, v61) NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE61(func, v2, v3, v4,
    v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22, v23, v24, v25,
    v26, v27, v28, v29, v30, v31, v32, v33, v34, v35, v36, v37, v38, v39, v40, v41, v42, v43, v44, v45,
    v46, v47, v48, v49, v50, v51, v52, v53, v54, v55, v56, v57, v58, v59, v60, v61)
02768 #define NLOHMANN_JSON_PASTE63(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
    v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35,
    v36, v37, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47, v48, v49, v50, v51, v52, v53, v54, v55,
    v56, v57, v58, v59, v60, v61, v62) NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE62(func, v2, v3,
    v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22, v23, v24,
    v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35, v36, v37, v38, v39, v40, v41, v42, v43, v44,
    v45, v46, v47, v48, v49, v50, v51, v52, v53, v54, v55, v56, v57, v58, v59, v60, v61, v62)
02769 #define NLOHMANN_JSON_PASTE64(func, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
    v16, v17, v18, v19, v20, v21, v22, v23, v24, v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35,
    v36, v37, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47, v48, v49, v50, v51, v52, v53, v54, v55,
    v56, v57, v58, v59, v60, v61, v62, v63) NLOHMANN_JSON_PASTE2(func, v1) NLOHMANN_JSON_PASTE63(func, v2,
    v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15, v16, v17, v18, v19, v20, v21, v22, v23, v24,
    v25, v26, v27, v28, v29, v30, v31, v32, v33, v34, v35, v36, v37, v38, v39, v40, v41, v42, v43, v44,
    v45, v46, v47, v48, v49, v50, v51, v52, v53, v54, v55, v56, v57, v58, v59, v60, v61, v62, v63)
02770
02771 #define NLOHMANN_JSON_TO(v1) nlohmann_json_j[#v1] = nlohmann_json_t.v1;
02772 #define NLOHMANN_JSON_FROM(v1) nlohmann_json_j.at(#v1).get_to(nlohmann_json_t.v1);
02773 #define NLOHMANN_JSON_FROM_WITH_DEFAULT(v1) nlohmann_json_t.v1 = !nlohmann_json_j.is_null() ?
    nlohmann_json_j.value(#v1, nlohmann_json_default_obj.v1) : nlohmann_json_default_obj.v1;
02774
02781 #define NLOHMANN_DEFINE_TYPE_INTRUSIVE(Type, ...)  \
02782     template<typename BasicJsonType,
    nlohmann::detail::enable_if_t<nlohmann::detail::is_basic_json<BasicJsonType>::value, int> = 0> \
```

```
02783     friend void to_json(BasicJsonType& nlohmann_json_j, const Type& nlohmann_json_t) {
      NLOHMANN_JSON_EXPAND(NLOHMANN_JSON_PASTE(NLOHMANN_JSON_TO, __VA_ARGS__)) } \
02784     template<typename BasicJsonType,
      nlohmann::detail::enable_if_t<nlohmann::detail::is_basic_json<BasicJsonType>::value, int> = 0> \
02785     friend void from_json(const BasicJsonType& nlohmann_json_j, Type& nlohmann_json_t) {
      NLOHMANN_JSON_EXPAND(NLOHMANN_JSON_PASTE(NLOHMANN_JSON_FROM, __VA_ARGS__)) }
02786
02793 #define NLOHMANN_DEFINE_TYPE_INTRUSIVE_WITH_DEFAULT(Type, ...)  \
02794     template<typename BasicJsonType,
      nlohmann::detail::enable_if_t<nlohmann::detail::is_basic_json<BasicJsonType>::value, int> = 0> \
02795     friend void to_json(BasicJsonType& nlohmann_json_j, const Type& nlohmann_json_t) {
      NLOHMANN_JSON_EXPAND(NLOHMANN_JSON_PASTE(NLOHMANN_JSON_TO, __VA_ARGS__)) } \
02796     template<typename BasicJsonType,
      nlohmann::detail::enable_if_t<nlohmann::detail::is_basic_json<BasicJsonType>::value, int> = 0> \
02797     friend void from_json(const BasicJsonType& nlohmann_json_j, Type& nlohmann_json_t) { const Type
      nlohmann_json_default_obj{}; NLOHMANN_JSON_EXPAND(NLOHMANN_JSON_PASTE(NLOHMANN_JSON_FROM_WITH_DEFAULT,
      __VA_ARGS__)) }
02798
02805 #define NLOHMANN_DEFINE_TYPE_INTRUSIVE_ONLY_SERIALIZE(Type, ...)  \
02806     template<typename BasicJsonType,
      nlohmann::detail::enable_if_t<nlohmann::detail::is_basic_json<BasicJsonType>::value, int> = 0> \
02807     friend void to_json(BasicJsonType& nlohmann_json_j, const Type& nlohmann_json_t) {
      NLOHMANN_JSON_EXPAND(NLOHMANN_JSON_PASTE(NLOHMANN_JSON_TO, __VA_ARGS__)) }
02808
02815 #define NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE(Type, ...)  \
02816     template<typename BasicJsonType,
      nlohmann::detail::enable_if_t<nlohmann::detail::is_basic_json<BasicJsonType>::value, int> = 0> \
02817     void to_json(BasicJsonType& nlohmann_json_j, const Type& nlohmann_json_t) {
      NLOHMANN_JSON_EXPAND(NLOHMANN_JSON_PASTE(NLOHMANN_JSON_TO, __VA_ARGS__)) } \
02818     template<typename BasicJsonType,
      nlohmann::detail::enable_if_t<nlohmann::detail::is_basic_json<BasicJsonType>::value, int> = 0> \
02819     void from_json(const BasicJsonType& nlohmann_json_j, Type& nlohmann_json_t) {
      NLOHMANN_JSON_EXPAND(NLOHMANN_JSON_PASTE(NLOHMANN_JSON_FROM, __VA_ARGS__)) }
02820
02827 #define NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE_WITH_DEFAULT(Type, ...)  \
02828     template<typename BasicJsonType,
      nlohmann::detail::enable_if_t<nlohmann::detail::is_basic_json<BasicJsonType>::value, int> = 0> \
02829     void to_json(BasicJsonType& nlohmann_json_j, const Type& nlohmann_json_t) {
      NLOHMANN_JSON_EXPAND(NLOHMANN_JSON_PASTE(NLOHMANN_JSON_TO, __VA_ARGS__)) } \
02830     template<typename BasicJsonType,
      nlohmann::detail::enable_if_t<nlohmann::detail::is_basic_json<BasicJsonType>::value, int> = 0> \
02831     void from_json(const BasicJsonType& nlohmann_json_j, Type& nlohmann_json_t) { const Type
      nlohmann_json_default_obj{}; NLOHMANN_JSON_EXPAND(NLOHMANN_JSON_PASTE(NLOHMANN_JSON_FROM_WITH_DEFAULT,
      __VA_ARGS__)) }
02832
02839 #define NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE_ONLY_SERIALIZE(Type, ...)  \
02840     template<typename BasicJsonType,
      nlohmann::detail::enable_if_t<nlohmann::detail::is_basic_json<BasicJsonType>::value, int> = 0> \
02841     void to_json(BasicJsonType& nlohmann_json_j, const Type& nlohmann_json_t) {
      NLOHMANN_JSON_EXPAND(NLOHMANN_JSON_PASTE(NLOHMANN_JSON_TO, __VA_ARGS__)) }
02842
02849 #define NLOHMANN_DEFINE_DERIVED_TYPE_INTRUSIVE(Type, BaseType, ...)  \
02850     template<typename BasicJsonType,
      nlohmann::detail::enable_if_t<nlohmann::detail::is_basic_json<BasicJsonType>::value, int> = 0> \
02851     friend void to_json(BasicJsonType& nlohmann_json_j, const Type& nlohmann_json_t) {
      nlohmann::to_json(nlohmann_json_j, static_cast<const BaseType &>(nlohmann_json_t));
      NLOHMANN_JSON_EXPAND(NLOHMANN_JSON_PASTE(NLOHMANN_JSON_TO, __VA_ARGS__)) } \
02852     template<typename BasicJsonType,
      nlohmann::detail::enable_if_t<nlohmann::detail::is_basic_json<BasicJsonType>::value, int> = 0> \
02853     friend void from_json(const BasicJsonType& nlohmann_json_j, Type& nlohmann_json_t) {
      nlohmann::from_json(nlohmann_json_j, static_cast<BaseType&>(nlohmann_json_t));
      NLOHMANN_JSON_EXPAND(NLOHMANN_JSON_PASTE(NLOHMANN_JSON_FROM, __VA_ARGS__)) }
02854
02861 #define NLOHMANN_DEFINE_DERIVED_TYPE_INTRUSIVE_WITH_DEFAULT(Type, BaseType, ...)  \
02862     template<typename BasicJsonType,
      nlohmann::detail::enable_if_t<nlohmann::detail::is_basic_json<BasicJsonType>::value, int> = 0> \
02863     friend void to_json(BasicJsonType& nlohmann_json_j, const Type& nlohmann_json_t) {
      nlohmann::to_json(nlohmann_json_j, static_cast<const BaseType&>(nlohmann_json_t));
      NLOHMANN_JSON_EXPAND(NLOHMANN_JSON_PASTE(NLOHMANN_JSON_TO, __VA_ARGS__)) } \
02864     template<typename BasicJsonType,
      nlohmann::detail::enable_if_t<nlohmann::detail::is_basic_json<BasicJsonType>::value, int> = 0> \
02865     friend void from_json(const BasicJsonType& nlohmann_json_j, Type& nlohmann_json_t) {
      nlohmann::from_json(nlohmann_json_j, static_cast<BaseType&>(nlohmann_json_t)); const Type
      nlohmann_json_default_obj{}; NLOHMANN_JSON_EXPAND(NLOHMANN_JSON_PASTE(NLOHMANN_JSON_FROM_WITH_DEFAULT,
      __VA_ARGS__)) }
02866
02873 #define NLOHMANN_DEFINE_DERIVED_TYPE_INTRUSIVE_ONLY_SERIALIZE(Type, BaseType, ...)  \
02874     template<typename BasicJsonType,
      nlohmann::detail::enable_if_t<nlohmann::detail::is_basic_json<BasicJsonType>::value, int> = 0> \
02875     friend void to_json(BasicJsonType& nlohmann_json_j, const Type& nlohmann_json_t) {
      nlohmann::to_json(nlohmann_json_j, static_cast<const BaseType &>(nlohmann_json_t));
      NLOHMANN_JSON_EXPAND(NLOHMANN_JSON_PASTE(NLOHMANN_JSON_TO, __VA_ARGS__)) }
02876
02883 #define NLOHMANN_DEFINE_DERIVED_TYPE_NON_INTRUSIVE(Type, BaseType, ...)  \
02884     template<typename BasicJsonType,
      nlohmann::detail::enable_if_t<nlohmann::detail::is_basic_json<BasicJsonType>::value, int> = 0> \
02885     void to_json(BasicJsonType& nlohmann_json_j, const Type& nlohmann_json_t) {
```

```
    nlohmann::to_json(nlohmann_json_j, static_cast<const BaseType &>(nlohmann_json_t));
    NLOHMANN_JSON_EXPAND(NLOHMANN_JSON_PASTE(NLOHMANN_JSON_TO, __VA_ARGS__)) } \
02886     template<typename BasicJsonType,
    nlohmann::detail::enable_if_t<nlohmann::detail::is_basic_json<BasicJsonType>::value, int> = 0> \
02887     void from_json(const BasicJsonType& nlohmann_json_j, Type& nlohmann_json_t) {
    nlohmann::from_json(nlohmann_json_j, static_cast<BaseType&>(nlohmann_json_t));
    NLOHMANN_JSON_EXPAND(NLOHMANN_JSON_PASTE(NLOHMANN_JSON_FROM, __VA_ARGS__)) }
02888
02895 #define NLOHMANN_DEFINE_DERIVED_TYPE_NON_INTRUSIVE_WITH_DEFAULT(Type, BaseType, ...)  \
02896     template<typename BasicJsonType,
    nlohmann::detail::enable_if_t<nlohmann::detail::is_basic_json<BasicJsonType>::value, int> = 0> \
02897     void to_json(BasicJsonType& nlohmann_json_j, const Type& nlohmann_json_t) {
    nlohmann::to_json(nlohmann_json_j, static_cast<const BaseType &>(nlohmann_json_t));
    NLOHMANN_JSON_EXPAND(NLOHMANN_JSON_PASTE(NLOHMANN_JSON_TO, __VA_ARGS__)) } \
02898     template<typename BasicJsonType,
    nlohmann::detail::enable_if_t<nlohmann::detail::is_basic_json<BasicJsonType>::value, int> = 0> \
02899     void from_json(const BasicJsonType& nlohmann_json_j, Type& nlohmann_json_t) {
    nlohmann::from_json(nlohmann_json_j, static_cast<BaseType&>(nlohmann_json_t)); const Type
    nlohmann_json_default_obj{}; NLOHMANN_JSON_EXPAND(NLOHMANN_JSON_PASTE(NLOHMANN_JSON_FROM_WITH_DEFAULT,
    __VA_ARGS__)) }
02900
02907 #define NLOHMANN_DEFINE_DERIVED_TYPE_NON_INTRUSIVE_ONLY_SERIALIZE(Type, BaseType, ...)  \
02908     template<typename BasicJsonType,
    nlohmann::detail::enable_if_t<nlohmann::detail::is_basic_json<BasicJsonType>::value, int> = 0> \
02909     void to_json(BasicJsonType& nlohmann_json_j, const Type& nlohmann_json_t) {
    nlohmann::to_json(nlohmann_json_j, static_cast<const BaseType &>(nlohmann_json_t));
    NLOHMANN_JSON_EXPAND(NLOHMANN_JSON_PASTE(NLOHMANN_JSON_TO, __VA_ARGS__)) }
02910
02911 // inspired from https://stackoverflow.com/a/26745591
02912 // allows calling any std function as if (e.g., with begin):
02913 // using std::begin; begin(x);
02914 //
02915 // it allows using the detected idiom to retrieve the return type
02916 // of such an expression
02917 #define NLOHMANN_CAN_CALL_STD_FUNC_IMPL(std_name)                                    \
02918     namespace detail {                                                               \
02919     using std::std_name;                                                             \
02920     \
02921     template<typename... T>                                                          \
02922     using result_of_##std_name = decltype(std_name(std::declval<T>()...));          \
02923     }                                                                                \
02924     \
02925     namespace detail2 {                                                              \
02926     struct std_name##_tag                                                            \
02927     {                                                                                \
02928     };                                                                               \
02929     \
02930     template<typename... T>                                                          \
02931     std_name##_tag std_name(T&&...);                                                 \
02932     \
02933     template<typename... T>                                                          \
02934     using result_of_##std_name = decltype(std_name(std::declval<T>()...));          \
02935     \
02936     template<typename... T>                                                          \
02937     struct would_call_std_##std_name                                                 \
02938     {                                                                                \
02939         static constexpr auto const value = ::nlohmann::detail::                    \
02940                                             is_detected_exact<std_name##_tag, result_of_##std_name,
    T...>::value; \
02941     };                                                                               \
02942     } /* namespace detail2 */ \
02943     \
02944     template<typename... T>                                                          \
02945     struct would_call_std_##std_name : detail2::would_call_std_##std_name<T...>      \
02946     {                                                                                \
02947     }
02948
02949 #ifndef JSON_USE_IMPLICIT_CONVERSIONS
02950     #define JSON_USE_IMPLICIT_CONVERSIONS 1
02951 #endif
02952
02953 #if JSON_USE_IMPLICIT_CONVERSIONS
02954     #define JSON_EXPLICIT
02955 #else
02956     #define JSON_EXPLICIT explicit
02957 #endif
02958
02959 #ifndef JSON_DISABLE_ENUM_SERIALIZATION
02960     #define JSON_DISABLE_ENUM_SERIALIZATION 0
02961 #endif
02962
02963 #ifndef JSON_USE_GLOBAL_UDLS
02964     #define JSON_USE_GLOBAL_UDLS 1
02965 #endif
02966
02967 #if JSON_HAS_THREE_WAY_COMPARISON
02968     #include <compare> // partial_ordering
```

```
02969 #endif
02970
02971 NLOHMANN_JSON_NAMESPACE_BEGIN
02972 namespace detail
02973 {
02974
02976 // JSON type enumeration //
02978
03003 enum class value_t : std::uint8_t
03004 {
03005     null,
03006     object,
03007     array,
03008     string,
03009     boolean,
03010     number_integer,
03011     number_unsigned,
03012     number_float,
03013     binary,
03014     discarded
03015 };
03016
03030 #if JSON_HAS_THREE_WAY_COMPARISON
03031     inline std::partial_ordering operator<=>(const value_t lhs, const value_t rhs) noexcept // *NOPAD*
03032 #else
03033     inline bool operator<(const value_t lhs, const value_t rhs) noexcept
03034 #endif
03035 {
03036     static constexpr std::array<std::uint8_t, 9> order = {{
03037             0 /* null */, 3 /* object */, 4 /* array */, 5 /* string */,
03038             1 /* boolean */, 2 /* integer */, 2 /* unsigned */, 2 /* float */,
03039             6 /* binary */
03040         }
03041     };
03042
03043     const auto l_index = static_cast<std::size_t>(lhs);
03044     const auto r_index = static_cast<std::size_t>(rhs);
03045 #if JSON_HAS_THREE_WAY_COMPARISON
03046     if (l_index < order.size() && r_index < order.size())
03047     {
03048         return order[l_index] <=> order[r_index]; // *NOPAD*
03049     }
03050     return std::partial_ordering::unordered;
03051 #else
03052     return l_index < order.size() && r_index < order.size() && order[l_index] < order[r_index];
03053 #endif
03054 }
03055
03056 // GCC selects the built-in operator< over an operator rewritten from
03057 // a user-defined spaceship operator
03058 // Clang, MSVC, and ICC select the rewritten candidate
03059 // (see GCC bug https://gcc.gnu.org/bugzilla/show_bug.cgi?id=105200)
03060 #if JSON_HAS_THREE_WAY_COMPARISON && defined(__GNUC__)
03061 inline bool operator<(const value_t lhs, const value_t rhs) noexcept
03062 {
03063     return std::is_lt(lhs <=> rhs); // *NOPAD*
03064 }
03065 #endif
03066
03067 }  // namespace detail
03068 NLOHMANN_JSON_NAMESPACE_END
03069
03070 // #include <nlohmann/detail/string_escape.hpp>
03071 //     __ _____ _____ _____
03072 //  __|  |   __|     |   | |  JSON for Modern C++
03073 // |  |  |__   |  |  | | | |  version 3.12.0
03074 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
03075 //
03076 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
03077 // SPDX-License-Identifier: MIT
03078
03079
03080
03081 // #include <nlohmann/detail/abi_macros.hpp>
03082
03083
03084 NLOHMANN_JSON_NAMESPACE_BEGIN
03085 namespace detail
03086 {
03087
03101 template<typename StringType>
03102 inline void replace_substring(StringType& s, const StringType& f,
03103                               const StringType& t)
03104 {
03105     JSON_ASSERT(!f.empty());
03106     for (auto pos = s.find(f);                // find the first occurrence of f
03107             pos != StringType::npos;          // make sure f was found
```

```
03108             s.replace(pos, f.size(), t),      // replace with t, and
03109             pos = s.find(f, pos + t.size()))  // find the next occurrence of f
03110     {}
03111 }
03112
03120 template<typename StringType>
03121 inline StringType escape(StringType s)
03122 {
03123     replace_substring(s, StringType{"~"}, StringType{"~0"});
03124     replace_substring(s, StringType{"/"}, StringType{"~1"});
03125     return s;
03126 }
03127
03135 template<typename StringType>
03136 inline void unescape(StringType& s)
03137 {
03138     replace_substring(s, StringType{"~1"}, StringType{"/"});
03139     replace_substring(s, StringType{"~0"}, StringType{"~"});
03140 }
03141
03142 }  // namespace detail
03143 NLOHMANN_JSON_NAMESPACE_END
03144
03145 // #include <nlohmann/detail/input/position_t.hpp>
03146 //     __ _____ _____ _____
03147 //  __|  |   __|     |   | |  JSON for Modern C++
03148 // |  |  |__   |  |  | | | |  version 3.12.0
03149 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
03150 //
03151 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
03152 // SPDX-License-Identifier: MIT
03153
03154
03155
03156 #include <cstddef> // size_t
03157
03158 // #include <nlohmann/detail/abi_macros.hpp>
03159
03160
03161 NLOHMANN_JSON_NAMESPACE_BEGIN
03162 namespace detail
03163 {
03164
03166 struct position_t
03167 {
03169     std::size_t chars_read_total = 0;
03171     std::size_t chars_read_current_line = 0;
03173     std::size_t lines_read = 0;
03174
03176     constexpr operator size_t() const
03177     {
03178         return chars_read_total;
03179     }
03180 };
03181
03182 }  // namespace detail
03183 NLOHMANN_JSON_NAMESPACE_END
03184
03185 // #include <nlohmann/detail/macro_scope.hpp>
03186
03187 // #include <nlohmann/detail/meta/cpp_future.hpp>
03188 //     __ _____ _____ _____
03189 //  __|  |   __|     |   | |  JSON for Modern C++
03190 // |  |  |__   |  |  | | | |  version 3.12.0
03191 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
03192 //
03193 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
03194 // SPDX-FileCopyrightText: 2018 The Abseil Authors
03195 // SPDX-License-Identifier: MIT
03196
03197
03198
03199 #include <array> // array
03200 #include <cstddef> // size_t
03201 #include <type_traits> // conditional, enable_if, false_type, integral_constant, is_constructible,
      is_integral, is_same, remove_cv, remove_reference, true_type
03202 #include <utility> // index_sequence, make_index_sequence, index_sequence_for
03203
03204 // #include <nlohmann/detail/macro_scope.hpp>
03205
03206
03207 NLOHMANN_JSON_NAMESPACE_BEGIN
03208 namespace detail
03209 {
03210
03211 template<typename T>
03212 using uncvref_t = typename std::remove_cv<typename std::remove_reference<T>::type>::type;
```

```
03213
03214 #ifdef JSON_HAS_CPP_14
03215
03216 // the following utilities are natively available in C++14
03217 using std::enable_if_t;
03218 using std::index_sequence;
03219 using std::make_index_sequence;
03220 using std::index_sequence_for;
03221
03222 #else
03223
03224 // alias templates to reduce boilerplate
03225 template<bool B, typename T = void>
03226 using enable_if_t = typename std::enable_if<B, T>::type;
03227
03228 // The following code is taken from
      https://github.com/abseil/abseil-cpp/blob/10cb35e459f5ecca5b2ff107635da0bfa41011b4/absl/utility/utility.h
03229 // which is part of Google Abseil (https://github.com/abseil/abseil-cpp), licensed under the Apache
      License 2.0.
03230
03232
03233 // integer_sequence
03234 //
03235 // Class template representing a compile-time integer sequence. An instantiation
03236 // of `integer_sequence<T, Ints...>` has a sequence of integers encoded in its
03237 // type through its template arguments (which is a common need when
03238 // working with C++11 variadic templates). `absl::integer_sequence` is designed
03239 // to be a drop-in replacement for C++14's `std::integer_sequence`.
03240 //
03241 // Example:
03242 //
03243 //   template< class T, T... Ints >
03244 //   void user_function(integer_sequence<T, Ints...>);
03245 //
03246 //   int main()
03247 //   {
03248 //     // user_function's `T` will be deduced to `int` and `Ints...`
03249 //     // will be deduced to `0, 1, 2, 3, 4`.
03250 //     user_function(make_integer_sequence<int, 5>());
03251 //   }
03252 template <typename T, T... Ints>
03253 struct integer_sequence
03254 {
03255     using value_type = T;
03256     static constexpr std::size_t size() noexcept
03257     {
03258         return sizeof...(Ints);
03259     }
03260 };
03261
03262 // index_sequence
03263 //
03264 // A helper template for an `integer_sequence` of `size_t`,
03265 // `absl::index_sequence` is designed to be a drop-in replacement for C++14's
03266 // `std::index_sequence`.
03267 template <size_t... Ints>
03268 using index_sequence = integer_sequence<size_t, Ints...>;
03269
03270 namespace utility_internal
03271 {
03272
03273 template <typename Seq, size_t SeqSize, size_t Rem>
03274 struct Extend;
03275
03276 // Note that SeqSize == sizeof...(Ints). It's passed explicitly for efficiency.
03277 template <typename T, T... Ints, size_t SeqSize>
03278 struct Extend<integer_sequence<T, Ints...>, SeqSize, 0>
03279 {
03280     using type = integer_sequence < T, Ints..., (Ints + SeqSize)... >;
03281 };
03282
03283 template <typename T, T... Ints, size_t SeqSize>
03284 struct Extend<integer_sequence<T, Ints...>, SeqSize, 1>
03285 {
03286     using type = integer_sequence < T, Ints..., (Ints + SeqSize)..., 2 * SeqSize >;
03287 };
03288
03289 // Recursion helper for 'make_integer_sequence<T, N>'.
03290 // 'Gen<T, N>::type' is an alias for 'integer_sequence<T, 0, 1, ... N-1>'.
03291 template <typename T, size_t N>
03292 struct Gen
03293 {
03294     using type =
03295         typename Extend < typename Gen < T, N / 2 >::type, N / 2, N % 2 >::type;
03296 };
03297
03298 template <typename T>
```

```
03299 struct Gen<T, 0>
03300 {
03301     using type = integer_sequence<T>;
03302 };
03303
03304 }  // namespace utility_internal
03305
03306 // Compile-time sequences of integers
03307
03308 // make_integer_sequence
03309 //
03310 // This template alias is equivalent to
03311 // `integer_sequence<int, 0, 1, ..., N-1>`, and is designed to be a drop-in
03312 // replacement for C++14's `std::make_integer_sequence`.
03313 template <typename T, T N>
03314 using make_integer_sequence = typename utility_internal::Gen<T, N>::type;
03315
03316 // make_index_sequence
03317 //
03318 // This template alias is equivalent to `index_sequence<0, 1, ..., N-1>`,
03319 // and is designed to be a drop-in replacement for C++14's
03320 // `std::make_index_sequence`.
03321 template <size_t N>
03322 using make_index_sequence = make_integer_sequence<size_t, N>;
03323
03324 // index_sequence_for
03325 //
03326 // Converts a typename pack into an index sequence of the same length, and
03327 // is designed to be a drop-in replacement for C++14's
03328 // `std::index_sequence_for()`
03329 template <typename... Ts>
03330 using index_sequence_for = make_index_sequence<sizeof...(Ts)>;
03331
03333
03334 #endif
03335
03336 // dispatch utility (taken from ranges-v3)
03337 template<unsigned N> struct priority_tag : priority_tag < N - 1 > {};
03338 template<> struct priority_tag<0> {};
03339
03340 // taken from ranges-v3
03341 template<typename T>
03342 struct static_const
03343 {
03344     static JSON_INLINE_VARIABLE constexpr T value{};
03345 };
03346
03347 #ifndef JSON_HAS_CPP_17
03348     template<typename T>
03349     constexpr T static_const<T>::value;
03350 #endif
03351
03352 template<typename T, typename... Args>
03353 constexpr std::array<T, sizeof...(Args)> make_array(Args&& ... args)
03354 {
03355     return std::array<T, sizeof...(Args)> {{static_cast<T>(std::forward<Args>(args))...}};
03356 }
03357
03358 }  // namespace detail
03359 NLOHMANN_JSON_NAMESPACE_END
03360
03361 // #include <nlohmann/detail/meta/type_traits.hpp>
03362 //     __ _____ _____ _____
03363 //  __|  |   __|     |   | |  JSON for Modern C++
03364 // |  |  |__   |  |  | | | |  version 3.12.0
03365 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
03366 //
03367 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
03368 // SPDX-License-Identifier: MIT
03369
03370
03371
03372 #include <limits> // numeric_limits
03373 #include <string> // char_traits
03374 #include <tuple> // tuple
03375 #include <type_traits> // false_type, is_constructible, is_integral, is_same, true_type
03376 #include <utility> // declval
03377 #if defined(__cpp_lib_byte) && __cpp_lib_byte >= 201603L
03378     #include <cstddef> // byte
03379 #endif
03380 // #include <nlohmann/detail/iterators/iterator_traits.hpp>
03381 //     __ _____ _____ _____
03382 //  __|  |   __|     |   | |  JSON for Modern C++
03383 // |  |  |__   |  |  | | | |  version 3.12.0
03384 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
03385 //
03386 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
```

```
03387 // SPDX-License-Identifier: MIT
03388
03389
03390
03391 #include <iterator> // random_access_iterator_tag
03392
03393 // #include <nlohmann/detail/abi_macros.hpp>
03394
03395 // #include <nlohmann/detail/meta/void_t.hpp>
03396
03397 // #include <nlohmann/detail/meta/cpp_future.hpp>
03398
03399
03400 NLOHMANN_JSON_NAMESPACE_BEGIN
03401 namespace detail
03402 {
03403
03404 template<typename It, typename = void>
03405 struct iterator_types {};
03406
03407 template<typename It>
03408 struct iterator_types <
03409     It,
03410     void_t<typename It::difference_type, typename It::value_type, typename It::pointer,
03411     typename It::reference, typename It::iterator_category »
03412 {
03413     using difference_type = typename It::difference_type;
03414     using value_type = typename It::value_type;
03415     using pointer = typename It::pointer;
03416     using reference = typename It::reference;
03417     using iterator_category = typename It::iterator_category;
03418 };
03419
03420 // This is required as some compilers implement std::iterator_traits in a way that
03421 // doesn't work with SFINAE. See https://github.com/nlohmann/json/issues/1341.
03422 template<typename T, typename = void>
03423 struct iterator_traits
03424 {
03425 };
03426
03427 template<typename T>
03428 struct iterator_traits < T, enable_if_t < !std::is_pointer<T>::value »
03429     : iterator_types<T>
03430 {
03431 };
03432
03433 template<typename T>
03434 struct iterator_traits<T*, enable_if_t<std::is_object<T>::value»
03435 {
03436     using iterator_category = std::random_access_iterator_tag;
03437     using value_type = T;
03438     using difference_type = ptrdiff_t;
03439     using pointer = T*;
03440     using reference = T&;
03441 };
03442
03443 }  // namespace detail
03444 NLOHMANN_JSON_NAMESPACE_END
03445
03446 // #include <nlohmann/detail/macro_scope.hpp>
03447
03448 // #include <nlohmann/detail/meta/call_std/begin.hpp>
03449 //     __ _____ _____ _____
03450 //  __|  |   __|     |   | |  JSON for Modern C++
03451 // |  |  |__   |  |  | | | |  version 3.12.0
03452 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
03453 //
03454 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
03455 // SPDX-License-Identifier: MIT
03456
03457
03458
03459 // #include <nlohmann/detail/macro_scope.hpp>
03460
03461
03462 NLOHMANN_JSON_NAMESPACE_BEGIN
03463
03464 NLOHMANN_CAN_CALL_STD_FUNC_IMPL(begin);
03465
03466 NLOHMANN_JSON_NAMESPACE_END
03467
03468 // #include <nlohmann/detail/meta/call_std/end.hpp>
03469 //     __ _____ _____ _____
03470 //  __|  |   __|     |   | |  JSON for Modern C++
03471 // |  |  |__   |  |  | | | |  version 3.12.0
03472 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
03473 //
```

```
03474 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
03475 // SPDX-License-Identifier: MIT
03476
03477
03478
03479 // #include <nlohmann/detail/macro_scope.hpp>
03480
03481
03482 NLOHMANN_JSON_NAMESPACE_BEGIN
03483
03484 NLOHMANN_CAN_CALL_STD_FUNC_IMPL(end);
03485
03486 NLOHMANN_JSON_NAMESPACE_END
03487
03488 // #include <nlohmann/detail/meta/cpp_future.hpp>
03489
03490 // #include <nlohmann/detail/meta/detected.hpp>
03491
03492 // #include <nlohmann/json_fwd.hpp>
03493 //      __ _____ _____ _____
03494 //   __|  |   __|     |   | |  JSON for Modern C++
03495 // |  |  |__   |  |  | | | |  version 3.12.0
03496 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
03497 //
03498 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
03499 // SPDX-License-Identifier: MIT
03500
03501 #ifndef INCLUDE_NLOHMANN_JSON_FWD_HPP_
03502     #define INCLUDE_NLOHMANN_JSON_FWD_HPP_
03503
03504     #include <cstdint> // int64_t, uint64_t
03505     #include <map> // map
03506     #include <memory> // allocator
03507     #include <string> // string
03508     #include <vector> // vector
03509
03510     // #include <nlohmann/detail/abi_macros.hpp>
03511
03512
03518     NLOHMANN_JSON_NAMESPACE_BEGIN
03519
03527     template<typename T = void, typename SFINAE = void>
03528     struct adl_serializer;
03529
03532     template<template<typename U, typename V, typename... Args> class ObjectType =
03533     std::map,
03534     template<typename U, typename... Args> class ArrayType = std::vector,
03535     class StringType = std::string, class BooleanType = bool,
03536     class NumberIntegerType = std::int64_t,
03537     class NumberUnsignedType = std::uint64_t,
03538     class NumberFloatType = double,
03539     template<typename U> class AllocatorType = std::allocator,
03540     template<typename T, typename SFINAE = void> class JSONSerializer =
03541     adl_serializer,
03542     class BinaryType = std::vector<std::uint8_t>, // cppcheck-suppress syntaxError
03543     class CustomBaseClass = void>
03544     class basic_json;
03545
03548     template<typename RefStringType>
03549     class json_pointer;
03550
03555     using json = basic_json<>;
03556
03559     template<class Key, class T, class IgnoredLess, class Allocator>
03560     struct ordered_map;
03561
03564     using ordered_json = basic_json<nlohmann::ordered_map>;
03565
03566     NLOHMANN_JSON_NAMESPACE_END
03567
03568 #endif  // INCLUDE_NLOHMANN_JSON_FWD_HPP_
03569
03570
03571 NLOHMANN_JSON_NAMESPACE_BEGIN
03580 namespace detail
03581 {
03582
03584 // helpers //
03586
03587 // Note to maintainers:
03588 //
03589 // Every trait in this file expects a non-CV-qualified type.
03590 // The only exceptions are in the 'aliases for detected' section
03591 // (i.e., those of the form: decltype(T::member_function(std::declval<T>())))
03592 //
03593 // In this case, T has to be properly CV-qualified to constraint the function arguments
03594 // (e.g., to_json(BasicJsonType&, const T&))
```

```
03595
03596 template<typename> struct is_basic_json : std::false_type {};
03597
03598 NLOHMANN_BASIC_JSON_TPL_DECLARATION
03599 struct is_basic_json<NLOHMANN_BASIC_JSON_TPL> : std::true_type {};
03600
03601 // used by exceptions create() member functions
03602 // true_type for the pointer to possibly cv-qualified basic_json or std::nullptr_t
03603 // false_type otherwise
03604 template<typename BasicJsonContext>
03605 struct is_basic_json_context :
03606     std::integral_constant < bool,
03607     is_basic_json<typename std::remove_cv<typename
      std::remove_pointer<BasicJsonContext>::type>::type>::value
03608     || std::is_same<BasicJsonContext, std::nullptr_t>::value >
03609 {};
03610
03612 // json_ref helpers //
03614
03615 template<typename>
03616 class json_ref;
03617
03618 template<typename>
03619 struct is_json_ref : std::false_type {};
03620
03621 template<typename T>
03622 struct is_json_ref<json_ref<T» : std::true_type {};
03623
03625 // aliases for detected //
03627
03628 template<typename T>
03629 using mapped_type_t = typename T::mapped_type;
03630
03631 template<typename T>
03632 using key_type_t = typename T::key_type;
03633
03634 template<typename T>
03635 using value_type_t = typename T::value_type;
03636
03637 template<typename T>
03638 using difference_type_t = typename T::difference_type;
03639
03640 template<typename T>
03641 using pointer_t = typename T::pointer;
03642
03643 template<typename T>
03644 using reference_t = typename T::reference;
03645
03646 template<typename T>
03647 using iterator_category_t = typename T::iterator_category;
03648
03649 template<typename T, typename... Args>
03650 using to_json_function = decltype(T::to_json(std::declval<Args>()...));
03651
03652 template<typename T, typename... Args>
03653 using from_json_function = decltype(T::from_json(std::declval<Args>()...));
03654
03655 template<typename T, typename U>
03656 using get_template_function = decltype(std::declval<T>().template get<U>());
03657
03658 // trait checking if JSONSerializer<T>::from_json(json const&, udt&) exists
03659 template<typename BasicJsonType, typename T, typename = void>
03660 struct has_from_json : std::false_type {};
03661
03662 // trait checking if j.get<T> is valid
03663 // use this trait instead of std::is_constructible or std::is_convertible,
03664 // both rely on, or make use of implicit conversions, and thus fail when T
03665 // has several constructors/operator= (see https://github.com/nlohmann/json/issues/958)
03666 template <typename BasicJsonType, typename T>
03667 struct is_getable
03668 {
03669     static constexpr bool value = is_detected<get_template_function, const BasicJsonType&, T>::value;
03670 };
03671
03672 template<typename BasicJsonType, typename T>
03673 struct has_from_json < BasicJsonType, T, enable_if_t < !is_basic_json<T>::value »
03674 {
03675     using serializer = typename BasicJsonType::template json_serializer<T, void>;
03676
03677     static constexpr bool value =
03678         is_detected_exact<void, from_json_function, serializer,
03679         const BasicJsonType&, T&>::value;
03680 };
03681
03682 // This trait checks if JSONSerializer<T>::from_json(json const&) exists
03683 // this overload is used for non-default-constructible user-defined-types
03684 template<typename BasicJsonType, typename T, typename = void>
```

```
03685 struct has_non_default_from_json : std::false_type {};
03686
03687 template<typename BasicJsonType, typename T>
03688 struct has_non_default_from_json < BasicJsonType, T, enable_if_t < !is_basic_json<T>::value »
03689 {
03690     using serializer = typename BasicJsonType::template json_serializer<T, void>;
03691
03692     static constexpr bool value =
03693         is_detected_exact<T, from_json_function, serializer,
03694         const BasicJsonType&>::value;
03695 };
03696
03697 // This trait checks if BasicJsonType::json_serializer<T>::to_json exists
03698 // Do not evaluate the trait when T is a basic_json type, to avoid template instantiation infinite
      recursion.
03699 template<typename BasicJsonType, typename T, typename = void>
03700 struct has_to_json : std::false_type {};
03701
03702 template<typename BasicJsonType, typename T>
03703 struct has_to_json < BasicJsonType, T, enable_if_t < !is_basic_json<T>::value »
03704 {
03705     using serializer = typename BasicJsonType::template json_serializer<T, void>;
03706
03707     static constexpr bool value =
03708         is_detected_exact<void, to_json_function, serializer, BasicJsonType&,
03709         T>::value;
03710 };
03711
03712 template<typename T>
03713 using detect_key_compare = typename T::key_compare;
03714
03715 template<typename T>
03716 struct has_key_compare : std::integral_constant<bool, is_detected<detect_key_compare, T>::value> {};
03717
03718 // obtains the actual object key comparator
03719 template<typename BasicJsonType>
03720 struct actual_object_comparator
03721 {
03722     using object_t = typename BasicJsonType::object_t;
03723     using object_comparator_t = typename BasicJsonType::default_object_comparator_t;
03724     using type = typename std::conditional < has_key_compare<object_t>::value,
03725           typename object_t::key_compare, object_comparator_t>::type;
03726 };
03727
03728 template<typename BasicJsonType>
03729 using actual_object_comparator_t = typename actual_object_comparator<BasicJsonType>::type;
03730
03732 // char_traits //
03734
03735 // Primary template of char_traits calls std char_traits
03736 template<typename T>
03737 struct char_traits : std::char_traits<T>
03738 {};
03739
03740 // Explicitly define char traits for unsigned char since it is not standard
03741 template<>
03742 struct char_traits<unsigned char> : std::char_traits<char>
03743 {
03744     using char_type = unsigned char;
03745     using int_type = uint64_t;
03746
03747     // Redefine to_int_type function
03748     static int_type to_int_type(char_type c) noexcept
03749     {
03750         return static_cast<int_type>(c);
03751     }
03752
03753     static char_type to_char_type(int_type i) noexcept
03754     {
03755         return static_cast<char_type>(i);
03756     }
03757
03758     static constexpr int_type eof() noexcept
03759     {
03760         return static_cast<int_type>(std::char_traits<char>::eof());
03761     }
03762 };
03763
03764 // Explicitly define char traits for signed char since it is not standard
03765 template<>
03766 struct char_traits<signed char> : std::char_traits<char>
03767 {
03768     using char_type = signed char;
03769     using int_type = uint64_t;
03770
03771     // Redefine to_int_type function
03772     static int_type to_int_type(char_type c) noexcept
```

```
03773     {
03774         return static_cast<int_type>(c);
03775     }
03776
03777     static char_type to_char_type(int_type i) noexcept
03778     {
03779         return static_cast<char_type>(i);
03780     }
03781
03782     static constexpr int_type eof() noexcept
03783     {
03784         return static_cast<int_type>(std::char_traits<char>::eof());
03785     }
03786 };
03787
03788 #if defined(__cpp_lib_byte) && __cpp_lib_byte >= 201603L
03789 template<>
03790 struct char_traits<std::byte> : std::char_traits<char>
03791 {
03792     using char_type = std::byte;
03793     using int_type = uint64_t;
03794
03795     static int_type to_int_type(char_type c) noexcept
03796     {
03797         return static_cast<int_type>(std::to_integer<unsigned char>(c));
03798     }
03799
03800     static char_type to_char_type(int_type i) noexcept
03801     {
03802         return std::byte(static_cast<unsigned char>(i));
03803     }
03804
03805     static constexpr int_type eof() noexcept
03806     {
03807         return static_cast<int_type>(std::char_traits<char>::eof());
03808     }
03809 };
03810 #endif
03811
03813 // is_ functions //
03815
03816 // https://en.cppreference.com/w/cpp/types/conjunction
03817 template<class...> struct conjunction : std::true_type { };
03818 template<class B> struct conjunction<B> : B { };
03819 template<class B, class... Bn>
03820 struct conjunction<B, Bn...>
03821 : std::conditional<static_cast<bool>(B::value), conjunction<Bn...>, B>::type {};
03822
03823 // https://en.cppreference.com/w/cpp/types/negation
03824 template<class B> struct negation : std::integral_constant < bool, !B::value > { };
03825
03826 // Reimplementation of is_constructible and is_default_constructible, due to them being broken for
03827 // std::pair and std::tuple until LWG 2367 fix (see
      https://cplusplus.github.io/LWG/lwg-defects.html#2367).
03828 // This causes compile errors in e.g., Clang 3.5 or GCC 4.9.
03829 template <typename T>
03830 struct is_default_constructible : std::is_default_constructible<T> {};
03831
03832 template <typename T1, typename T2>
03833 struct is_default_constructible<std::pair<T1, T2»
03834     : conjunction<is_default_constructible<T1>, is_default_constructible<T2» {};
03835
03836 template <typename T1, typename T2>
03837 struct is_default_constructible<const std::pair<T1, T2»
03838     : conjunction<is_default_constructible<T1>, is_default_constructible<T2» {};
03839
03840 template <typename... Ts>
03841 struct is_default_constructible<std::tuple<Ts...»
03842     : conjunction<is_default_constructible<Ts>...> {};
03843
03844 template <typename... Ts>
03845 struct is_default_constructible<const std::tuple<Ts...»
03846     : conjunction<is_default_constructible<Ts>...> {};
03847
03848 template <typename T, typename... Args>
03849 struct is_constructible : std::is_constructible<T, Args...> {};
03850
03851 template <typename T1, typename T2>
03852 struct is_constructible<std::pair<T1, T2» : is_default_constructible<std::pair<T1, T2» {};
03853
03854 template <typename T1, typename T2>
03855 struct is_constructible<const std::pair<T1, T2» : is_default_constructible<const std::pair<T1, T2» {};
03856
03857 template <typename... Ts>
03858 struct is_constructible<std::tuple<Ts...» : is_default_constructible<std::tuple<Ts...» {};
03859
03860 template <typename... Ts>
```

```
03861 struct is_constructible<const std::tuple<Ts...>> : is_default_constructible<const std::tuple<Ts...>> {};
03862
03863 template<typename T, typename = void>
03864 struct is_iterator_traits : std::false_type {};
03865
03866 template<typename T>
03867 struct is_iterator_traits<iterator_traits<T>>
03868 {
03869   private:
03870     using traits = iterator_traits<T>;
03871
03872   public:
03873     static constexpr auto value =
03874         is_detected<value_type_t, traits>::value &&
03875         is_detected<difference_type_t, traits>::value &&
03876         is_detected<pointer_t, traits>::value &&
03877         is_detected<iterator_category_t, traits>::value &&
03878         is_detected<reference_t, traits>::value;
03879 };
03880
03881 template<typename T>
03882 struct is_range
03883 {
03884   private:
03885     using t_ref = typename std::add_lvalue_reference<T>::type;
03886
03887     using iterator = detected_t<result_of_begin, t_ref>;
03888     using sentinel = detected_t<result_of_end, t_ref>;
03889
03890     // to be 100% correct, it should use
      https://en.cppreference.com/w/cpp/iterator/input_or_output_iterator
03891     // and https://en.cppreference.com/w/cpp/iterator/sentinel_for
03892     // but reimplementing these would be too much work, as a lot of other concepts are used underneath
03893     static constexpr auto is_iterator_begin =
03894         is_iterator_traits<iterator_traits<iterator>>::value;
03895
03896   public:
03897     static constexpr bool value = !std::is_same<iterator, nonesuch>::value && !std::is_same<sentinel,
      nonesuch>::value && is_iterator_begin;
03898 };
03899
03900 template<typename R>
03901 using iterator_t = enable_if_t<is_range<R>::value, result_of_begin<decltype(std::declval<R&>())>>;
03902
03903 template<typename T>
03904 using range_value_t = value_type_t<iterator_traits<iterator_t<T>>>;
03905
03906 // The following implementation of is_complete_type is taken from
03907 //
      https://blogs.msdn.microsoft.com/vcblog/2015/12/02/partial-support-for-expression-sfinae-in-vs-2015-update-1/
03908 // and is written by Xiang Fan who agreed to use it in this library.
03909
03910 template<typename T, typename = void>
03911 struct is_complete_type : std::false_type {};
03912
03913 template<typename T>
03914 struct is_complete_type<T, decltype(void(sizeof(T)))> : std::true_type {};
03915
03916 template<typename BasicJsonType, typename CompatibleObjectType,
03917          typename = void>
03918 struct is_compatible_object_type_impl : std::false_type {};
03919
03920 template<typename BasicJsonType, typename CompatibleObjectType>
03921 struct is_compatible_object_type_impl <
03922     BasicJsonType, CompatibleObjectType,
03923     enable_if_t < is_detected<mapped_type_t, CompatibleObjectType>::value&&
03924     is_detected<key_type_t, CompatibleObjectType>::value >>
03925 {
03926     using object_t = typename BasicJsonType::object_t;
03927
03928     // macOS's is_constructible does not play well with nonesuch...
03929     static constexpr bool value =
03930         is_constructible<typename object_t::key_type,
03931         typename CompatibleObjectType::key_type>::value &&
03932         is_constructible<typename object_t::mapped_type,
03933         typename CompatibleObjectType::mapped_type>::value;
03934 };
03935
03936 template<typename BasicJsonType, typename CompatibleObjectType>
03937 struct is_compatible_object_type
03938     : is_compatible_object_type_impl<BasicJsonType, CompatibleObjectType> {};
03939
03940 template<typename BasicJsonType, typename ConstructibleObjectType,
03941          typename = void>
03942 struct is_constructible_object_type_impl : std::false_type {};
03943
03944 template<typename BasicJsonType, typename ConstructibleObjectType>
```

```
03945 struct is_constructible_object_type_impl <
03946     BasicJsonType, ConstructibleObjectType,
03947     enable_if_t < is_detected<mapped_type_t, ConstructibleObjectType>::value&&
03948     is_detected<key_type_t, ConstructibleObjectType>::value >
03949 {
03950     using object_t = typename BasicJsonType::object_t;
03951
03952     static constexpr bool value =
03953         (is_default_constructible<ConstructibleObjectType>::value &&
03954          (std::is_move_assignable<ConstructibleObjectType>::value ||
03955           std::is_copy_assignable<ConstructibleObjectType>::value) &&
03956          (is_constructible<typename ConstructibleObjectType::key_type,
03957           typename object_t::key_type>::value &&
03958           std::is_same <
03959           typename object_t::mapped_type,
03960           typename ConstructibleObjectType::mapped_type >::value)) ||
03961         (has_from_json<BasicJsonType,
03962          typename ConstructibleObjectType::mapped_type>::value ||
03963          has_non_default_from_json <
03964          BasicJsonType,
03965          typename ConstructibleObjectType::mapped_type >::value);
03966 };
03967
03968 template<typename BasicJsonType, typename ConstructibleObjectType>
03969 struct is_constructible_object_type
03970     : is_constructible_object_type_impl<BasicJsonType,
03971       ConstructibleObjectType> {};
03972
03973 template<typename BasicJsonType, typename CompatibleStringType>
03974 struct is_compatible_string_type
03975 {
03976     static constexpr auto value =
03977         is_constructible<typename BasicJsonType::string_t, CompatibleStringType>::value;
03978 };
03979
03980 template<typename BasicJsonType, typename ConstructibleStringType>
03981 struct is_constructible_string_type
03982 {
03983     // launder type through decltype() to fix compilation failure on ICPC
03984 #ifdef __INTEL_COMPILER
03985     using laundered_type = decltype(std::declval<ConstructibleStringType>());
03986 #else
03987     using laundered_type = ConstructibleStringType;
03988 #endif
03989
03990     static constexpr auto value =
03991         conjunction <
03992         is_constructible<laundered_type, typename BasicJsonType::string_t>,
03993         is_detected_exact<typename BasicJsonType::string_t::value_type,
03994         value_type_t, laundered_type >::value;
03995 };
03996
03997 template<typename BasicJsonType, typename CompatibleArrayType, typename = void>
03998 struct is_compatible_array_type_impl : std::false_type {};
03999
04000 template<typename BasicJsonType, typename CompatibleArrayType>
04001 struct is_compatible_array_type_impl <
04002     BasicJsonType, CompatibleArrayType,
04003     enable_if_t <
04004     is_detected<iterator_t, CompatibleArrayType>::value&&
04005     is_iterator_traits<iterator_traits<detected_t<iterator_t, CompatibleArrayType>>::value&&
04006 // special case for types like std::filesystem::path whose iterator's value_type are themselves
04007 // c.f. https://github.com/nlohmann/json/pull/3073
04008     !std::is_same<CompatibleArrayType, detected_t<range_value_t, CompatibleArrayType>::value >
04009 {
04010     static constexpr bool value =
04011         is_constructible<BasicJsonType,
04012         range_value_t<CompatibleArrayType>::value;
04013 };
04014
04015 template<typename BasicJsonType, typename CompatibleArrayType>
04016 struct is_compatible_array_type
04017     : is_compatible_array_type_impl<BasicJsonType, CompatibleArrayType> {};
04018
04019 template<typename BasicJsonType, typename ConstructibleArrayType, typename = void>
04020 struct is_constructible_array_type_impl : std::false_type {};
04021
04022 template<typename BasicJsonType, typename ConstructibleArrayType>
04023 struct is_constructible_array_type_impl <
04024     BasicJsonType, ConstructibleArrayType,
04025     enable_if_t<std::is_same<ConstructibleArrayType,
04026     typename BasicJsonType::value_type>::value >
04027             : std::true_type {};
04028
04029 template<typename BasicJsonType, typename ConstructibleArrayType>
04030 struct is_constructible_array_type_impl <
04031     BasicJsonType, ConstructibleArrayType,
```

```
04032     enable_if_t < !std::is_same<ConstructibleArrayType,
04033     typename BasicJsonType::value_type>::value&&
04034     !is_compatible_string_type<BasicJsonType, ConstructibleArrayType>::value&&
04035     is_default_constructible<ConstructibleArrayType>::value&&
04036 (std::is_move_assignable<ConstructibleArrayType>::value ||
04037  std::is_copy_assignable<ConstructibleArrayType>::value)&&
04038 is_detected<iterator_t, ConstructibleArrayType>::value&&
04039 is_iterator_traits<iterator_traits<detected_t<iterator_t, ConstructibleArrayType»>::value&&
04040 is_detected<range_value_t, ConstructibleArrayType>::value&&
04041 // special case for types like std::filesystem::path whose iterator's value_type are themselves
04042 // c.f. https://github.com/nlohmann/json/pull/3073
04043 !std::is_same<ConstructibleArrayType, detected_t<range_value_t, ConstructibleArrayType»::value&&
04044 is_complete_type <
04045 detected_t<range_value_t, ConstructibleArrayType »::value »
04046 {
04047     using value_type = range_value_t<ConstructibleArrayType>;
04048
04049     static constexpr bool value =
04050         std::is_same<value_type,
04051         typename BasicJsonType::array_t::value_type>::value ||
04052         has_from_json<BasicJsonType,
04053         value_type>::value ||
04054         has_non_default_from_json <
04055         BasicJsonType,
04056         value_type >::value;
04057 };
04058
04059 template<typename BasicJsonType, typename ConstructibleArrayType>
04060 struct is_constructible_array_type
04061     : is_constructible_array_type_impl<BasicJsonType, ConstructibleArrayType> {};
04062
04063 template<typename RealIntegerType, typename CompatibleNumberIntegerType,
04064          typename = void>
04065 struct is_compatible_integer_type_impl : std::false_type {};
04066
04067 template<typename RealIntegerType, typename CompatibleNumberIntegerType>
04068 struct is_compatible_integer_type_impl <
04069     RealIntegerType, CompatibleNumberIntegerType,
04070     enable_if_t < std::is_integral<RealIntegerType>::value&&
04071     std::is_integral<CompatibleNumberIntegerType>::value&&
04072     !std::is_same<bool, CompatibleNumberIntegerType>::value »
04073 {
04074     // is there an assert somewhere on overflows?
04075     using RealLimits = std::numeric_limits<RealIntegerType>;
04076     using CompatibleLimits = std::numeric_limits<CompatibleNumberIntegerType>;
04077
04078     static constexpr auto value =
04079         is_constructible<RealIntegerType,
04080         CompatibleNumberIntegerType>::value &&
04081         CompatibleLimits::is_integer &&
04082         RealLimits::is_signed == CompatibleLimits::is_signed;
04083 };
04084
04085 template<typename RealIntegerType, typename CompatibleNumberIntegerType>
04086 struct is_compatible_integer_type
04087     : is_compatible_integer_type_impl<RealIntegerType,
04088       CompatibleNumberIntegerType> {};
04089
04090 template<typename BasicJsonType, typename CompatibleType, typename = void>
04091 struct is_compatible_type_impl: std::false_type {};
04092
04093 template<typename BasicJsonType, typename CompatibleType>
04094 struct is_compatible_type_impl <
04095     BasicJsonType, CompatibleType,
04096     enable_if_t<is_complete_type<CompatibleType>::value »
04097 {
04098     static constexpr bool value =
04099         has_to_json<BasicJsonType, CompatibleType>::value;
04100 };
04101
04102 template<typename BasicJsonType, typename CompatibleType>
04103 struct is_compatible_type
04104     : is_compatible_type_impl<BasicJsonType, CompatibleType> {};
04105
04106 template<typename T1, typename T2>
04107 struct is_constructible_tuple : std::false_type {};
04108
04109 template<typename T1, typename... Args>
04110 struct is_constructible_tuple<T1, std::tuple<Args...» : conjunction<is_constructible<T1, Args>...> {};
04111
04112 template<typename BasicJsonType, typename T>
04113 struct is_json_iterator_of : std::false_type {};
04114
04115 template<typename BasicJsonType>
04116 struct is_json_iterator_of<BasicJsonType, typename BasicJsonType::iterator> : std::true_type {};
04117
04118 template<typename BasicJsonType>
```

```
04119 struct is_json_iterator_of<BasicJsonType, typename BasicJsonType::const_iterator> : std::true_type
04120 {};
04121
04122 // checks if a given type T is a template specialization of Primary
04123 template<template <typename...> class Primary, typename T>
04124 struct is_specialization_of : std::false_type {};
04125
04126 template<template <typename...> class Primary, typename... Args>
04127 struct is_specialization_of<Primary, Primary<Args...>> : std::true_type {};
04128
04129 template<typename T>
04130 using is_json_pointer = is_specialization_of<::nlohmann::json_pointer, uncvref_t<T>>;
04131
04132 // checks if B is a json_pointer<A>
04133 template <typename A, typename B>
04134 struct is_json_pointer_of : std::false_type {};
04135
04136 template <typename A>
04137 struct is_json_pointer_of<A, ::nlohmann::json_pointer<A>> : std::true_type {};
04138
04139 template <typename A>
04140 struct is_json_pointer_of<A, ::nlohmann::json_pointer<A>&> : std::true_type {};
04141
04142 // checks if A and B are comparable using Compare functor
04143 template<typename Compare, typename A, typename B, typename = void>
04144 struct is_comparable : std::false_type {};
04145
04146 // We exclude json_pointer here, because the checks using Compare(A, B) will
04147 // use json_pointer::operator string_t() which triggers a deprecation warning
04148 // for GCC. See https://github.com/nlohmann/json/issues/4621. The call to
04149 // is_json_pointer_of can be removed once the deprecated function has been
04150 // removed.
04151 template<typename Compare, typename A, typename B>
04152 struct is_comparable < Compare, A, B, enable_if_t < !is_json_pointer_of<A, B>::value
04153 && std::is_constructible <decltype(std::declval<Compare>()(std::declval<A>(),
      std::declval<B>()))>::value
04154 && std::is_constructible <decltype(std::declval<Compare>()(std::declval<B>(),
      std::declval<A>()))>::value
04155 >> : std::true_type {};
04156
04157 template<typename T>
04158 using detect_is_transparent = typename T::is_transparent;
04159
04160 // type trait to check if KeyType can be used as an object key (without a BasicJsonType)
04161 // see is_usable_as_basic_json_key_type below
04162 template<typename Comparator, typename ObjectKeyType, typename KeyTypeCVRef, bool
      RequireTransparentComparator = true,
04163          bool ExcludeObjectKeyType = RequireTransparentComparator, typename KeyType =
      uncvref_t<KeyTypeCVRef>>
04164 using is_usable_as_key_type = typename std::conditional <
04165                               is_comparable<Comparator, ObjectKeyType, KeyTypeCVRef>::value
04166                               && !(ExcludeObjectKeyType && std::is_same<KeyType,
04167                                    ObjectKeyType>::value)
04168                               && (!RequireTransparentComparator
04169                                   || is_detected <detect_is_transparent, Comparator>::value)
04170                               && !is_json_pointer<KeyType>::value,
04171                               std::true_type,
04172                               std::false_type >::type;
04173
04174 // type trait to check if KeyType can be used as an object key
04175 // true if:
04176 //   - KeyType is comparable with BasicJsonType::object_t::key_type
04177 //   - if ExcludeObjectKeyType is true, KeyType is not BasicJsonType::object_t::key_type
04178 //   - the comparator is transparent or RequireTransparentComparator is false
04179 //   - KeyType is not a JSON iterator or json_pointer
04180 template<typename BasicJsonType, typename KeyTypeCVRef, bool RequireTransparentComparator = true,
04181          bool ExcludeObjectKeyType = RequireTransparentComparator, typename KeyType =
      uncvref_t<KeyTypeCVRef>>
04182 using is_usable_as_basic_json_key_type = typename std::conditional <
04183     (is_usable_as_key_type<typename BasicJsonType::object_comparator_t,
04184      typename BasicJsonType::object_t::key_type, KeyTypeCVRef,
04185      RequireTransparentComparator, ExcludeObjectKeyType>::value
04186      && !is_json_iterator_of<BasicJsonType, KeyType>::value)
04187 #ifdef JSON_HAS_CPP_17
04188     || std::is_convertible<KeyType, std::string_view>::value
04189 #endif
04190     , std::true_type,
04191     std::false_type >::type;
04192
04193 template<typename ObjectType, typename KeyType>
04194 using detect_erase_with_key_type =
      decltype(std::declval<ObjectType&>().erase(std::declval<KeyType>()));
04195
04196 // type trait to check if object_t has an erase() member functions accepting KeyType
04197 template<typename BasicJsonType, typename KeyType>
04198 using has_erase_with_key_type = typename std::conditional <
04199                                 is_detected <
```

```
04200                                 detect_erase_with_key_type,
04201                                 typename BasicJsonType::object_t, KeyType >::value,
04202                                 std::true_type,
04203                                 std::false_type >::type;
04204
04205 // a naive helper to check if a type is an ordered_map (exploits the fact that
04206 // ordered_map inherits capacity() from std::vector)
04207 template <typename T>
04208 struct is_ordered_map
04209 {
04210     using one = char;
04211
04212     struct two
04213     {
04214         char x[2]; //
      NOLINT(cppcoreguidelines-avoid-c-arrays,hicpp-avoid-c-arrays,modernize-avoid-c-arrays)
04215     };
04216
04217     template <typename C> static one test( decltype(&C::capacity) ) ;
04218     template <typename C> static two test(...);
04219
04220     enum { value = sizeof(test<T>(nullptr)) == sizeof(char) }; //
      NOLINT(cppcoreguidelines-pro-type-vararg,hicpp-vararg,cppcoreguidelines-use-enum-class)
04221 };
04222
04223 // to avoid useless casts (see https://github.com/nlohmann/json/issues/2893#issuecomment-889152324)
04224 template < typename T, typename U, enable_if_t < !std::is_same<T, U>::value, int > = 0 >
04225 T conditional_static_cast(U value)
04226 {
04227     return static_cast<T>(value);
04228 }
04229
04230 template<typename T, typename U, enable_if_t<std::is_same<T, U>::value, int> = 0>
04231 T conditional_static_cast(U value)
04232 {
04233     return value;
04234 }
04235
04236 template<typename... Types>
04237 using all_integral = conjunction<std::is_integral<Types>...>;
04238
04239 template<typename... Types>
04240 using all_signed = conjunction<std::is_signed<Types>...>;
04241
04242 template<typename... Types>
04243 using all_unsigned = conjunction<std::is_unsigned<Types>...>;
04244
04245 // there's a disjunction trait in another PR; replace when merged
04246 template<typename... Types>
04247 using same_sign = std::integral_constant < bool,
04248       all_signed<Types...>::value || all_unsigned<Types...>::value >;
04249
04250 template<typename OfType, typename T>
04251 using never_out_of_range = std::integral_constant < bool,
04252       (std::is_signed<OfType>::value && (sizeof(T) < sizeof(OfType)))
04253       || (same_sign<OfType, T>::value && sizeof(OfType) == sizeof(T)) >;
04254
04255 template<typename OfType, typename T,
04256          bool OfTypeSigned = std::is_signed<OfType>::value,
04257          bool TSigned = std::is_signed<T>::value>
04258 struct value_in_range_of_impl2;
04259
04260 template<typename OfType, typename T>
04261 struct value_in_range_of_impl2<OfType, T, false, false>
04262 {
04263     static constexpr bool test(T val)
04264     {
04265         using CommonType = typename std::common_type<OfType, T>::type;
04266         return static_cast<CommonType>(val) <=
      static_cast<CommonType>((std::numeric_limits<OfType>::max)());
04267     }
04268 };
04269
04270 template<typename OfType, typename T>
04271 struct value_in_range_of_impl2<OfType, T, true, false>
04272 {
04273     static constexpr bool test(T val)
04274     {
04275         using CommonType = typename std::common_type<OfType, T>::type;
04276         return static_cast<CommonType>(val) <=
      static_cast<CommonType>((std::numeric_limits<OfType>::max)());
04277     }
04278 };
04279
04280 template<typename OfType, typename T>
04281 struct value_in_range_of_impl2<OfType, T, false, true>
04282 {
```

```
04283     static constexpr bool test(T val)
04284     {
04285         using CommonType = typename std::common_type<OfType, T>::type;
04286         return val >= 0 && static_cast<CommonType>(val) <=
      static_cast<CommonType>((std::numeric_limits<OfType>::max)());
04287     }
04288 };
04289
04290 template<typename OfType, typename T>
04291 struct value_in_range_of_impl2<OfType, T, true, true>
04292 {
04293     static constexpr bool test(T val)
04294     {
04295         using CommonType = typename std::common_type<OfType, T>::type;
04296         return static_cast<CommonType>(val) >=
      static_cast<CommonType>((std::numeric_limits<OfType>::min)())
04297                 && static_cast<CommonType>(val) <=
      static_cast<CommonType>((std::numeric_limits<OfType>::max)());
04298     }
04299 };
04300
04301 template<typename OfType, typename T,
04302          bool NeverOutOfRange = never_out_of_range<OfType, T>::value,
04303          typename = detail::enable_if_t<all_integral<OfType, T>::value»
04304 struct value_in_range_of_impl1;
04305
04306 template<typename OfType, typename T>
04307 struct value_in_range_of_impl1<OfType, T, false>
04308 {
04309     static constexpr bool test(T val)
04310     {
04311         return value_in_range_of_impl2<OfType, T>::test(val);
04312     }
04313 };
04314
04315 template<typename OfType, typename T>
04316 struct value_in_range_of_impl1<OfType, T, true>
04317 {
04318     static constexpr bool test(T /*val*/)
04319     {
04320         return true;
04321     }
04322 };
04323
04324 template<typename OfType, typename T>
04325 constexpr bool value_in_range_of(T val)
04326 {
04327     return value_in_range_of_impl1<OfType, T>::test(val);
04328 }
04329
04330 template<bool Value>
04331 using bool_constant = std::integral_constant<bool, Value>;
04332
04334 // is_c_string
04336
04337 namespace impl
04338 {
04339
04340 template<typename T>
04341 constexpr bool is_c_string()
04342 {
04343     using TUnExt = typename std::remove_extent<T>::type;
04344     using TUnCVExt = typename std::remove_cv<TUnExt>::type;
04345     using TUnPtr = typename std::remove_pointer<T>::type;
04346     using TUnCVPtr = typename std::remove_cv<TUnPtr>::type;
04347     return
04348         (std::is_array<T>::value && std::is_same<TUnCVExt, char>::value)
04349         || (std::is_pointer<T>::value && std::is_same<TUnCVPtr, char>::value);
04350 }
04351
04352 }  // namespace impl
04353
04354 // checks whether T is a [cv] char */[cv] char[] C string
04355 template<typename T>
04356 struct is_c_string : bool_constant<impl::is_c_string<T>()> {};
04357
04358 template<typename T>
04359 using is_c_string_uncvref = is_c_string<uncvref_t<T>>;
04360
04362 // is_transparent
04364
04365 namespace impl
04366 {
04367
04368 template<typename T>
04369 constexpr bool is_transparent()
04370 {
```

```
04371     return is_detected<detect_is_transparent, T>::value;
04372 }
04373
04374 }  // namespace impl
04375
04376 // checks whether T has a member named is_transparent
04377 template<typename T>
04378 struct is_transparent : bool_constant<impl::is_transparent<T>()> {};
04379
04381
04382 }  // namespace detail
04383 NLOHMANN_JSON_NAMESPACE_END
04384
04385 // #include <nlohmann/detail/string_concat.hpp>
04386 //      __ _____ _____ _____
04387 //   __|  |   __|     |   | |  JSON for Modern C++
04388 // |  |  |__   |  |  | | | |  version 3.12.0
04389 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
04390 //
04391 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
04392 // SPDX-License-Identifier: MIT
04393
04394
04395
04396 #include <cstring> // strlen
04397 #include <string> // string
04398 #include <utility> // forward
04399
04400 // #include <nlohmann/detail/meta/cpp_future.hpp>
04401
04402 // #include <nlohmann/detail/meta/detected.hpp>
04403
04404
04405 NLOHMANN_JSON_NAMESPACE_BEGIN
04406 namespace detail
04407 {
04408
04409 inline std::size_t concat_length()
04410 {
04411     return 0;
04412 }
04413
04414 template<typename... Args>
04415 inline std::size_t concat_length(const char* cstr, const Args& ... rest);
04416
04417 template<typename StringType, typename... Args>
04418 inline std::size_t concat_length(const StringType& str, const Args& ... rest);
04419
04420 template<typename... Args>
04421 inline std::size_t concat_length(const char /*c*/, const Args& ... rest)
04422 {
04423     return 1 + concat_length(rest...);
04424 }
04425
04426 template<typename... Args>
04427 inline std::size_t concat_length(const char* cstr, const Args& ... rest)
04428 {
04429     // cppcheck-suppress ignoredReturnValue
04430     return ::strlen(cstr) + concat_length(rest...);
04431 }
04432
04433 template<typename StringType, typename... Args>
04434 inline std::size_t concat_length(const StringType& str, const Args& ... rest)
04435 {
04436     return str.size() + concat_length(rest...);
04437 }
04438
04439 template<typename OutStringType>
04440 inline void concat_into(OutStringType& /*out*/)
04441 {}
04442
04443 template<typename StringType, typename Arg>
04444 using string_can_append = decltype(std::declval<StringType&>().append(std::declval < Arg && > ()));
04445
04446 template<typename StringType, typename Arg>
04447 using detect_string_can_append = is_detected<string_can_append, StringType, Arg>;
04448
04449 template<typename StringType, typename Arg>
04450 using string_can_append_op = decltype(std::declval<StringType&>() += std::declval < Arg && > ());
04451
04452 template<typename StringType, typename Arg>
04453 using detect_string_can_append_op = is_detected<string_can_append_op, StringType, Arg>;
04454
04455 template<typename StringType, typename Arg>
04456 using string_can_append_iter = decltype(std::declval<StringType&>().append(std::declval<const
      Arg&>().begin(), std::declval<const Arg&>().end()));
04457
```

```
04458 template<typename StringType, typename Arg>
04459 using detect_string_can_append_iter = is_detected<string_can_append_iter, StringType, Arg>;
04460
04461 template<typename StringType, typename Arg>
04462 using string_can_append_data = decltype(std::declval<StringType&>().append(std::declval<const
      Arg&>().data(), std::declval<const Arg&>().size()));
04463
04464 template<typename StringType, typename Arg>
04465 using detect_string_can_append_data = is_detected<string_can_append_data, StringType, Arg>;
04466
04467 template < typename OutStringType, typename Arg, typename... Args,
04468            enable_if_t < !detect_string_can_append<OutStringType, Arg>::value
04469                          && detect_string_can_append_op<OutStringType, Arg>::value, int > = 0 >
04470 inline void concat_into(OutStringType& out, Arg && arg, Args && ... rest);
04471
04472 template < typename OutStringType, typename Arg, typename... Args,
04473            enable_if_t < !detect_string_can_append<OutStringType, Arg>::value
04474                          && !detect_string_can_append_op<OutStringType, Arg>::value
04475                          && detect_string_can_append_iter<OutStringType, Arg>::value, int > = 0 >
04476 inline void concat_into(OutStringType& out, const Arg& arg, Args && ... rest);
04477
04478 template < typename OutStringType, typename Arg, typename... Args,
04479            enable_if_t < !detect_string_can_append<OutStringType, Arg>::value
04480                          && !detect_string_can_append_op<OutStringType, Arg>::value
04481                          && !detect_string_can_append_iter<OutStringType, Arg>::value
04482                          && detect_string_can_append_data<OutStringType, Arg>::value, int > = 0 >
04483 inline void concat_into(OutStringType& out, const Arg& arg, Args && ... rest);
04484
04485 template<typename OutStringType, typename Arg, typename... Args,
04486          enable_if_t<detect_string_can_append<OutStringType, Arg>::value, int> = 0>
04487 inline void concat_into(OutStringType& out, Arg && arg, Args && ... rest)
04488 {
04489     out.append(std::forward<Arg>(arg));
04490     concat_into(out, std::forward<Args>(rest)...);
04491 }
04492
04493 template < typename OutStringType, typename Arg, typename... Args,
04494            enable_if_t < !detect_string_can_append<OutStringType, Arg>::value
04495                          && detect_string_can_append_op<OutStringType, Arg>::value, int > >
04496 inline void concat_into(OutStringType& out, Arg&& arg, Args&& ... rest)
04497 {
04498     out += std::forward<Arg>(arg);
04499     concat_into(out, std::forward<Args>(rest)...);
04500 }
04501
04502 template < typename OutStringType, typename Arg, typename... Args,
04503            enable_if_t < !detect_string_can_append<OutStringType, Arg>::value
04504                          && !detect_string_can_append_op<OutStringType, Arg>::value
04505                          && detect_string_can_append_iter<OutStringType, Arg>::value, int > >
04506 inline void concat_into(OutStringType& out, const Arg& arg, Args&& ... rest)
04507 {
04508     out.append(arg.begin(), arg.end());
04509     concat_into(out, std::forward<Args>(rest)...);
04510 }
04511
04512 template < typename OutStringType, typename Arg, typename... Args,
04513            enable_if_t < !detect_string_can_append<OutStringType, Arg>::value
04514                          && !detect_string_can_append_op<OutStringType, Arg>::value
04515                          && !detect_string_can_append_iter<OutStringType, Arg>::value
04516                          && detect_string_can_append_data<OutStringType, Arg>::value, int > >
04517 inline void concat_into(OutStringType& out, const Arg& arg, Args&& ... rest)
04518 {
04519     out.append(arg.data(), arg.size());
04520     concat_into(out, std::forward<Args>(rest)...);
04521 }
04522
04523 template<typename OutStringType = std::string, typename... Args>
04524 inline OutStringType concat(Args && ... args)
04525 {
04526     OutStringType str;
04527     str.reserve(concat_length(args...));
04528     concat_into(str, std::forward<Args>(args)...);
04529     return str;
04530 }
04531
04532 }  // namespace detail
04533 NLOHMANN_JSON_NAMESPACE_END
04534
04535
04536 // With -Wweak-vtables, Clang will complain about the exception classes as they
04537 // have no out-of-line virtual method definitions and their vtable will be
04538 // emitted in every translation unit. This issue cannot be fixed with a
04539 // header-only library as there is no implementation file to move these
04540 // functions to. As a result, we suppress this warning here to avoid client
04541 // code stumbling over this. See https://github.com/nlohmann/json/issues/4087
04542 // for a discussion.
04543 #if defined(__clang__)
```

```
04544     #pragma clang diagnostic push
04545     #pragma clang diagnostic ignored "-Wweak-vtables"
04546 #endif
04547
04548 NLOHMANN_JSON_NAMESPACE_BEGIN
04549 namespace detail
04550 {
04551
04553 // exceptions //
04555
04558 class exception : public std::exception
04559 {
04560   public:
04562     const char* what() const noexcept override
04563     {
04564         return m.what();
04565     }
04566
04568     const int id; // NOLINT(cppcoreguidelines-non-private-member-variables-in-classes)
04569
04570   protected:
04571     JSON_HEDLEY_NON_NULL(3)
04572     exception(int id_, const char* what_arg) : id(id_), m(what_arg) {} //
      NOLINT(bugprone-throw-keyword-missing)
04573
04574     static std::string name(const std::string& ename, int id_)
04575     {
04576         return concat("[json.exception.", ename, '.', std::to_string(id_), "] ");
04577     }
04578
04579     static std::string diagnostics(std::nullptr_t /*leaf_element*/)
04580     {
04581         return "";
04582     }
04583
04584     template<typename BasicJsonType>
04585     static std::string diagnostics(const BasicJsonType* leaf_element)
04586     {
04587 #if JSON_DIAGNOSTICS
04588         std::vector<std::string> tokens;
04589         for (const auto* current = leaf_element; current != nullptr && current->m_parent != nullptr;
      current = current->m_parent)
04590         {
04591             switch (current->m_parent->type())
04592             {
04593                 case value_t::array:
04594                 {
04595                     for (std::size_t i = 0; i < current->m_parent->m_data.m_value.array->size(); ++i)
04596                     {
04597                         if (&current->m_parent->m_data.m_value.array->operator[](i) == current)
04598                         {
04599                             tokens.emplace_back(std::to_string(i));
04600                             break;
04601                         }
04602                     }
04603                     break;
04604                 }
04605
04606                 case value_t::object:
04607                 {
04608                     for (const auto& element : *current->m_parent->m_data.m_value.object)
04609                     {
04610                         if (&element.second == current)
04611                         {
04612                             tokens.emplace_back(element.first.c_str());
04613                             break;
04614                         }
04615                     }
04616                     break;
04617                 }
04618
04619                 case value_t::null: // LCOV_EXCL_LINE
04620                 case value_t::string: // LCOV_EXCL_LINE
04621                 case value_t::boolean: // LCOV_EXCL_LINE
04622                 case value_t::number_integer: // LCOV_EXCL_LINE
04623                 case value_t::number_unsigned: // LCOV_EXCL_LINE
04624                 case value_t::number_float: // LCOV_EXCL_LINE
04625                 case value_t::binary: // LCOV_EXCL_LINE
04626                 case value_t::discarded: // LCOV_EXCL_LINE
04627                 default:   // LCOV_EXCL_LINE
04628                     break; // LCOV_EXCL_LINE
04629             }
04630         }
04631
04632         if (tokens.empty())
04633         {
04634             return "";
```

```
04635         }
04636
04637         auto str = std::accumulate(tokens.rbegin(), tokens.rend(), std::string{},
04638                                    [](const std::string & a, const std::string & b)
04639         {
04640             return concat(a, '/', detail::escape(b));
04641         });
04642
04643         return concat('(', str, ") ", get_byte_positions(leaf_element));
04644 #else
04645         return get_byte_positions(leaf_element);
04646 #endif
04647     }
04648
04649   private:
04651     std::runtime_error m;
04652 #if JSON_DIAGNOSTIC_POSITIONS
04653     template<typename BasicJsonType>
04654     static std::string get_byte_positions(const BasicJsonType* leaf_element)
04655     {
04656         if ((leaf_element->start_pos() != std::string::npos) && (leaf_element->end_pos() !=
      std::string::npos))
04657         {
04658             return concat("(bytes ", std::to_string(leaf_element->start_pos()), "-",
      std::to_string(leaf_element->end_pos()), ") ");
04659         }
04660         return "";
04661     }
04662 #else
04663     template<typename BasicJsonType>
04664     static std::string get_byte_positions(const BasicJsonType* leaf_element)
04665     {
04666         static_cast<void>(leaf_element);
04667         return "";
04668     }
04669 #endif
04670 };
04671
04674 class parse_error : public exception
04675 {
04676   public:
04686     template<typename BasicJsonContext, enable_if_t<is_basic_json_context<BasicJsonContext>::value,
      int> = 0>
04687     static parse_error create(int id_, const position_t& pos, const std::string& what_arg,
      BasicJsonContext context)
04688     {
04689         const std::string w = concat(exception::name("parse_error", id_), "parse error",
04690                                      position_string(pos), ": ", exception::diagnostics(context),
      what_arg);
04691         return {id_, pos.chars_read_total, w.c_str()};
04692     }
04693
04694     template<typename BasicJsonContext, enable_if_t<is_basic_json_context<BasicJsonContext>::value,
      int> = 0>
04695     static parse_error create(int id_, std::size_t byte_, const std::string& what_arg,
      BasicJsonContext context)
04696     {
04697         const std::string w = concat(exception::name("parse_error", id_), "parse error",
04698                                      (byte_ != 0 ? (concat(" at byte ", std::to_string(byte_))) : ""),
04699                                      ": ", exception::diagnostics(context), what_arg);
04700         return {id_, byte_, w.c_str()};
04701     }
04702
04712     const std::size_t byte;
04713
04714   private:
04715     parse_error(int id_, std::size_t byte_, const char* what_arg)
04716         : exception(id_, what_arg), byte(byte_) {}
04717
04718     static std::string position_string(const position_t& pos)
04719     {
04720         return concat(" at line ", std::to_string(pos.lines_read + 1),
04721                       ", column ", std::to_string(pos.chars_read_current_line));
04722     }
04723 };
04724
04727 class invalid_iterator : public exception
04728 {
04729   public:
04730     template<typename BasicJsonContext, enable_if_t<is_basic_json_context<BasicJsonContext>::value,
      int> = 0>
04731     static invalid_iterator create(int id_, const std::string& what_arg, BasicJsonContext context)
04732     {
04733         const std::string w = concat(exception::name("invalid_iterator", id_),
      exception::diagnostics(context), what_arg);
04734         return {id_, w.c_str()};
04735     }
```

```
04736
04737   private:
04738     JSON_HEDLEY_NON_NULL(3)
04739     invalid_iterator(int id_, const char* what_arg)
04740         : exception(id_, what_arg) {}
04741 };
04742
04745 class type_error : public exception
04746 {
04747   public:
04748     template<typename BasicJsonContext, enable_if_t<is_basic_json_context<BasicJsonContext>::value,
      int> = 0>
04749     static type_error create(int id_, const std::string& what_arg, BasicJsonContext context)
04750     {
04751         const std::string w = concat(exception::name("type_error", id_),
      exception::diagnostics(context), what_arg);
04752         return {id_, w.c_str()};
04753     }
04754
04755   private:
04756     JSON_HEDLEY_NON_NULL(3)
04757     type_error(int id_, const char* what_arg) : exception(id_, what_arg) {}
04758 };
04759
04762 class out_of_range : public exception
04763 {
04764   public:
04765     template<typename BasicJsonContext, enable_if_t<is_basic_json_context<BasicJsonContext>::value,
      int> = 0>
04766     static out_of_range create(int id_, const std::string& what_arg, BasicJsonContext context)
04767     {
04768         const std::string w = concat(exception::name("out_of_range", id_),
      exception::diagnostics(context), what_arg);
04769         return {id_, w.c_str()};
04770     }
04771
04772   private:
04773     JSON_HEDLEY_NON_NULL(3)
04774     out_of_range(int id_, const char* what_arg) : exception(id_, what_arg) {}
04775 };
04776
04779 class other_error : public exception
04780 {
04781   public:
04782     template<typename BasicJsonContext, enable_if_t<is_basic_json_context<BasicJsonContext>::value,
      int> = 0>
04783     static other_error create(int id_, const std::string& what_arg, BasicJsonContext context)
04784     {
04785         const std::string w = concat(exception::name("other_error", id_),
      exception::diagnostics(context), what_arg);
04786         return {id_, w.c_str()};
04787     }
04788
04789   private:
04790     JSON_HEDLEY_NON_NULL(3)
04791     other_error(int id_, const char* what_arg) : exception(id_, what_arg) {}
04792 };
04793
04794 }  // namespace detail
04795 NLOHMANN_JSON_NAMESPACE_END
04796
04797 #if defined(__clang__)
04798     #pragma clang diagnostic pop
04799 #endif
04800
04801 // #include <nlohmann/detail/macro_scope.hpp>
04802
04803 // #include <nlohmann/detail/meta/cpp_future.hpp>
04804
04805 // #include <nlohmann/detail/meta/identity_tag.hpp>
04806 //     __ _____ _____ _____
04807 //  __|  |   __|     |   | |  JSON for Modern C++
04808 // |  |  |__   |  |  | | | |  version 3.12.0
04809 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
04810 //
04811 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
04812 // SPDX-License-Identifier: MIT
04813
04814
04815
04816 // #include <nlohmann/detail/abi_macros.hpp>
04817
04818
04819 NLOHMANN_JSON_NAMESPACE_BEGIN
04820 namespace detail
04821 {
04822
```

```
04823 // dispatching helper struct
04824 template <class T> struct identity_tag {};
04825
04826 }  // namespace detail
04827 NLOHMANN_JSON_NAMESPACE_END
04828
04829 // #include <nlohmann/detail/meta/std_fs.hpp>
04830 //     __ _____ _____ _____
04831 //  __|  |   __|     |   | |  JSON for Modern C++
04832 // |  |  |__   |  |  | | | |  version 3.12.0
04833 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
04834 //
04835 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
04836 // SPDX-License-Identifier: MIT
04837
04838
04839
04840 // #include <nlohmann/detail/macro_scope.hpp>
04841
04842
04843 #if JSON_HAS_EXPERIMENTAL_FILESYSTEM
04844 #include <experimental/filesystem>
04845 NLOHMANN_JSON_NAMESPACE_BEGIN
04846 namespace detail
04847 {
04848 namespace std_fs = std::experimental::filesystem;
04849 }  // namespace detail
04850 NLOHMANN_JSON_NAMESPACE_END
04851 #elif JSON_HAS_FILESYSTEM
04852 #include <filesystem> // NOLINT(build/c++17)
04853 NLOHMANN_JSON_NAMESPACE_BEGIN
04854 namespace detail
04855 {
04856 namespace std_fs = std::filesystem;
04857 }  // namespace detail
04858 NLOHMANN_JSON_NAMESPACE_END
04859 #endif
04860
04861 // #include <nlohmann/detail/meta/type_traits.hpp>
04862
04863 // #include <nlohmann/detail/string_concat.hpp>
04864
04865 // #include <nlohmann/detail/value_t.hpp>
04866
04867
04868 // include after macro_scope.hpp
04869 #ifdef JSON_HAS_CPP_17
04870     #include <optional> // optional
04871 #endif
04872
04873 #if JSON_HAS_FILESYSTEM || JSON_HAS_EXPERIMENTAL_FILESYSTEM
04874     #include <string_view> // u8string_view
04875 #endif
04876
04877 NLOHMANN_JSON_NAMESPACE_BEGIN
04878 namespace detail
04879 {
04880
04881 template<typename BasicJsonType>
04882 inline void from_json(const BasicJsonType& j, typename std::nullptr_t& n)
04883 {
04884     if (JSON_HEDLEY_UNLIKELY(!j.is_null()))
04885     {
04886         JSON_THROW(type_error::create(302, concat("type must be null, but is ", j.type_name()), &j));
04887     }
04888     n = nullptr;
04889 }
04890
04891 #ifdef JSON_HAS_CPP_17
04892 template < typename BasicJsonType, typename T,
04893            typename std::enable_if < !nlohmann::detail::is_basic_json<T>::value, int >::type = 0 >
04894 void from_json(const BasicJsonType& j, std::optional<T>& opt)
04895 {
04896     if (j.is_null())
04897     {
04898         opt = std::nullopt;
04899     }
04900     else
04901     {
04902         opt.emplace(j.template get<T>());
04903     }
04904 }
04905 #endif // JSON_HAS_CPP_17
04906
04907 // overloads for basic_json template parameters
04908 template < typename BasicJsonType, typename ArithmeticType,
04909            enable_if_t < std::is_arithmetic<ArithmeticType>::value&&
```

```
04910                         !std::is_same<ArithmeticType, typename BasicJsonType::boolean_t>::value,
04911                         int > = 0 >
04912 void get_arithmetic_value(const BasicJsonType& j, ArithmeticType& val)
04913 {
04914     switch (static_cast<value_t>(j))
04915     {
04916         case value_t::number_unsigned:
04917         {
04918             val = static_cast<ArithmeticType>(*j.template get_ptr<const typename
      BasicJsonType::number_unsigned_t*>());
04919             break;
04920         }
04921         case value_t::number_integer:
04922         {
04923             val = static_cast<ArithmeticType>(*j.template get_ptr<const typename
      BasicJsonType::number_integer_t*>());
04924             break;
04925         }
04926         case value_t::number_float:
04927         {
04928             val = static_cast<ArithmeticType>(*j.template get_ptr<const typename
      BasicJsonType::number_float_t*>());
04929             break;
04930         }
04931
04932         case value_t::null:
04933         case value_t::object:
04934         case value_t::array:
04935         case value_t::string:
04936         case value_t::boolean:
04937         case value_t::binary:
04938         case value_t::discarded:
04939         default:
04940             JSON_THROW(type_error::create(302, concat("type must be number, but is ", j.type_name()),
      &j));
04941     }
04942 }
04943
04944 template<typename BasicJsonType>
04945 inline void from_json(const BasicJsonType& j, typename BasicJsonType::boolean_t& b)
04946 {
04947     if (JSON_HEDLEY_UNLIKELY(!j.is_boolean()))
04948     {
04949         JSON_THROW(type_error::create(302, concat("type must be boolean, but is ", j.type_name()),
      &j));
04950     }
04951     b = *j.template get_ptr<const typename BasicJsonType::boolean_t*>();
04952 }
04953
04954 template<typename BasicJsonType>
04955 inline void from_json(const BasicJsonType& j, typename BasicJsonType::string_t& s)
04956 {
04957     if (JSON_HEDLEY_UNLIKELY(!j.is_string()))
04958     {
04959         JSON_THROW(type_error::create(302, concat("type must be string, but is ", j.type_name()),
      &j));
04960     }
04961     s = *j.template get_ptr<const typename BasicJsonType::string_t*>();
04962 }
04963
04964 template <
04965     typename BasicJsonType, typename StringType,
04966     enable_if_t <
04967         std::is_assignable<StringType&, const typename BasicJsonType::string_t>::value
04968         && is_detected_exact<typename BasicJsonType::string_t::value_type, value_type_t,
      StringType>::value
04969         && !std::is_same<typename BasicJsonType::string_t, StringType>::value
04970         && !is_json_ref<StringType>::value, int > = 0 >
04971 inline void from_json(const BasicJsonType& j, StringType& s)
04972 {
04973     if (JSON_HEDLEY_UNLIKELY(!j.is_string()))
04974     {
04975         JSON_THROW(type_error::create(302, concat("type must be string, but is ", j.type_name()),
      &j));
04976     }
04977
04978     s = *j.template get_ptr<const typename BasicJsonType::string_t*>();
04979 }
04980
04981 template<typename BasicJsonType>
04982 inline void from_json(const BasicJsonType& j, typename BasicJsonType::number_float_t& val)
04983 {
04984     get_arithmetic_value(j, val);
04985 }
04986
04987 template<typename BasicJsonType>
04988 inline void from_json(const BasicJsonType& j, typename BasicJsonType::number_unsigned_t& val)
```

```
04989 {
04990     get_arithmetic_value(j, val);
04991 }
04992
04993 template<typename BasicJsonType>
04994 inline void from_json(const BasicJsonType& j, typename BasicJsonType::number_integer_t& val)
04995 {
04996     get_arithmetic_value(j, val);
04997 }
04998
04999 #if !JSON_DISABLE_ENUM_SERIALIZATION
05000 template<typename BasicJsonType, typename EnumType,
05001          enable_if_t<std::is_enum<EnumType>::value, int> = 0>
05002 inline void from_json(const BasicJsonType& j, EnumType& e)
05003 {
05004     typename std::underlying_type<EnumType>::type val;
05005     get_arithmetic_value(j, val);
05006     e = static_cast<EnumType>(val);
05007 }
05008 #endif  // JSON_DISABLE_ENUM_SERIALIZATION
05009
05010 // forward_list doesn't have an insert method
05011 template<typename BasicJsonType, typename T, typename Allocator,
05012          enable_if_t<is_getable<BasicJsonType, T>::value, int> = 0>
05013 inline void from_json(const BasicJsonType& j, std::forward_list<T, Allocator>& l)
05014 {
05015     if (JSON_HEDLEY_UNLIKELY(!j.is_array()))
05016     {
05017         JSON_THROW(type_error::create(302, concat("type must be array, but is ", j.type_name()), &j));
05018     }
05019     l.clear();
05020     std::transform(j.rbegin(), j.rend(),
05021                    std::front_inserter(l), [](const BasicJsonType & i)
05022     {
05023         return i.template get<T>();
05024     });
05025 }
05026
05027 // valarray doesn't have an insert method
05028 template<typename BasicJsonType, typename T,
05029          enable_if_t<is_getable<BasicJsonType, T>::value, int> = 0>
05030 inline void from_json(const BasicJsonType& j, std::valarray<T>& l)
05031 {
05032     if (JSON_HEDLEY_UNLIKELY(!j.is_array()))
05033     {
05034         JSON_THROW(type_error::create(302, concat("type must be array, but is ", j.type_name()), &j));
05035     }
05036     l.resize(j.size());
05037     std::transform(j.begin(), j.end(), std::begin(l),
05038                    [](const BasicJsonType & elem)
05039     {
05040         return elem.template get<T>();
05041     });
05042 }
05043
05044 template<typename BasicJsonType, typename T, std::size_t N>
05045 auto from_json(const BasicJsonType& j, T (&arr)[N])  //
      NOLINT(cppcoreguidelines-avoid-c-arrays,hicpp-avoid-c-arrays,modernize-avoid-c-arrays)
05046 -> decltype(j.template get<T>(), void())
05047 {
05048     for (std::size_t i = 0; i < N; ++i)
05049     {
05050         arr[i] = j.at(i).template get<T>();
05051     }
05052 }
05053
05054 template<typename BasicJsonType, typename T, std::size_t N1, std::size_t N2>
05055 auto from_json(const BasicJsonType& j, T (&arr)[N1][N2])  //
      NOLINT(cppcoreguidelines-avoid-c-arrays,hicpp-avoid-c-arrays,modernize-avoid-c-arrays)
05056 -> decltype(j.template get<T>(), void())
05057 {
05058     for (std::size_t i1 = 0; i1 < N1; ++i1)
05059     {
05060         for (std::size_t i2 = 0; i2 < N2; ++i2)
05061         {
05062             arr[i1][i2] = j.at(i1).at(i2).template get<T>();
05063         }
05064     }
05065 }
05066
05067 template<typename BasicJsonType, typename T, std::size_t N1, std::size_t N2, std::size_t N3>
05068 auto from_json(const BasicJsonType& j, T (&arr)[N1][N2][N3])  //
      NOLINT(cppcoreguidelines-avoid-c-arrays,hicpp-avoid-c-arrays,modernize-avoid-c-arrays)
05069 -> decltype(j.template get<T>(), void())
05070 {
05071     for (std::size_t i1 = 0; i1 < N1; ++i1)
05072     {
```

```
05073         for (std::size_t i2 = 0; i2 < N2; ++i2)
05074         {
05075             for (std::size_t i3 = 0; i3 < N3; ++i3)
05076             {
05077                 arr[i1][i2][i3] = j.at(i1).at(i2).at(i3).template get<T>();
05078             }
05079         }
05080     }
05081 }
05082
05083 template<typename BasicJsonType, typename T, std::size_t N1, std::size_t N2, std::size_t N3,
      std::size_t N4>
05084 auto from_json(const BasicJsonType& j, T (&arr)[N1][N2][N3][N4])  //
      NOLINT(cppcoreguidelines-avoid-c-arrays,hicpp-avoid-c-arrays,modernize-avoid-c-arrays)
05085 -> decltype(j.template get<T>(), void())
05086 {
05087     for (std::size_t i1 = 0; i1 < N1; ++i1)
05088     {
05089         for (std::size_t i2 = 0; i2 < N2; ++i2)
05090         {
05091             for (std::size_t i3 = 0; i3 < N3; ++i3)
05092             {
05093                 for (std::size_t i4 = 0; i4 < N4; ++i4)
05094                 {
05095                     arr[i1][i2][i3][i4] = j.at(i1).at(i2).at(i3).at(i4).template get<T>();
05096                 }
05097             }
05098         }
05099     }
05100 }
05101
05102 template<typename BasicJsonType>
05103 inline void from_json_array_impl(const BasicJsonType& j, typename BasicJsonType::array_t& arr,
      priority_tag<3> /*unused*/)
05104 {
05105     arr = *j.template get_ptr<const typename BasicJsonType::array_t*>();
05106 }
05107
05108 template<typename BasicJsonType, typename T, std::size_t N>
05109 auto from_json_array_impl(const BasicJsonType& j, std::array<T, N>& arr,
05110                           priority_tag<2> /*unused*/)
05111 -> decltype(j.template get<T>(), void())
05112 {
05113     for (std::size_t i = 0; i < N; ++i)
05114     {
05115         arr[i] = j.at(i).template get<T>();
05116     }
05117 }
05118
05119 template<typename BasicJsonType, typename ConstructibleArrayType,
05120          enable_if_t<
05121              std::is_assignable<ConstructibleArrayType&, ConstructibleArrayType>::value,
05122              int> = 0>
05123 auto from_json_array_impl(const BasicJsonType& j, ConstructibleArrayType& arr, priority_tag<1>
      /*unused*/)
05124 -> decltype(
05125     arr.reserve(std::declval<typename ConstructibleArrayType::size_type>()),
05126     j.template get<typename ConstructibleArrayType::value_type>(),
05127     void())
05128 {
05129     using std::end;
05130
05131     ConstructibleArrayType ret;
05132     ret.reserve(j.size());
05133     std::transform(j.begin(), j.end(),
05134                    std::inserter(ret, end(ret)), [](const BasicJsonType & i)
05135     {
05136         // get<BasicJsonType>() returns *this, this won't call a from_json
05137         // method when value_type is BasicJsonType
05138         return i.template get<typename ConstructibleArrayType::value_type>();
05139     });
05140     arr = std::move(ret);
05141 }
05142
05143 template<typename BasicJsonType, typename ConstructibleArrayType,
05144          enable_if_t<
05145              std::is_assignable<ConstructibleArrayType&, ConstructibleArrayType>::value,
05146              int> = 0>
05147 inline void from_json_array_impl(const BasicJsonType& j, ConstructibleArrayType& arr,
05148                                  priority_tag<0> /*unused*/)
05149 {
05150     using std::end;
05151
05152     ConstructibleArrayType ret;
05153     std::transform(
05154         j.begin(), j.end(), std::inserter(ret, end(ret)),
05155         [](const BasicJsonType & i)
```

```
05156     {
05157         // get<BasicJsonType>() returns *this, this won't call a from_json
05158         // method when value_type is BasicJsonType
05159         return i.template get<typename ConstructibleArrayType::value_type>();
05160     });
05161     arr = std::move(ret);
05162 }
05163
05164 template < typename BasicJsonType, typename ConstructibleArrayType,
05165            enable_if_t <
05166                is_constructible_array_type<BasicJsonType, ConstructibleArrayType>::value&&
05167                !is_constructible_object_type<BasicJsonType, ConstructibleArrayType>::value&&
05168                !is_constructible_string_type<BasicJsonType, ConstructibleArrayType>::value&&
05169                !std::is_same<ConstructibleArrayType, typename BasicJsonType::binary_t>::value&&
05170                !is_basic_json<ConstructibleArrayType>::value,
05171                int > = 0 >
05172 auto from_json(const BasicJsonType& j, ConstructibleArrayType& arr)
05173 -> decltype(from_json_array_impl(j, arr, priority_tag<3> {}),
05174 j.template get<typename ConstructibleArrayType::value_type>(),
05175 void())
05176 {
05177     if (JSON_HEDLEY_UNLIKELY(!j.is_array()))
05178     {
05179         JSON_THROW(type_error::create(302, concat("type must be array, but is ", j.type_name()), &j));
05180     }
05181
05182     from_json_array_impl(j, arr, priority_tag<3> {});
05183 }
05184
05185 template < typename BasicJsonType, typename T, std::size_t... Idx >
05186 std::array<T, sizeof...(Idx)> from_json_inplace_array_impl(BasicJsonType&& j,
05187                       identity_tag<std::array<T, sizeof...(Idx)» /*unused*/, index_sequence<Idx...>
      /*unused*/)
05188 {
05189     return { { std::forward<BasicJsonType>(j).at(Idx).template get<T>()... } };
05190 }
05191
05192 template < typename BasicJsonType, typename T, std::size_t N >
05193 auto from_json(BasicJsonType&& j, identity_tag<std::array<T, N» tag)
05194 -> decltype(from_json_inplace_array_impl(std::forward<BasicJsonType>(j), tag, make_index_sequence<N>
      {}))
05195 {
05196     if (JSON_HEDLEY_UNLIKELY(!j.is_array()))
05197     {
05198         JSON_THROW(type_error::create(302, concat("type must be array, but is ", j.type_name()), &j));
05199     }
05200
05201     return from_json_inplace_array_impl(std::forward<BasicJsonType>(j), tag, make_index_sequence<N>
      {});
05202 }
05203
05204 template<typename BasicJsonType>
05205 inline void from_json(const BasicJsonType& j, typename BasicJsonType::binary_t& bin)
05206 {
05207     if (JSON_HEDLEY_UNLIKELY(!j.is_binary()))
05208     {
05209         JSON_THROW(type_error::create(302, concat("type must be binary, but is ", j.type_name()),
      &j));
05210     }
05211
05212     bin = *j.template get_ptr<const typename BasicJsonType::binary_t*>();
05213 }
05214
05215 template<typename BasicJsonType, typename ConstructibleObjectType,
05216          enable_if_t<is_constructible_object_type<BasicJsonType, ConstructibleObjectType>::value, int>
      = 0>
05217 inline void from_json(const BasicJsonType& j, ConstructibleObjectType& obj)
05218 {
05219     if (JSON_HEDLEY_UNLIKELY(!j.is_object()))
05220     {
05221         JSON_THROW(type_error::create(302, concat("type must be object, but is ", j.type_name()),
      &j));
05222     }
05223
05224     ConstructibleObjectType ret;
05225     const auto* inner_object = j.template get_ptr<const typename BasicJsonType::object_t*>();
05226     using value_type = typename ConstructibleObjectType::value_type;
05227     std::transform(
05228         inner_object->begin(), inner_object->end(),
05229         std::inserter(ret, ret.begin()),
05230         [](typename BasicJsonType::object_t::value_type const & p)
05231     {
05232         return value_type(p.first, p.second.template get<typename
      ConstructibleObjectType::mapped_type>());
05233     });
05234     obj = std::move(ret);
05235 }
```

```
05236
05237 // overload for arithmetic types, not chosen for basic_json template arguments
05238 // (BooleanType, etc.); note: Is it really necessary to provide explicit
05239 // overloads for boolean_t etc. in case of a custom BooleanType which is not
05240 // an arithmetic type?
05241 template < typename BasicJsonType, typename ArithmeticType,
05242            enable_if_t <
05243                std::is_arithmetic<ArithmeticType>::value&&
05244                !std::is_same<ArithmeticType, typename BasicJsonType::number_unsigned_t>::value&&
05245                !std::is_same<ArithmeticType, typename BasicJsonType::number_integer_t>::value&&
05246                !std::is_same<ArithmeticType, typename BasicJsonType::number_float_t>::value&&
05247                !std::is_same<ArithmeticType, typename BasicJsonType::boolean_t>::value,
05248                int > = 0 >
05249 inline void from_json(const BasicJsonType& j, ArithmeticType& val)
05250 {
05251     switch (static_cast<value_t>(j))
05252     {
05253         case value_t::number_unsigned:
05254         {
05255             val = static_cast<ArithmeticType>(*j.template get_ptr<const typename
      BasicJsonType::number_unsigned_t*>());
05256             break;
05257         }
05258         case value_t::number_integer:
05259         {
05260             val = static_cast<ArithmeticType>(*j.template get_ptr<const typename
      BasicJsonType::number_integer_t*>());
05261             break;
05262         }
05263         case value_t::number_float:
05264         {
05265             val = static_cast<ArithmeticType>(*j.template get_ptr<const typename
      BasicJsonType::number_float_t*>());
05266             break;
05267         }
05268         case value_t::boolean:
05269         {
05270             val = static_cast<ArithmeticType>(*j.template get_ptr<const typename
      BasicJsonType::boolean_t*>());
05271             break;
05272         }
05273
05274         case value_t::null:
05275         case value_t::object:
05276         case value_t::array:
05277         case value_t::string:
05278         case value_t::binary:
05279         case value_t::discarded:
05280         default:
05281             JSON_THROW(type_error::create(302, concat("type must be number, but is ", j.type_name()),
      &j));
05282     }
05283 }
05284
05285 template<typename BasicJsonType, typename... Args, std::size_t... Idx>
05286 std::tuple<Args...> from_json_tuple_impl_base(BasicJsonType&& j, index_sequence<Idx...> /*unused*/)
05287 {
05288     return std::make_tuple(std::forward<BasicJsonType>(j).at(Idx).template get<Args>()...);
05289 }
05290
05291 template<typename BasicJsonType>
05292 std::tuple<> from_json_tuple_impl_base(BasicJsonType& /*unused*/, index_sequence<> /*unused*/)
05293 {
05294     return {};
05295 }
05296
05297 template < typename BasicJsonType, class A1, class A2 >
05298 std::pair<A1, A2> from_json_tuple_impl(BasicJsonType&& j, identity_tag<std::pair<A1, A2>> /*unused*/,
      priority_tag<0> /*unused*/)
05299 {
05300     return {std::forward<BasicJsonType>(j).at(0).template get<A1>(),
05301             std::forward<BasicJsonType>(j).at(1).template get<A2>()};
05302 }
05303
05304 template<typename BasicJsonType, typename A1, typename A2>
05305 inline void from_json_tuple_impl(BasicJsonType&& j, std::pair<A1, A2>& p, priority_tag<1> /*unused*/)
05306 {
05307     p = from_json_tuple_impl(std::forward<BasicJsonType>(j), identity_tag<std::pair<A1, A2>> {},
      priority_tag<0> {});
05308 }
05309
05310 template<typename BasicJsonType, typename... Args>
05311 std::tuple<Args...> from_json_tuple_impl(BasicJsonType&& j, identity_tag<std::tuple<Args...>>
      /*unused*/, priority_tag<2> /*unused*/)
05312 {
05313     return from_json_tuple_impl_base<BasicJsonType, Args...>(std::forward<BasicJsonType>(j),
      index_sequence_for<Args...> {});
```

```
05314 }
05315
05316 template<typename BasicJsonType, typename... Args>
05317 inline void from_json_tuple_impl(BasicJsonType&& j, std::tuple<Args...>& t, priority_tag<3>
      /*unused*/)
05318 {
05319     t = from_json_tuple_impl_base<BasicJsonType, Args...>(std::forward<BasicJsonType>(j),
      index_sequence_for<Args...> {});
05320 }
05321
05322 template<typename BasicJsonType, typename TupleRelated>
05323 auto from_json(BasicJsonType&& j, TupleRelated&& t)
05324 -> decltype(from_json_tuple_impl(std::forward<BasicJsonType>(j), std::forward<TupleRelated>(t),
      priority_tag<3> {}))
05325 {
05326     if (JSON_HEDLEY_UNLIKELY(!j.is_array()))
05327     {
05328         JSON_THROW(type_error::create(302, concat("type must be array, but is ", j.type_name()), &j));
05329     }
05330
05331     return from_json_tuple_impl(std::forward<BasicJsonType>(j), std::forward<TupleRelated>(t),
      priority_tag<3> {});
05332 }
05333
05334 template < typename BasicJsonType, typename Key, typename Value, typename Compare, typename Allocator,
05335            typename = enable_if_t < !std::is_constructible <
05336                                         typename BasicJsonType::string_t, Key >::value >
05337 inline void from_json(const BasicJsonType& j, std::map<Key, Value, Compare, Allocator>& m)
05338 {
05339     if (JSON_HEDLEY_UNLIKELY(!j.is_array()))
05340     {
05341         JSON_THROW(type_error::create(302, concat("type must be array, but is ", j.type_name()), &j));
05342     }
05343     m.clear();
05344     for (const auto& p : j)
05345     {
05346         if (JSON_HEDLEY_UNLIKELY(!p.is_array()))
05347         {
05348             JSON_THROW(type_error::create(302, concat("type must be array, but is ", p.type_name()),
      &j));
05349         }
05350         m.emplace(p.at(0).template get<Key>(), p.at(1).template get<Value>());
05351     }
05352 }
05353
05354 template < typename BasicJsonType, typename Key, typename Value, typename Hash, typename KeyEqual,
      typename Allocator,
05355            typename = enable_if_t < !std::is_constructible <
05356                                         typename BasicJsonType::string_t, Key >::value >
05357 inline void from_json(const BasicJsonType& j, std::unordered_map<Key, Value, Hash, KeyEqual,
      Allocator>& m)
05358 {
05359     if (JSON_HEDLEY_UNLIKELY(!j.is_array()))
05360     {
05361         JSON_THROW(type_error::create(302, concat("type must be array, but is ", j.type_name()), &j));
05362     }
05363     m.clear();
05364     for (const auto& p : j)
05365     {
05366         if (JSON_HEDLEY_UNLIKELY(!p.is_array()))
05367         {
05368             JSON_THROW(type_error::create(302, concat("type must be array, but is ", p.type_name()),
      &j));
05369         }
05370         m.emplace(p.at(0).template get<Key>(), p.at(1).template get<Value>());
05371     }
05372 }
05373
05374 #if JSON_HAS_FILESYSTEM || JSON_HAS_EXPERIMENTAL_FILESYSTEM
05375 template<typename BasicJsonType>
05376 inline void from_json(const BasicJsonType& j, std_fs::path& p)
05377 {
05378     if (JSON_HEDLEY_UNLIKELY(!j.is_string()))
05379     {
05380         JSON_THROW(type_error::create(302, concat("type must be string, but is ", j.type_name()),
      &j));
05381     }
05382     const auto& s = *j.template get_ptr<const typename BasicJsonType::string_t*>();
05383     // Checking for C++20 standard or later can be insufficient in case the
05384     // library support for char8_t is either incomplete or was disabled
05385     // altogether. Use the __cpp_lib_char8_t feature test instead.
05386 #if defined(__cpp_lib_char8_t) && (__cpp_lib_char8_t >= 201907L)
05387     p = std_fs::path(std::u8string_view(reinterpret_cast<const char8_t*>(s.data()), s.size()));
05388 #else
05389     p = std_fs::u8path(s); // accepts UTF-8 encoded std::string in C++17, deprecated in C++20
05390 #endif
05391 }
```

```
05392 #endif
05393
05394 struct from_json_fn
05395 {
05396     template<typename BasicJsonType, typename T>
05397     auto operator()(const BasicJsonType& j, T&& val) const
05398     noexcept(noexcept(from_json(j, std::forward<T>(val))))
05399     -> decltype(from_json(j, std::forward<T>(val)))
05400     {
05401         return from_json(j, std::forward<T>(val));
05402     }
05403 };
05404
05405 }  // namespace detail
05406
05407 #ifndef JSON_HAS_CPP_17
05411 namespace // NOLINT(cert-dcl59-cpp,fuchsia-header-anon-namespaces,google-build-namespaces)
05412 {
05413 #endif
05414 JSON_INLINE_VARIABLE constexpr const auto& from_json = // NOLINT(misc-definitions-in-headers)
05415     detail::static_const<detail::from_json_fn>::value;
05416 #ifndef JSON_HAS_CPP_17
05417 }  // namespace
05418 #endif
05419
05420 NLOHMANN_JSON_NAMESPACE_END
05421
05422 // #include <nlohmann/detail/conversions/to_json.hpp>
05423 //     __ _____ _____ _____
05424 //  __|  |   __|     |   | |  JSON for Modern C++
05425 // |  |  |__   |  |  | | | |  version 3.12.0
05426 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
05427 //
05428 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
05429 // SPDX-License-Identifier: MIT
05430
05431
05432
05433 // #include <nlohmann/detail/macro_scope.hpp>
05434 // JSON_HAS_CPP_17
05435 #ifdef JSON_HAS_CPP_17
05436     #include <optional> // optional
05437 #endif
05438
05439 #include <algorithm> // copy
05440 #include <iterator> // begin, end
05441 #include <memory> // allocator_traits
05442 #include <string> //  basic_string, char_traits
05443 #include <tuple> // tuple, get
05444 #include <type_traits> // is_same, is_constructible, is_floating_point, is_enum, underlying_type
05445 #include <utility> // move, forward, declval, pair
05446 #include <valarray> // valarray
05447 #include <vector> // vector
05448
05449 // #include <nlohmann/detail/iterators/iteration_proxy.hpp>
05450 //     __ _____ _____ _____
05451 //  __|  |   __|     |   | |  JSON for Modern C++
05452 // |  |  |__   |  |  | | | |  version 3.12.0
05453 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
05454 //
05455 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
05456 // SPDX-License-Identifier: MIT
05457
05458
05459
05460 #include <cstddef> // size_t
05461 #include <iterator> // forward_iterator_tag
05462 #include <tuple> // tuple_size, get, tuple_element
05463 #include <utility> // move
05464
05465 #if JSON_HAS_RANGES
05466     #include <ranges> // enable_borrowed_range
05467 #endif
05468
05469 // #include <nlohmann/detail/abi_macros.hpp>
05470
05471 // #include <nlohmann/detail/meta/type_traits.hpp>
05472
05473 // #include <nlohmann/detail/string_utils.hpp>
05474 //     __ _____ _____ _____
05475 //  __|  |   __|     |   | |  JSON for Modern C++
05476 // |  |  |__   |  |  | | | |  version 3.12.0
05477 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
05478 //
05479 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
05480 // SPDX-License-Identifier: MIT
05481
```

```
05482
05483
05484 #include <cstddef> // size_t
05485 #include <string> // string, to_string
05486
05487 // #include <nlohmann/detail/abi_macros.hpp>
05488
05489
05490 NLOHMANN_JSON_NAMESPACE_BEGIN
05491 namespace detail
05492 {
05493
05494 template<typename StringType>
05495 void int_to_string(StringType& target, std::size_t value)
05496 {
05497     // For ADL
05498     using std::to_string;
05499     target = to_string(value);
05500 }
05501
05502 template<typename StringType>
05503 StringType to_string(std::size_t value)
05504 {
05505     StringType result;
05506     int_to_string(result, value);
05507     return result;
05508 }
05509
05510 }  // namespace detail
05511 NLOHMANN_JSON_NAMESPACE_END
05512
05513 // #include <nlohmann/detail/value_t.hpp>
05514
05515
05516 NLOHMANN_JSON_NAMESPACE_BEGIN
05517 namespace detail
05518 {
05519
05520 template<typename IteratorType> class iteration_proxy_value
05521 {
05522   public:
05523     using difference_type = std::ptrdiff_t;
05524     using value_type = iteration_proxy_value;
05525     using pointer = value_type *;
05526     using reference = value_type &;
05527     using iterator_category = std::forward_iterator_tag;
05528     using string_type = typename std::remove_cv< typename std::remove_reference<decltype(
      std::declval<IteratorType>().key() ) >::type >::type;
05529
05530   private:
05532     IteratorType anchor{};
05534     std::size_t array_index = 0;
05536     mutable std::size_t array_index_last = 0;
05538     mutable string_type array_index_str = "0";
05540     string_type empty_str{};
05541
05542   public:
05543     explicit iteration_proxy_value() = default;
05544     explicit iteration_proxy_value(IteratorType it, std::size_t array_index_ = 0)
05545     noexcept(std::is_nothrow_move_constructible<IteratorType>::value
05546              && std::is_nothrow_default_constructible<string_type>::value)
05547         : anchor(std::move(it))
05548         , array_index(array_index_)
05549     {}
05550
05551     iteration_proxy_value(iteration_proxy_value const&) = default;
05552     iteration_proxy_value& operator=(iteration_proxy_value const&) = default;
05553     // older GCCs are a bit fussy and require explicit noexcept specifiers on defaulted functions
05554     iteration_proxy_value(iteration_proxy_value&&)
05555     noexcept(std::is_nothrow_move_constructible<IteratorType>::value
05556              && std::is_nothrow_move_constructible<string_type>::value) = default; //
      NOLINT(hicpp-noexcept-move,performance-noexcept-move-constructor,cppcoreguidelines-noexcept-move-operations)
05557     iteration_proxy_value& operator=(iteration_proxy_value&&)
05558     noexcept(std::is_nothrow_move_assignable<IteratorType>::value
05559              && std::is_nothrow_move_assignable<string_type>::value) = default; //
      NOLINT(hicpp-noexcept-move,performance-noexcept-move-constructor,cppcoreguidelines-noexcept-move-operations)
05560     ~iteration_proxy_value() = default;
05561
05563     const iteration_proxy_value& operator*() const
05564     {
05565         return *this;
05566     }
05567
05569     iteration_proxy_value& operator++()
05570     {
05571         ++anchor;
05572         ++array_index;
```

```
05573
05574         return *this;
05575     }
05576
05577     iteration_proxy_value operator++(int)& // NOLINT(cert-dcl21-cpp)
05578     {
05579         auto tmp = iteration_proxy_value(anchor, array_index);
05580         ++anchor;
05581         ++array_index;
05582         return tmp;
05583     }
05584
05586     bool operator==(const iteration_proxy_value& o) const
05587     {
05588         return anchor == o.anchor;
05589     }
05590
05592     bool operator!=(const iteration_proxy_value& o) const
05593     {
05594         return anchor != o.anchor;
05595     }
05596
05598     const string_type& key() const
05599     {
05600         JSON_ASSERT(anchor.m_object != nullptr);
05601
05602         switch (anchor.m_object->type())
05603         {
05604             // use integer array index as key
05605             case value_t::array:
05606             {
05607                 if (array_index != array_index_last)
05608                 {
05609                     int_to_string( array_index_str, array_index );
05610                     array_index_last = array_index;
05611                 }
05612                 return array_index_str;
05613             }
05614
05615             // use key from the object
05616             case value_t::object:
05617                 return anchor.key();
05618
05619             // use an empty key for all primitive types
05620             case value_t::null:
05621             case value_t::string:
05622             case value_t::boolean:
05623             case value_t::number_integer:
05624             case value_t::number_unsigned:
05625             case value_t::number_float:
05626             case value_t::binary:
05627             case value_t::discarded:
05628             default:
05629                 return empty_str;
05630         }
05631     }
05632
05634     typename IteratorType::reference value() const
05635     {
05636         return anchor.value();
05637     }
05638 };
05639
05641 template<typename IteratorType> class iteration_proxy
05642 {
05643   private:
05645     typename IteratorType::pointer container = nullptr;
05646
05647   public:
05648     explicit iteration_proxy() = default;
05649
05651     explicit iteration_proxy(typename IteratorType::reference cont) noexcept
05652         : container(&cont) {}
05653
05654     iteration_proxy(iteration_proxy const&) = default;
05655     iteration_proxy& operator=(iteration_proxy const&) = default;
05656     iteration_proxy(iteration_proxy&&) noexcept = default;
05657     iteration_proxy& operator=(iteration_proxy&&) noexcept = default;
05658     ~iteration_proxy() = default;
05659
05661     iteration_proxy_value<IteratorType> begin() const noexcept
05662     {
05663         return iteration_proxy_value<IteratorType>(container->begin());
05664     }
05665
05667     iteration_proxy_value<IteratorType> end() const noexcept
05668     {
```

```
05669         return iteration_proxy_value<IteratorType>(container->end());
05670     }
05671 };
05672
05673 // Structured Bindings Support
05674 // For further reference see https://blog.tartanllama.xyz/structured-bindings/
05675 // And see https://github.com/nlohmann/json/pull/1391
05676 template<std::size_t N, typename IteratorType, enable_if_t<N == 0, int> = 0>
05677 auto get(const nlohmann::detail::iteration_proxy_value<IteratorType>& i) -> decltype(i.key())
05678 {
05679     return i.key();
05680 }
05681 // Structured Bindings Support
05682 // For further reference see https://blog.tartanllama.xyz/structured-bindings/
05683 // And see https://github.com/nlohmann/json/pull/1391
05684 template<std::size_t N, typename IteratorType, enable_if_t<N == 1, int> = 0>
05685 auto get(const nlohmann::detail::iteration_proxy_value<IteratorType>& i) -> decltype(i.value())
05686 {
05687     return i.value();
05688 }
05689
05690 }  // namespace detail
05691 NLOHMANN_JSON_NAMESPACE_END
05692
05693 // The Addition to the STD Namespace is required to add
05694 // Structured Bindings Support to the iteration_proxy_value class
05695 // For further reference see https://blog.tartanllama.xyz/structured-bindings/
05696 // And see https://github.com/nlohmann/json/pull/1391
05697 namespace std
05698 {
05699
05700 #if defined(__clang__)
05701     // Fix: https://github.com/nlohmann/json/issues/1401
05702     #pragma clang diagnostic push
05703     #pragma clang diagnostic ignored "-Wmismatched-tags"
05704 #endif
05705 template<typename IteratorType>
05706 class tuple_size<::nlohmann::detail::iteration_proxy_value<IteratorType» // NOLINT(cert-dcl58-cpp)
05707     : public std::integral_constant<std::size_t, 2> {};
05708
05709 template<std::size_t N, typename IteratorType>
05710 class tuple_element<N, ::nlohmann::detail::iteration_proxy_value<IteratorType » //
      NOLINT(cert-dcl58-cpp)
05711 {
05712   public:
05713     using type = decltype(
05714                       get<N>(std::declval <
05715                              ::nlohmann::detail::iteration_proxy_value<IteratorType » ()));
05716 };
05717 #if defined(__clang__)
05718     #pragma clang diagnostic pop
05719 #endif
05720
05721 }  // namespace std
05722
05723 #if JSON_HAS_RANGES
05724     template <typename IteratorType>
05725     inline constexpr bool
      ::std::ranges::enable_borrowed_range<::nlohmann::detail::iteration_proxy<IteratorType» = true;
05726 #endif
05727
05728 // #include <nlohmann/detail/meta/cpp_future.hpp>
05729
05730 // #include <nlohmann/detail/meta/std_fs.hpp>
05731
05732 // #include <nlohmann/detail/meta/type_traits.hpp>
05733
05734 // #include <nlohmann/detail/value_t.hpp>
05735
05736
05737 NLOHMANN_JSON_NAMESPACE_BEGIN
05738 namespace detail
05739 {
05740
05742 // constructors //
05744
05745 /*
05746  * Note all external_constructor<>::construct functions need to call
05747  * j.m_data.m_value.destroy(j.m_data.m_type) to avoid a memory leak in case j contains an
05748  * allocated value (e.g., a string). See bug issue
05749  * https://github.com/nlohmann/json/issues/2865 for more information.
05750  */
05751
05752 template<value_t> struct external_constructor;
05753
05754 template<>
05755 struct external_constructor<value_t::boolean>
```

```
05756 {
05757     template<typename BasicJsonType>
05758     static void construct(BasicJsonType& j, typename BasicJsonType::boolean_t b) noexcept
05759     {
05760         j.m_data.m_value.destroy(j.m_data.m_type);
05761         j.m_data.m_type = value_t::boolean;
05762         j.m_data.m_value = b;
05763         j.assert_invariant();
05764     }
05765 };
05766
05767 template<>
05768 struct external_constructor<value_t::string>
05769 {
05770     template<typename BasicJsonType>
05771     static void construct(BasicJsonType& j, const typename BasicJsonType::string_t& s)
05772     {
05773         j.m_data.m_value.destroy(j.m_data.m_type);
05774         j.m_data.m_type = value_t::string;
05775         j.m_data.m_value = s;
05776         j.assert_invariant();
05777     }
05778
05779     template<typename BasicJsonType>
05780     static void construct(BasicJsonType& j, typename BasicJsonType::string_t&& s)
05781     {
05782         j.m_data.m_value.destroy(j.m_data.m_type);
05783         j.m_data.m_type = value_t::string;
05784         j.m_data.m_value = std::move(s);
05785         j.assert_invariant();
05786     }
05787
05788     template < typename BasicJsonType, typename CompatibleStringType,
05789                enable_if_t < !std::is_same<CompatibleStringType, typename
      BasicJsonType::string_t>::value,
05790                              int > = 0 >
05791     static void construct(BasicJsonType& j, const CompatibleStringType& str)
05792     {
05793         j.m_data.m_value.destroy(j.m_data.m_type);
05794         j.m_data.m_type = value_t::string;
05795         j.m_data.m_value.string = j.template create<typename BasicJsonType::string_t>(str);
05796         j.assert_invariant();
05797     }
05798 };
05799
05800 template<>
05801 struct external_constructor<value_t::binary>
05802 {
05803     template<typename BasicJsonType>
05804     static void construct(BasicJsonType& j, const typename BasicJsonType::binary_t& b)
05805     {
05806         j.m_data.m_value.destroy(j.m_data.m_type);
05807         j.m_data.m_type = value_t::binary;
05808         j.m_data.m_value = typename BasicJsonType::binary_t(b);
05809         j.assert_invariant();
05810     }
05811
05812     template<typename BasicJsonType>
05813     static void construct(BasicJsonType& j, typename BasicJsonType::binary_t&& b)
05814     {
05815         j.m_data.m_value.destroy(j.m_data.m_type);
05816         j.m_data.m_type = value_t::binary;
05817         j.m_data.m_value = typename BasicJsonType::binary_t(std::move(b));
05818         j.assert_invariant();
05819     }
05820 };
05821
05822 template<>
05823 struct external_constructor<value_t::number_float>
05824 {
05825     template<typename BasicJsonType>
05826     static void construct(BasicJsonType& j, typename BasicJsonType::number_float_t val) noexcept
05827     {
05828         j.m_data.m_value.destroy(j.m_data.m_type);
05829         j.m_data.m_type = value_t::number_float;
05830         j.m_data.m_value = val;
05831         j.assert_invariant();
05832     }
05833 };
05834
05835 template<>
05836 struct external_constructor<value_t::number_unsigned>
05837 {
05838     template<typename BasicJsonType>
05839     static void construct(BasicJsonType& j, typename BasicJsonType::number_unsigned_t val) noexcept
05840     {
05841         j.m_data.m_value.destroy(j.m_data.m_type);
```

```
05842         j.m_data.m_type = value_t::number_unsigned;
05843         j.m_data.m_value = val;
05844         j.assert_invariant();
05845     }
05846 };
05847
05848 template<>
05849 struct external_constructor<value_t::number_integer>
05850 {
05851     template<typename BasicJsonType>
05852     static void construct(BasicJsonType& j, typename BasicJsonType::number_integer_t val) noexcept
05853     {
05854         j.m_data.m_value.destroy(j.m_data.m_type);
05855         j.m_data.m_type = value_t::number_integer;
05856         j.m_data.m_value = val;
05857         j.assert_invariant();
05858     }
05859 };
05860
05861 template<>
05862 struct external_constructor<value_t::array>
05863 {
05864     template<typename BasicJsonType>
05865     static void construct(BasicJsonType& j, const typename BasicJsonType::array_t& arr)
05866     {
05867         j.m_data.m_value.destroy(j.m_data.m_type);
05868         j.m_data.m_type = value_t::array;
05869         j.m_data.m_value = arr;
05870         j.set_parents();
05871         j.assert_invariant();
05872     }
05873
05874     template<typename BasicJsonType>
05875     static void construct(BasicJsonType& j, typename BasicJsonType::array_t&& arr)
05876     {
05877         j.m_data.m_value.destroy(j.m_data.m_type);
05878         j.m_data.m_type = value_t::array;
05879         j.m_data.m_value = std::move(arr);
05880         j.set_parents();
05881         j.assert_invariant();
05882     }
05883
05884     template < typename BasicJsonType, typename CompatibleArrayType,
05885                enable_if_t < !std::is_same<CompatibleArrayType, typename
      BasicJsonType::array_t>::value,
05886                              int > = 0 >
05887     static void construct(BasicJsonType& j, const CompatibleArrayType& arr)
05888     {
05889         using std::begin;
05890         using std::end;
05891
05892         j.m_data.m_value.destroy(j.m_data.m_type);
05893         j.m_data.m_type = value_t::array;
05894         j.m_data.m_value.array = j.template create<typename BasicJsonType::array_t>(begin(arr),
      end(arr));
05895         j.set_parents();
05896         j.assert_invariant();
05897     }
05898
05899     template<typename BasicJsonType>
05900     static void construct(BasicJsonType& j, const std::vector<bool>& arr)
05901     {
05902         j.m_data.m_value.destroy(j.m_data.m_type);
05903         j.m_data.m_type = value_t::array;
05904         j.m_data.m_value = value_t::array;
05905         j.m_data.m_value.array->reserve(arr.size());
05906         for (const bool x : arr)
05907         {
05908             j.m_data.m_value.array->push_back(x);
05909             j.set_parent(j.m_data.m_value.array->back());
05910         }
05911         j.assert_invariant();
05912     }
05913
05914     template<typename BasicJsonType, typename T,
05915              enable_if_t<std::is_convertible<T, BasicJsonType>::value, int> = 0>
05916     static void construct(BasicJsonType& j, const std::valarray<T>& arr)
05917     {
05918         j.m_data.m_value.destroy(j.m_data.m_type);
05919         j.m_data.m_type = value_t::array;
05920         j.m_data.m_value = value_t::array;
05921         j.m_data.m_value.array->resize(arr.size());
05922         if (arr.size() > 0)
05923         {
05924             std::copy(std::begin(arr), std::end(arr), j.m_data.m_value.array->begin());
05925         }
05926         j.set_parents();
```

```
05927         j.assert_invariant();
05928     }
05929 };
05930
05931 template<>
05932 struct external_constructor<value_t::object>
05933 {
05934     template<typename BasicJsonType>
05935     static void construct(BasicJsonType& j, const typename BasicJsonType::object_t& obj)
05936     {
05937         j.m_data.m_value.destroy(j.m_data.m_type);
05938         j.m_data.m_type = value_t::object;
05939         j.m_data.m_value = obj;
05940         j.set_parents();
05941         j.assert_invariant();
05942     }
05943
05944     template<typename BasicJsonType>
05945     static void construct(BasicJsonType& j, typename BasicJsonType::object_t&& obj)
05946     {
05947         j.m_data.m_value.destroy(j.m_data.m_type);
05948         j.m_data.m_type = value_t::object;
05949         j.m_data.m_value = std::move(obj);
05950         j.set_parents();
05951         j.assert_invariant();
05952     }
05953
05954     template < typename BasicJsonType, typename CompatibleObjectType,
05955                enable_if_t < !std::is_same<CompatibleObjectType, typename
      BasicJsonType::object_t>::value, int > = 0 >
05956     static void construct(BasicJsonType& j, const CompatibleObjectType& obj)
05957     {
05958         using std::begin;
05959         using std::end;
05960
05961         j.m_data.m_value.destroy(j.m_data.m_type);
05962         j.m_data.m_type = value_t::object;
05963         j.m_data.m_value.object = j.template create<typename BasicJsonType::object_t>(begin(obj),
      end(obj));
05964         j.set_parents();
05965         j.assert_invariant();
05966     }
05967 };
05968
05970 // to_json //
05972
05973 #ifdef JSON_HAS_CPP_17
05974 template<typename BasicJsonType, typename T,
05975          enable_if_t<std::is_constructible<BasicJsonType, T>::value, int> = 0>
05976 void to_json(BasicJsonType& j, const std::optional<T>& opt) noexcept
05977 {
05978     if (opt.has_value())
05979     {
05980         j = *opt;
05981     }
05982     else
05983     {
05984         j = nullptr;
05985     }
05986 }
05987 #endif
05988
05989 template<typename BasicJsonType, typename T,
05990          enable_if_t<std::is_same<T, typename BasicJsonType::boolean_t>::value, int> = 0>
05991 inline void to_json(BasicJsonType& j, T b) noexcept
05992 {
05993     external_constructor<value_t::boolean>::construct(j, b);
05994 }
05995
05996 template < typename BasicJsonType, typename BoolRef,
05997            enable_if_t <
05998                ((std::is_same<std::vector<bool>::reference, BoolRef>::value
05999                  && !std::is_same <std::vector<bool>::reference, typename
      BasicJsonType::boolean_t&>::value)
06000                 || (std::is_same<std::vector<bool>::const_reference, BoolRef>::value
06001                     && !std::is_same <detail::uncvref_t<std::vector<bool>::const_reference>,
06002                                       typename BasicJsonType::boolean_t >::value))
06003                && std::is_convertible<const BoolRef&, typename BasicJsonType::boolean_t>::value, int >
      = 0 >
06004 inline void to_json(BasicJsonType& j, const BoolRef& b) noexcept
06005 {
06006     external_constructor<value_t::boolean>::construct(j, static_cast<typename
      BasicJsonType::boolean_t>(b));
06007 }
06008
06009 template<typename BasicJsonType, typename CompatibleString,
06010          enable_if_t<std::is_constructible<typename BasicJsonType::string_t, CompatibleString>::value,
```

```
      int> = 0>
06011 inline void to_json(BasicJsonType& j, const CompatibleString& s)
06012 {
06013     external_constructor<value_t::string>::construct(j, s);
06014 }
06015
06016 template<typename BasicJsonType>
06017 inline void to_json(BasicJsonType& j, typename BasicJsonType::string_t&& s)
06018 {
06019     external_constructor<value_t::string>::construct(j, std::move(s));
06020 }
06021
06022 template<typename BasicJsonType, typename FloatType,
06023          enable_if_t<std::is_floating_point<FloatType>::value, int> = 0>
06024 inline void to_json(BasicJsonType& j, FloatType val) noexcept
06025 {
06026     external_constructor<value_t::number_float>::construct(j, static_cast<typename
      BasicJsonType::number_float_t>(val));
06027 }
06028
06029 template<typename BasicJsonType, typename CompatibleNumberUnsignedType,
06030          enable_if_t<is_compatible_integer_type<typename BasicJsonType::number_unsigned_t,
      CompatibleNumberUnsignedType>::value, int> = 0>
06031 inline void to_json(BasicJsonType& j, CompatibleNumberUnsignedType val) noexcept
06032 {
06033     external_constructor<value_t::number_unsigned>::construct(j, static_cast<typename
      BasicJsonType::number_unsigned_t>(val));
06034 }
06035
06036 template<typename BasicJsonType, typename CompatibleNumberIntegerType,
06037          enable_if_t<is_compatible_integer_type<typename BasicJsonType::number_integer_t,
      CompatibleNumberIntegerType>::value, int> = 0>
06038 inline void to_json(BasicJsonType& j, CompatibleNumberIntegerType val) noexcept
06039 {
06040     external_constructor<value_t::number_integer>::construct(j, static_cast<typename
      BasicJsonType::number_integer_t>(val));
06041 }
06042
06043 #if !JSON_DISABLE_ENUM_SERIALIZATION
06044 template<typename BasicJsonType, typename EnumType,
06045          enable_if_t<std::is_enum<EnumType>::value, int> = 0>
06046 inline void to_json(BasicJsonType& j, EnumType e) noexcept
06047 {
06048     using underlying_type = typename std::underlying_type<EnumType>::type;
06049     static constexpr value_t integral_value_t = std::is_unsigned<underlying_type>::value ?
      value_t::number_unsigned : value_t::number_integer;
06050     external_constructor<integral_value_t>::construct(j, static_cast<underlying_type>(e));
06051 }
06052 #endif  // JSON_DISABLE_ENUM_SERIALIZATION
06053
06054 template<typename BasicJsonType>
06055 inline void to_json(BasicJsonType& j, const std::vector<bool>& e)
06056 {
06057     external_constructor<value_t::array>::construct(j, e);
06058 }
06059
06060 template < typename BasicJsonType, typename CompatibleArrayType,
06061            enable_if_t < is_compatible_array_type<BasicJsonType,
06062                          CompatibleArrayType>::value&&
06063                          !is_compatible_object_type<BasicJsonType, CompatibleArrayType>::value&&
06064                          !is_compatible_string_type<BasicJsonType, CompatibleArrayType>::value&&
06065                          !std::is_same<typename BasicJsonType::binary_t, CompatibleArrayType>::value&&
06066                          !is_basic_json<CompatibleArrayType>::value,
06067                          int > = 0 >
06068 inline void to_json(BasicJsonType& j, const CompatibleArrayType& arr)
06069 {
06070     external_constructor<value_t::array>::construct(j, arr);
06071 }
06072
06073 template<typename BasicJsonType>
06074 inline void to_json(BasicJsonType& j, const typename BasicJsonType::binary_t& bin)
06075 {
06076     external_constructor<value_t::binary>::construct(j, bin);
06077 }
06078
06079 template<typename BasicJsonType, typename T,
06080          enable_if_t<std::is_convertible<T, BasicJsonType>::value, int> = 0>
06081 inline void to_json(BasicJsonType& j, const std::valarray<T>& arr)
06082 {
06083     external_constructor<value_t::array>::construct(j, std::move(arr));
06084 }
06085
06086 template<typename BasicJsonType>
06087 inline void to_json(BasicJsonType& j, typename BasicJsonType::array_t&& arr)
06088 {
06089     external_constructor<value_t::array>::construct(j, std::move(arr));
06090 }
```

```
06091
06092 template < typename BasicJsonType, typename CompatibleObjectType,
06093            enable_if_t < is_compatible_object_type<BasicJsonType, CompatibleObjectType>::value&&
      !is_basic_json<CompatibleObjectType>::value, int > = 0 >
06094 inline void to_json(BasicJsonType& j, const CompatibleObjectType& obj)
06095 {
06096     external_constructor<value_t::object>::construct(j, obj);
06097 }
06098
06099 template<typename BasicJsonType>
06100 inline void to_json(BasicJsonType& j, typename BasicJsonType::object_t&& obj)
06101 {
06102     external_constructor<value_t::object>::construct(j, std::move(obj));
06103 }
06104
06105 template <
06106     typename BasicJsonType, typename T, std::size_t N,
06107     enable_if_t < !std::is_constructible<typename BasicJsonType::string_t,
06108                   const T(&)[N]>::value, //
      NOLINT(cppcoreguidelines-avoid-c-arrays,hicpp-avoid-c-arrays,modernize-avoid-c-arrays)
06109                   int > = 0 >
06110 inline void to_json(BasicJsonType& j, const T(&arr)[N]) //
      NOLINT(cppcoreguidelines-avoid-c-arrays,hicpp-avoid-c-arrays,modernize-avoid-c-arrays)
06111 {
06112     external_constructor<value_t::array>::construct(j, arr);
06113 }
06114
06115 template < typename BasicJsonType, typename T1, typename T2, enable_if_t <
      std::is_constructible<BasicJsonType, T1>::value&& std::is_constructible<BasicJsonType, T2>::value, int
      > = 0 >
06116 inline void to_json(BasicJsonType& j, const std::pair<T1, T2>& p)
06117 {
06118     j = { p.first, p.second };
06119 }
06120
06121 // for https://github.com/nlohmann/json/pull/1134
06122 template<typename BasicJsonType, typename T,
06123          enable_if_t<std::is_same<T, iteration_proxy_value<typename BasicJsonType::iterator>>::value,
      int> = 0>
06124 inline void to_json(BasicJsonType& j, const T& b)
06125 {
06126     j = { {b.key(), b.value()} };
06127 }
06128
06129 template<typename BasicJsonType, typename Tuple, std::size_t... Idx>
06130 inline void to_json_tuple_impl(BasicJsonType& j, const Tuple& t, index_sequence<Idx...> /*unused*/)
06131 {
06132     j = { std::get<Idx>(t)... };
06133 }
06134
06135 template<typename BasicJsonType, typename Tuple>
06136 inline void to_json_tuple_impl(BasicJsonType& j, const Tuple& /*unused*/, index_sequence<> /*unused*/)
06137 {
06138     using array_t = typename BasicJsonType::array_t;
06139     j = array_t();
06140 }
06141
06142 template<typename BasicJsonType, typename T, enable_if_t<is_constructible_tuple<BasicJsonType,
      T>::value, int > = 0>
06143 inline void to_json(BasicJsonType& j, const T& t)
06144 {
06145     to_json_tuple_impl(j, t, make_index_sequence<std::tuple_size<T>::value> {});
06146 }
06147
06148 #if JSON_HAS_FILESYSTEM || JSON_HAS_EXPERIMENTAL_FILESYSTEM
06149 #if defined(__cpp_lib_char8_t)
06150 template<typename BasicJsonType, typename Tr, typename Allocator>
06151 inline void to_json(BasicJsonType& j, const std::basic_string<char8_t, Tr, Allocator>& s)
06152 {
06153     using OtherAllocator = typename std::allocator_traits<Allocator>::template rebind_alloc<char>;
06154     j = std::basic_string<char, std::char_traits<char>, OtherAllocator>(s.begin(), s.end(),
      s.get_allocator());
06155 }
06156 #endif
06157
06158 template<typename BasicJsonType>
06159 inline void to_json(BasicJsonType& j, const std_fs::path& p)
06160 {
06161     // Returns either a std::string or a std::u8string depending whether library
06162     // support for char8_t is enabled.
06163     j = p.u8string();
06164 }
06165 #endif
06166
06167 struct to_json_fn
06168 {
06169     template<typename BasicJsonType, typename T>
```

```
06170     auto operator()(BasicJsonType& j, T&& val) const noexcept(noexcept(to_json(j,
      std::forward<T>(val))))
06171     -> decltype(to_json(j, std::forward<T>(val)), void())
06172     {
06173         return to_json(j, std::forward<T>(val));
06174     }
06175 };
06176 }  // namespace detail
06177
06178 #ifndef JSON_HAS_CPP_17
06182 namespace // NOLINT(cert-dcl59-cpp,fuchsia-header-anon-namespaces,google-build-namespaces)
06183 {
06184 #endif
06185 JSON_INLINE_VARIABLE constexpr const auto& to_json = // NOLINT(misc-definitions-in-headers)
06186     detail::static_const<detail::to_json_fn>::value;
06187 #ifndef JSON_HAS_CPP_17
06188 }  // namespace
06189 #endif
06190
06191 NLOHMANN_JSON_NAMESPACE_END
06192
06193 // #include <nlohmann/detail/meta/identity_tag.hpp>
06194
06195
06196 NLOHMANN_JSON_NAMESPACE_BEGIN
06197
06199 template<typename ValueType, typename>
06200 struct adl_serializer
06201 {
06204     template<typename BasicJsonType, typename TargetType = ValueType>
06205     static auto from_json(BasicJsonType && j, TargetType& val) noexcept(
06206         noexcept(::nlohmann::from_json(std::forward<BasicJsonType>(j), val)))
06207     -> decltype(::nlohmann::from_json(std::forward<BasicJsonType>(j), val), void())
06208     {
06209         ::nlohmann::from_json(std::forward<BasicJsonType>(j), val);
06210     }
06211
06214     template<typename BasicJsonType, typename TargetType = ValueType>
06215     static auto from_json(BasicJsonType && j) noexcept(
06216     noexcept(::nlohmann::from_json(std::forward<BasicJsonType>(j), detail::identity_tag<TargetType>
      {})))
06217     -> decltype(::nlohmann::from_json(std::forward<BasicJsonType>(j), detail::identity_tag<TargetType>
      {}))
06218     {
06219         return ::nlohmann::from_json(std::forward<BasicJsonType>(j), detail::identity_tag<TargetType>
      {});
06220     }
06221
06224     template<typename BasicJsonType, typename TargetType = ValueType>
06225     static auto to_json(BasicJsonType& j, TargetType && val) noexcept(
06226         noexcept(::nlohmann::to_json(j, std::forward<TargetType>(val))))
06227     -> decltype(::nlohmann::to_json(j, std::forward<TargetType>(val)), void())
06228     {
06229         ::nlohmann::to_json(j, std::forward<TargetType>(val));
06230     }
06231 };
06232
06233 NLOHMANN_JSON_NAMESPACE_END
06234
06235 // #include <nlohmann/byte_container_with_subtype.hpp>
06236 //     __ _____ _____ _____
06237 //  __|  |   __|     |   | |  JSON for Modern C++
06238 // |  |  |__   |  |  | | | |  version 3.12.0
06239 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
06240 //
06241 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
06242 // SPDX-License-Identifier: MIT
06243
06244
06245
06246 #include <cstdint> // uint8_t, uint64_t
06247 #include <tuple> // tie
06248 #include <utility> // move
06249
06250 // #include <nlohmann/detail/abi_macros.hpp>
06251
06252
06253 NLOHMANN_JSON_NAMESPACE_BEGIN
06254
06257 template<typename BinaryType>
06258 class byte_container_with_subtype : public BinaryType
06259 {
06260   public:
06261     using container_type = BinaryType;
06262     using subtype_type = std::uint64_t;
06263
06265     byte_container_with_subtype() noexcept(noexcept(container_type()))
```

```
06266         : container_type()
06267     {}
06268
06270     byte_container_with_subtype(const container_type& b) noexcept(noexcept(container_type(b)))
06271         : container_type(b)
06272     {}
06273
06275     byte_container_with_subtype(container_type&& b) noexcept(noexcept(container_type(std::move(b))))
06276         : container_type(std::move(b))
06277     {}
06278
06280     byte_container_with_subtype(const container_type& b, subtype_type subtype_)
      noexcept(noexcept(container_type(b)))
06281         : container_type(b)
06282         , m_subtype(subtype_)
06283         , m_has_subtype(true)
06284     {}
06285
06287     byte_container_with_subtype(container_type&& b, subtype_type subtype_)
      noexcept(noexcept(container_type(std::move(b))))
06288         : container_type(std::move(b))
06289         , m_subtype(subtype_)
06290         , m_has_subtype(true)
06291     {}
06292
06293     bool operator==(const byte_container_with_subtype& rhs) const
06294     {
06295         return std::tie(static_cast<const BinaryType&>(*this), m_subtype, m_has_subtype) ==
06296                std::tie(static_cast<const BinaryType&>(rhs), rhs.m_subtype, rhs.m_has_subtype);
06297     }
06298
06299     bool operator!=(const byte_container_with_subtype& rhs) const
06300     {
06301         return !(rhs == *this);
06302     }
06303
06306     void set_subtype(subtype_type subtype_) noexcept
06307     {
06308         m_subtype = subtype_;
06309         m_has_subtype = true;
06310     }
06311
06314     constexpr subtype_type subtype() const noexcept
06315     {
06316         return m_has_subtype ? m_subtype : static_cast<subtype_type>(-1);
06317     }
06318
06321     constexpr bool has_subtype() const noexcept
06322     {
06323         return m_has_subtype;
06324     }
06325
06328     void clear_subtype() noexcept
06329     {
06330         m_subtype = 0;
06331         m_has_subtype = false;
06332     }
06333
06334   private:
06335     subtype_type m_subtype = 0;
06336     bool m_has_subtype = false;
06337 };
06338
06339 NLOHMANN_JSON_NAMESPACE_END
06340
06341 // #include <nlohmann/detail/conversions/from_json.hpp>
06342
06343 // #include <nlohmann/detail/conversions/to_json.hpp>
06344
06345 // #include <nlohmann/detail/exceptions.hpp>
06346
06347 // #include <nlohmann/detail/hash.hpp>
06348 //     __ _____ _____ _____
06349 //  __|  |   __|     |   | |  JSON for Modern C++
06350 // |  |  |__   |  |  | | | |  version 3.12.0
06351 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
06352 //
06353 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
06354 // SPDX-License-Identifier: MIT
06355
06356
06357
06358 #include <cstdint> // uint8_t
06359 #include <cstddef> // size_t
06360 #include <functional> // hash
06361
06362 // #include <nlohmann/detail/abi_macros.hpp>
```

```
06363
06364 // #include <nlohmann/detail/value_t.hpp>
06365
06366
06367 NLOHMANN_JSON_NAMESPACE_BEGIN
06368 namespace detail
06369 {
06370
06371 // boost::hash_combine
06372 inline std::size_t combine(std::size_t seed, std::size_t h) noexcept
06373 {
06374     seed ^= h + 0x9e3779b9 + (seed << 6U) + (seed >> 2U);
06375     return seed;
06376 }
06377
06389 template<typename BasicJsonType>
06390 std::size_t hash(const BasicJsonType& j)
06391 {
06392     using string_t = typename BasicJsonType::string_t;
06393     using number_integer_t = typename BasicJsonType::number_integer_t;
06394     using number_unsigned_t = typename BasicJsonType::number_unsigned_t;
06395     using number_float_t = typename BasicJsonType::number_float_t;
06396
06397     const auto type = static_cast<std::size_t>(j.type());
06398     switch (j.type())
06399     {
06400         case BasicJsonType::value_t::null:
06401         case BasicJsonType::value_t::discarded:
06402         {
06403             return combine(type, 0);
06404         }
06405
06406         case BasicJsonType::value_t::object:
06407         {
06408             auto seed = combine(type, j.size());
06409             for (const auto& element : j.items())
06410             {
06411                 const auto h = std::hash<string_t> {}(element.key());
06412                 seed = combine(seed, h);
06413                 seed = combine(seed, hash(element.value()));
06414             }
06415             return seed;
06416         }
06417
06418         case BasicJsonType::value_t::array:
06419         {
06420             auto seed = combine(type, j.size());
06421             for (const auto& element : j)
06422             {
06423                 seed = combine(seed, hash(element));
06424             }
06425             return seed;
06426         }
06427
06428         case BasicJsonType::value_t::string:
06429         {
06430             const auto h = std::hash<string_t> {}(j.template get_ref<const string_t&>());
06431             return combine(type, h);
06432         }
06433
06434         case BasicJsonType::value_t::boolean:
06435         {
06436             const auto h = std::hash<bool> {}(j.template get<bool>());
06437             return combine(type, h);
06438         }
06439
06440         case BasicJsonType::value_t::number_integer:
06441         {
06442             const auto h = std::hash<number_integer_t> {}(j.template get<number_integer_t>());
06443             return combine(type, h);
06444         }
06445
06446         case BasicJsonType::value_t::number_unsigned:
06447         {
06448             const auto h = std::hash<number_unsigned_t> {}(j.template get<number_unsigned_t>());
06449             return combine(type, h);
06450         }
06451
06452         case BasicJsonType::value_t::number_float:
06453         {
06454             const auto h = std::hash<number_float_t> {}(j.template get<number_float_t>());
06455             return combine(type, h);
06456         }
06457
06458         case BasicJsonType::value_t::binary:
06459         {
06460             auto seed = combine(type, j.get_binary().size());
```

```
06461             const auto h = std::hash<bool> {}(j.get_binary().has_subtype());
06462             seed = combine(seed, h);
06463             seed = combine(seed, static_cast<std::size_t>(j.get_binary().subtype()));
06464             for (const auto byte : j.get_binary())
06465             {
06466                 seed = combine(seed, std::hash<std::uint8_t> {}(byte));
06467             }
06468             return seed;
06469         }
06470
06471         default:                    // LCOV_EXCL_LINE
06472             JSON_ASSERT(false); // NOLINT(cert-dcl03-c,hicpp-static-assert,misc-static-assert)
      LCOV_EXCL_LINE
06473             return 0;               // LCOV_EXCL_LINE
06474     }
06475 }
06476
06477 }  // namespace detail
06478 NLOHMANN_JSON_NAMESPACE_END
06479
06480 // #include <nlohmann/detail/input/binary_reader.hpp>
06481 //      __ _____ _____ _____
06482 //   __|  |   __|     |   | |  JSON for Modern C++
06483 // |  |  |__   |  |  | | | |  version 3.12.0
06484 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
06485 //
06486 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
06487 // SPDX-License-Identifier: MIT
06488
06489
06490
06491 #include <algorithm> // generate_n
06492 #include <array> // array
06493 #include <cmath> // ldexp
06494 #include <cstddef> // size_t
06495 #include <cstdint> // uint8_t, uint16_t, uint32_t, uint64_t
06496 #include <cstdio> // snprintf
06497 #include <cstring> // memcpy
06498 #include <iterator> // back_inserter
06499 #include <limits> // numeric_limits
06500 #include <string> // char_traits, string
06501 #include <utility> // make_pair, move
06502 #include <vector> // vector
06503 #ifdef __cpp_lib_byteswap
06504     #include <bit>  //byteswap
06505 #endif
06506
06507 // #include <nlohmann/detail/exceptions.hpp>
06508
06509 // #include <nlohmann/detail/input/input_adapters.hpp>
06510 //      __ _____ _____ _____
06511 //   __|  |   __|     |   | |  JSON for Modern C++
06512 // |  |  |__   |  |  | | | |  version 3.12.0
06513 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
06514 //
06515 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
06516 // SPDX-License-Identifier: MIT
06517
06518
06519
06520 #include <array> // array
06521 #include <cstddef> // size_t
06522 #include <cstring> // strlen
06523 #include <iterator> // begin, end, iterator_traits, random_access_iterator_tag, distance, next
06524 #include <memory> // shared_ptr, make_shared, addressof
06525 #include <numeric> // accumulate
06526 #include <streambuf> // streambuf
06527 #include <string> // string, char_traits
06528 #include <type_traits> // enable_if, is_base_of, is_pointer, is_integral, remove_pointer
06529 #include <utility> // pair, declval
06530
06531 #ifndef JSON_NO_IO
06532     #include <cstdio>   // FILE *
06533     #include <istream>  // istream
06534 #endif                  // JSON_NO_IO
06535
06536 // #include <nlohmann/detail/exceptions.hpp>
06537
06538 // #include <nlohmann/detail/iterators/iterator_traits.hpp>
06539
06540 // #include <nlohmann/detail/macro_scope.hpp>
06541
06542 // #include <nlohmann/detail/meta/type_traits.hpp>
06543
06544
06545 NLOHMANN_JSON_NAMESPACE_BEGIN
06546 namespace detail
```

```
06547 {
06548
06550 enum class input_format_t { json, cbor, msgpack, ubjson, bson, bjdata };
06551
06553 // input adapters //
06555
06556 #ifndef JSON_NO_IO
06561 class file_input_adapter
06562 {
06563   public:
06564     using char_type = char;
06565
06566     JSON_HEDLEY_NON_NULL(2)
06567     explicit file_input_adapter(std::FILE* f) noexcept
06568         : m_file(f)
06569     {
06570         JSON_ASSERT(m_file != nullptr);
06571     }
06572
06573     // make class move-only
06574     file_input_adapter(const file_input_adapter&) = delete;
06575     file_input_adapter(file_input_adapter&&) noexcept = default;
06576     file_input_adapter& operator=(const file_input_adapter&) = delete;
06577     file_input_adapter& operator=(file_input_adapter&&) = delete;
06578     ~file_input_adapter() = default;
06579
06580     std::char_traits<char>::int_type get_character() noexcept
06581     {
06582         return std::fgetc(m_file);
06583     }
06584
06585     // returns the number of characters successfully read
06586     template<class T>
06587     std::size_t get_elements(T* dest, std::size_t count = 1)
06588     {
06589         return fread(dest, 1, sizeof(T) * count, m_file);
06590     }
06591
06592   private:
06594     std::FILE* m_file;
06595 };
06596
06606 class input_stream_adapter
06607 {
06608   public:
06609     using char_type = char;
06610
06611     ~input_stream_adapter()
06612     {
06613         // clear stream flags; we use underlying streambuf I/O, do not
06614         // maintain ifstream flags, except eof
06615         if (is != nullptr)
06616         {
06617             is->clear(is->rdstate() & std::ios::eofbit);
06618         }
06619     }
06620
06621     explicit input_stream_adapter(std::istream& i)
06622         : is(&i), sb(i.rdbuf())
06623     {}
06624
06625     // deleted because of pointer members
06626     input_stream_adapter(const input_stream_adapter&) = delete;
06627     input_stream_adapter& operator=(input_stream_adapter&) = delete;
06628     input_stream_adapter& operator=(input_stream_adapter&&) = delete;
06629
06630     input_stream_adapter(input_stream_adapter&& rhs) noexcept
06631         : is(rhs.is), sb(rhs.sb)
06632     {
06633         rhs.is = nullptr;
06634         rhs.sb = nullptr;
06635     }
06636
06637     // std::istream/std::streambuf use std::char_traits<char>::to_int_type, to
06638     // ensure that std::char_traits<char>::eof() and the character 0xFF do not
06639     // end up as the same value, e.g., 0xFFFFFFFF.
06640     std::char_traits<char>::int_type get_character()
06641     {
06642         auto res = sb->sbumpc();
06643         // set eof manually, as we don't use the istream interface.
06644         if (JSON_HEDLEY_UNLIKELY(res == std::char_traits<char>::eof()))
06645         {
06646             is->clear(is->rdstate() | std::ios::eofbit);
06647         }
06648         return res;
06649     }
06650
```

```
06651     template<class T>
06652     std::size_t get_elements(T* dest, std::size_t count = 1)
06653     {
06654         auto res = static_cast<std::size_t>(sb->sgetn(reinterpret_cast<char*>(dest),
      static_cast<std::streamsize>(count * sizeof(T))));
06655         if (JSON_HEDLEY_UNLIKELY(res < count * sizeof(T)))
06656         {
06657             is->clear(is->rdstate() | std::ios::eofbit);
06658         }
06659         return res;
06660     }
06661
06662   private:
06664     std::istream* is = nullptr;
06665     std::streambuf* sb = nullptr;
06666 };
06667 #endif  // JSON_NO_IO
06668
06669 // General-purpose iterator-based adapter. It might not be as fast as
06670 // theoretically possible for some containers, but it is extremely versatile.
06671 template<typename IteratorType>
06672 class iterator_input_adapter
06673 {
06674   public:
06675     using char_type = typename std::iterator_traits<IteratorType>::value_type;
06676
06677     iterator_input_adapter(IteratorType first, IteratorType last)
06678         : current(std::move(first)), end(std::move(last))
06679     {}
06680
06681     typename char_traits<char_type>::int_type get_character()
06682     {
06683         if (JSON_HEDLEY_LIKELY(current != end))
06684         {
06685             auto result = char_traits<char_type>::to_int_type(*current);
06686             std::advance(current, 1);
06687             return result;
06688         }
06689
06690         return char_traits<char_type>::eof();
06691     }
06692
06693     // for general iterators, we cannot really do something better than falling back to processing the
      range one-by-one
06694     template<class T>
06695     std::size_t get_elements(T* dest, std::size_t count = 1)
06696     {
06697         auto* ptr = reinterpret_cast<char*>(dest);
06698         for (std::size_t read_index = 0; read_index < count * sizeof(T); ++read_index)
06699         {
06700             if (JSON_HEDLEY_LIKELY(current != end))
06701             {
06702                 ptr[read_index] = static_cast<char>(*current);
06703                 std::advance(current, 1);
06704             }
06705             else
06706             {
06707                 return read_index;
06708             }
06709         }
06710         return count * sizeof(T);
06711     }
06712
06713   private:
06714     IteratorType current;
06715     IteratorType end;
06716
06717     template<typename BaseInputAdapter, size_t T>
06718     friend struct wide_string_input_helper;
06719
06720     bool empty() const
06721     {
06722         return current == end;
06723     }
06724 };
06725
06726 template<typename BaseInputAdapter, size_t T>
06727 struct wide_string_input_helper;
06728
06729 template<typename BaseInputAdapter>
06730 struct wide_string_input_helper<BaseInputAdapter, 4>
06731 {
06732     // UTF-32
06733     static void fill_buffer(BaseInputAdapter& input,
06734                             std::array<std::char_traits<char>::int_type, 4>& utf8_bytes,
06735                             size_t& utf8_bytes_index,
06736                             size_t& utf8_bytes_filled)
```

```
06737     {
06738         utf8_bytes_index = 0;
06739 
06740         if (JSON_HEDLEY_UNLIKELY(input.empty()))
06741         {
06742             utf8_bytes[0] = std::char_traits<char>::eof();
06743             utf8_bytes_filled = 1;
06744         }
06745         else
06746         {
06747             // get the current character
06748             const auto wc = input.get_character();
06749 
06750             // UTF-32 to UTF-8 encoding
06751             if (wc < 0x80)
06752             {
06753                 utf8_bytes[0] = static_cast<std::char_traits<char>::int_type>(wc);
06754                 utf8_bytes_filled = 1;
06755             }
06756             else if (wc <= 0x7FF)
06757             {
06758                 utf8_bytes[0] = static_cast<std::char_traits<char>::int_type>(0xC0u |
    ((static_cast<unsigned int>(wc) » 6u) & 0x1Fu));
06759                 utf8_bytes[1] = static_cast<std::char_traits<char>::int_type>(0x80u |
    (static_cast<unsigned int>(wc) & 0x3Fu));
06760                 utf8_bytes_filled = 2;
06761             }
06762             else if (wc <= 0xFFFF)
06763             {
06764                 utf8_bytes[0] = static_cast<std::char_traits<char>::int_type>(0xE0u |
    ((static_cast<unsigned int>(wc) » 12u) & 0x0Fu));
06765                 utf8_bytes[1] = static_cast<std::char_traits<char>::int_type>(0x80u |
    ((static_cast<unsigned int>(wc) » 6u) & 0x3Fu));
06766                 utf8_bytes[2] = static_cast<std::char_traits<char>::int_type>(0x80u |
    (static_cast<unsigned int>(wc) & 0x3Fu));
06767                 utf8_bytes_filled = 3;
06768             }
06769             else if (wc <= 0x10FFFF)
06770             {
06771                 utf8_bytes[0] = static_cast<std::char_traits<char>::int_type>(0xF0u |
    ((static_cast<unsigned int>(wc) » 18u) & 0x07u));
06772                 utf8_bytes[1] = static_cast<std::char_traits<char>::int_type>(0x80u |
    ((static_cast<unsigned int>(wc) » 12u) & 0x3Fu));
06773                 utf8_bytes[2] = static_cast<std::char_traits<char>::int_type>(0x80u |
    ((static_cast<unsigned int>(wc) » 6u) & 0x3Fu));
06774                 utf8_bytes[3] = static_cast<std::char_traits<char>::int_type>(0x80u |
    (static_cast<unsigned int>(wc) & 0x3Fu));
06775                 utf8_bytes_filled = 4;
06776             }
06777             else
06778             {
06779                 // unknown character
06780                 utf8_bytes[0] = static_cast<std::char_traits<char>::int_type>(wc);
06781                 utf8_bytes_filled = 1;
06782             }
06783         }
06784     }
06785 };
06786 
06787 template<typename BaseInputAdapter>
06788 struct wide_string_input_helper<BaseInputAdapter, 2>
06789 {
06790     // UTF-16
06791     static void fill_buffer(BaseInputAdapter& input,
06792                             std::array<std::char_traits<char>::int_type, 4>& utf8_bytes,
06793                             size_t& utf8_bytes_index,
06794                             size_t& utf8_bytes_filled)
06795     {
06796         utf8_bytes_index = 0;
06797 
06798         if (JSON_HEDLEY_UNLIKELY(input.empty()))
06799         {
06800             utf8_bytes[0] = std::char_traits<char>::eof();
06801             utf8_bytes_filled = 1;
06802         }
06803         else
06804         {
06805             // get the current character
06806             const auto wc = input.get_character();
06807 
06808             // UTF-16 to UTF-8 encoding
06809             if (wc < 0x80)
06810             {
06811                 utf8_bytes[0] = static_cast<std::char_traits<char>::int_type>(wc);
06812                 utf8_bytes_filled = 1;
06813             }
06814             else if (wc <= 0x7FF)
```

```
06815             {
06816                 utf8_bytes[0] = static_cast<std::char_traits<char>::int_type>(0xC0u |
    ((static_cast<unsigned int>(wc) » 6u)));
06817                 utf8_bytes[1] = static_cast<std::char_traits<char>::int_type>(0x80u |
    (static_cast<unsigned int>(wc) & 0x3Fu));
06818                 utf8_bytes_filled = 2;
06819             }
06820             else if (0xD800 > wc || wc >= 0xE000)
06821             {
06822                 utf8_bytes[0] = static_cast<std::char_traits<char>::int_type>(0xE0u |
    ((static_cast<unsigned int>(wc) » 12u)));
06823                 utf8_bytes[1] = static_cast<std::char_traits<char>::int_type>(0x80u |
    ((static_cast<unsigned int>(wc) » 6u) & 0x3Fu));
06824                 utf8_bytes[2] = static_cast<std::char_traits<char>::int_type>(0x80u |
    (static_cast<unsigned int>(wc) & 0x3Fu));
06825                 utf8_bytes_filled = 3;
06826             }
06827             else
06828             {
06829                 if (JSON_HEDLEY_UNLIKELY(!input.empty()))
06830                 {
06831                     const auto wc2 = static_cast<unsigned int>(input.get_character());
06832                     const auto charcode = 0x10000u + (((static_cast<unsigned int>(wc) & 0x3FFu) « 10u)
    | (wc2 & 0x3FFu));
06833                     utf8_bytes[0] = static_cast<std::char_traits<char>::int_type>(0xF0u | (charcode »
    18u));
06834                     utf8_bytes[1] = static_cast<std::char_traits<char>::int_type>(0x80u | ((charcode »
    12u) & 0x3Fu));
06835                     utf8_bytes[2] = static_cast<std::char_traits<char>::int_type>(0x80u | ((charcode »
    6u) & 0x3Fu));
06836                     utf8_bytes[3] = static_cast<std::char_traits<char>::int_type>(0x80u | (charcode &
    0x3Fu));
06837                     utf8_bytes_filled = 4;
06838                 }
06839                 else
06840                 {
06841                     utf8_bytes[0] = static_cast<std::char_traits<char>::int_type>(wc);
06842                     utf8_bytes_filled = 1;
06843                 }
06844             }
06845         }
06846     }
06847 };
06848
06849 // Wraps another input adapter to convert wide character types into individual bytes.
06850 template<typename BaseInputAdapter, typename WideCharType>
06851 class wide_string_input_adapter
06852 {
06853   public:
06854     using char_type = char;
06855
06856     wide_string_input_adapter(BaseInputAdapter base)
06857         : base_adapter(base) {}
06858
06859     typename std::char_traits<char>::int_type get_character() noexcept
06860     {
06861         // check if the buffer needs to be filled
06862         if (utf8_bytes_index == utf8_bytes_filled)
06863         {
06864             fill_buffer<sizeof(WideCharType)>();
06865
06866             JSON_ASSERT(utf8_bytes_filled > 0);
06867             JSON_ASSERT(utf8_bytes_index == 0);
06868         }
06869
06870         // use buffer
06871         JSON_ASSERT(utf8_bytes_filled > 0);
06872         JSON_ASSERT(utf8_bytes_index < utf8_bytes_filled);
06873         return utf8_bytes[utf8_bytes_index++];
06874     }
06875
06876     // parsing binary with wchar doesn't make sense, but since the parsing mode can be runtime, we
    need something here
06877     template<class T>
06878     std::size_t get_elements(T* /*dest*/, std::size_t /*count*/ = 1)
06879     {
06880         JSON_THROW(parse_error::create(112, 1, "wide string type cannot be interpreted as binary
    data", nullptr));
06881     }
06882
06883   private:
06884     BaseInputAdapter base_adapter;
06885
06886     template<size_t T>
06887     void fill_buffer()
06888     {
06889         wide_string_input_helper<BaseInputAdapter, T>::fill_buffer(base_adapter, utf8_bytes,
```

```
    utf8_bytes_index, utf8_bytes_filled);
06890     }
06891
06893     std::array<std::char_traits<char>::int_type, 4> utf8_bytes = {{0, 0, 0, 0}};
06894
06896     std::size_t utf8_bytes_index = 0;
06898     std::size_t utf8_bytes_filled = 0;
06899 };
06900
06901 template<typename IteratorType, typename Enable = void>
06902 struct iterator_input_adapter_factory
06903 {
06904     using iterator_type = IteratorType;
06905     using char_type = typename std::iterator_traits<iterator_type>::value_type;
06906     using adapter_type = iterator_input_adapter<iterator_type>;
06907
06908     static adapter_type create(IteratorType first, IteratorType last)
06909     {
06910         return adapter_type(std::move(first), std::move(last));
06911     }
06912 };
06913
06914 template<typename T>
06915 struct is_iterator_of_multibyte
06916 {
06917     using value_type = typename std::iterator_traits<T>::value_type;
06918     enum // NOLINT(cppcoreguidelines-use-enum-class)
06919     {
06920         value = sizeof(value_type) > 1
06921     };
06922 };
06923
06924 template<typename IteratorType>
06925 struct iterator_input_adapter_factory<IteratorType,
    enable_if_t<is_iterator_of_multibyte<IteratorType>::value»
06926 {
06927     using iterator_type = IteratorType;
06928     using char_type = typename std::iterator_traits<iterator_type>::value_type;
06929     using base_adapter_type = iterator_input_adapter<iterator_type>;
06930     using adapter_type = wide_string_input_adapter<base_adapter_type, char_type>;
06931
06932     static adapter_type create(IteratorType first, IteratorType last)
06933     {
06934         return adapter_type(base_adapter_type(std::move(first), std::move(last)));
06935     }
06936 };
06937
06938 // General purpose iterator-based input
06939 template<typename IteratorType>
06940 typename iterator_input_adapter_factory<IteratorType>::adapter_type input_adapter(IteratorType first,
    IteratorType last)
06941 {
06942     using factory_type = iterator_input_adapter_factory<IteratorType>;
06943     return factory_type::create(first, last);
06944 }
06945
06946 // Convenience shorthand from container to iterator
06947 // Enables ADL on begin(container) and end(container)
06948 // Encloses the using declarations in namespace for not to leak them to outside scope
06949
06950 namespace container_input_adapter_factory_impl
06951 {
06952
06953 using std::begin;
06954 using std::end;
06955
06956 template<typename ContainerType, typename Enable = void>
06957 struct container_input_adapter_factory {};
06958
06959 template<typename ContainerType>
06960 struct container_input_adapter_factory< ContainerType,
06961        void_t<decltype(begin(std::declval<ContainerType>()), end(std::declval<ContainerType>()))»
06962        {
06963            using adapter_type = decltype(input_adapter(begin(std::declval<ContainerType>()),
    end(std::declval<ContainerType>())));
06964
06965            static adapter_type create(const ContainerType& container)
06966 {
06967     return input_adapter(begin(container), end(container));
06968 }
06969        };
06970
06971 }  // namespace container_input_adapter_factory_impl
06972
06973 template<typename ContainerType>
06974 typename
    container_input_adapter_factory_impl::container_input_adapter_factory<ContainerType>::adapter_type
```

```
    input_adapter(const ContainerType& container)
06975 {
06976     return
    container_input_adapter_factory_impl::container_input_adapter_factory<ContainerType>::create(container);
06977 }
06978
06979 // specialization for std::string
06980 using string_input_adapter_type = decltype(input_adapter(std::declval<std::string>()));
06981
06982 #ifndef JSON_NO_IO
06983 // Special cases with fast paths
06984 inline file_input_adapter input_adapter(std::FILE* file)
06985 {
06986     if (file == nullptr)
06987     {
06988         JSON_THROW(parse_error::create(101, 0, "attempting to parse an empty input; check that your
    input string or stream contains the expected JSON", nullptr));
06989     }
06990     return file_input_adapter(file);
06991 }
06992
06993 inline input_stream_adapter input_adapter(std::istream& stream)
06994 {
06995     return input_stream_adapter(stream);
06996 }
06997
06998 inline input_stream_adapter input_adapter(std::istream&& stream)
06999 {
07000     return input_stream_adapter(stream);
07001 }
07002 #endif  // JSON_NO_IO
07003
07004 using contiguous_bytes_input_adapter = decltype(input_adapter(std::declval<const char*>(),
    std::declval<const char*>()));
07005
07006 // Null-delimited strings, and the like.
07007 template < typename CharT,
07008            typename std::enable_if <
07009                std::is_pointer<CharT>::value&&
07010                !std::is_array<CharT>::value&&
07011                std::is_integral<typename std::remove_pointer<CharT>::type>::value&&
07012                sizeof(typename std::remove_pointer<CharT>::type) == 1,
07013                int >::type = 0 >
07014 contiguous_bytes_input_adapter input_adapter(CharT b)
07015 {
07016     if (b == nullptr)
07017     {
07018         JSON_THROW(parse_error::create(101, 0, "attempting to parse an empty input; check that your
    input string or stream contains the expected JSON", nullptr));
07019     }
07020     auto length = std::strlen(reinterpret_cast<const char*>(b));
07021     const auto* ptr = reinterpret_cast<const char*>(b);
07022     return input_adapter(ptr, ptr + length); // cppcheck-suppress[nullPointerArithmeticRedundantCheck]
07023 }
07024
07025 template<typename T, std::size_t N>
07026 auto input_adapter(T (&array)[N]) -> decltype(input_adapter(array, array + N)) //
    NOLINT(cppcoreguidelines-avoid-c-arrays,hicpp-avoid-c-arrays,modernize-avoid-c-arrays)
07027 {
07028     return input_adapter(array, array + N);
07029 }
07030
07031 // This class only handles inputs of input_buffer_adapter type.
07032 // It's required so that expressions like {ptr, len} can be implicitly cast
07033 // to the correct adapter.
07034 class span_input_adapter
07035 {
07036   public:
07037     template < typename CharT,
07038                typename std::enable_if <
07039                    std::is_pointer<CharT>::value&&
07040                    std::is_integral<typename std::remove_pointer<CharT>::type>::value&&
07041                    sizeof(typename std::remove_pointer<CharT>::type) == 1,
07042                    int >::type = 0 >
07043     span_input_adapter(CharT b, std::size_t l)
07044         : ia(reinterpret_cast<const char*>(b), reinterpret_cast<const char*>(b) + l) {}
07045
07046     template<class IteratorType,
07047              typename std::enable_if<
07048                  std::is_same<typename iterator_traits<IteratorType>::iterator_category,
    std::random_access_iterator_tag>::value,
07049                  int>::type = 0>
07050     span_input_adapter(IteratorType first, IteratorType last)
07051         : ia(input_adapter(first, last)) {}
07052
07053     contiguous_bytes_input_adapter&& get()
07054     {
```

```
07055         return std::move(ia); // NOLINT(hicpp-move-const-arg,performance-move-const-arg)
07056     }
07057
07058   private:
07059     contiguous_bytes_input_adapter ia;
07060 };
07061
07062 }  // namespace detail
07063 NLOHMANN_JSON_NAMESPACE_END
07064
07065 // #include <nlohmann/detail/input/json_sax.hpp>
07066 //     __ _____ _____ _____
07067 //  __|  |   __|     |   | |  JSON for Modern C++
07068 // |  |  |__   |  |  | | | |  version 3.12.0
07069 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
07070 //
07071 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
07072 // SPDX-License-Identifier: MIT
07073
07074
07075
07076 #include <cstddef>
07077 #include <string> // string
07078 #include <type_traits> // enable_if_t
07079 #include <utility> // move
07080 #include <vector> // vector
07081
07082 // #include <nlohmann/detail/exceptions.hpp>
07083
07084 // #include <nlohmann/detail/input/lexer.hpp>
07085 //     __ _____ _____ _____
07086 //  __|  |   __|     |   | |  JSON for Modern C++
07087 // |  |  |__   |  |  | | | |  version 3.12.0
07088 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
07089 //
07090 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
07091 // SPDX-License-Identifier: MIT
07092
07093
07094
07095 #include <array> // array
07096 #include <clocale> // localeconv
07097 #include <cstddef> // size_t
07098 #include <cstdio> // snprintf
07099 #include <cstdlib> // strtof, strtod, strtold, strtoll, strtoull
07100 #include <initializer_list> // initializer_list
07101 #include <string> // char_traits, string
07102 #include <utility> // move
07103 #include <vector> // vector
07104
07105 // #include <nlohmann/detail/input/input_adapters.hpp>
07106
07107 // #include <nlohmann/detail/input/position_t.hpp>
07108
07109 // #include <nlohmann/detail/macro_scope.hpp>
07110
07111 // #include <nlohmann/detail/meta/type_traits.hpp>
07112
07113
07114 NLOHMANN_JSON_NAMESPACE_BEGIN
07115 namespace detail
07116 {
07117
07119 // lexer //
07121
07122 template<typename BasicJsonType>
07123 class lexer_base
07124 {
07125   public:
07127     enum class token_type
07128     {
07129         uninitialized,
07130         literal_true,
07131         literal_false,
07132         literal_null,
07133         value_string,
07134         value_unsigned,
07135         value_integer,
07136         value_float,
07137         begin_array,
07138         begin_object,
07139         end_array,
07140         end_object,
07141         name_separator,
07142         value_separator,
07143         parse_error,
07144         end_of_input,
```

```
07145         literal_or_value
07146     };
07147
07149     JSON_HEDLEY_RETURNS_NON_NULL
07150     JSON_HEDLEY_CONST
07151     static const char* token_type_name(const token_type t) noexcept
07152     {
07153         switch (t)
07154         {
07155             case token_type::uninitialized:
07156                 return "<uninitialized>";
07157             case token_type::literal_true:
07158                 return "true literal";
07159             case token_type::literal_false:
07160                 return "false literal";
07161             case token_type::literal_null:
07162                 return "null literal";
07163             case token_type::value_string:
07164                 return "string literal";
07165             case token_type::value_unsigned:
07166             case token_type::value_integer:
07167             case token_type::value_float:
07168                 return "number literal";
07169             case token_type::begin_array:
07170                 return "'['";
07171             case token_type::begin_object:
07172                 return "'{'";
07173             case token_type::end_array:
07174                 return "']'";
07175             case token_type::end_object:
07176                 return "'}'";
07177             case token_type::name_separator:
07178                 return "':'";
07179             case token_type::value_separator:
07180                 return "','";
07181             case token_type::parse_error:
07182                 return "<parse error>";
07183             case token_type::end_of_input:
07184                 return "end of input";
07185             case token_type::literal_or_value:
07186                 return "'[', '{', or a literal";
07187             // LCOV_EXCL_START
07188             default: // catch non-enum values
07189                 return "unknown token";
07190                 // LCOV_EXCL_STOP
07191         }
07192     }
07193 };
07199 template<typename BasicJsonType, typename InputAdapterType>
07200 class lexer : public lexer_base<BasicJsonType>
07201 {
07202     using number_integer_t = typename BasicJsonType::number_integer_t;
07203     using number_unsigned_t = typename BasicJsonType::number_unsigned_t;
07204     using number_float_t = typename BasicJsonType::number_float_t;
07205     using string_t = typename BasicJsonType::string_t;
07206     using char_type = typename InputAdapterType::char_type;
07207     using char_int_type = typename char_traits<char_type>::int_type;
07208
07209   public:
07210     using token_type = typename lexer_base<BasicJsonType>::token_type;
07211
07212     explicit lexer(InputAdapterType&& adapter, bool ignore_comments_ = false) noexcept
07213         : ia(std::move(adapter))
07214         , ignore_comments(ignore_comments_)
07215         , decimal_point_char(static_cast<char_int_type>(get_decimal_point()))
07216     {}
07217
07218     // deleted because of pointer members
07219     lexer(const lexer&) = delete;
07220     lexer(lexer&&) = default; // NOLINT(hicpp-noexcept-move,performance-noexcept-move-constructor)
07221     lexer& operator=(lexer&) = delete;
07222     lexer& operator=(lexer&&) = default; //
      NOLINT(hicpp-noexcept-move,performance-noexcept-move-constructor)
07223     ~lexer() = default;
07224
07225   private:
07227     // locales
07229
07231     JSON_HEDLEY_PURE
07232     static char get_decimal_point() noexcept
07233     {
07234         const auto* loc = localeconv();
07235         JSON_ASSERT(loc != nullptr);
07236         return (loc->decimal_point == nullptr) ? '.' : *(loc->decimal_point);
07237     }
07238
07240     // scan functions
```

```
07242
07258     int get_codepoint()
07259     {
07260         // this function only makes sense after reading `\u`
07261         JSON_ASSERT(current == 'u');
07262         int codepoint = 0;
07263
07264         const auto factors = { 12u, 8u, 4u, 0u };
07265         for (const auto factor : factors)
07266         {
07267             get();
07268
07269             if (current >= '0' && current <= '9')
07270             {
07271                 codepoint += static_cast<int>((static_cast<unsigned int>(current) - 0x30u) << factor);
07272             }
07273             else if (current >= 'A' && current <= 'F')
07274             {
07275                 codepoint += static_cast<int>((static_cast<unsigned int>(current) - 0x37u) << factor);
07276             }
07277             else if (current >= 'a' && current <= 'f')
07278             {
07279                 codepoint += static_cast<int>((static_cast<unsigned int>(current) - 0x57u) << factor);
07280             }
07281             else
07282             {
07283                 return -1;
07284             }
07285         }
07286
07287         JSON_ASSERT(0x0000 <= codepoint && codepoint <= 0xFFFF);
07288         return codepoint;
07289     }
07290
07306     bool next_byte_in_range(std::initializer_list<char_int_type> ranges)
07307     {
07308         JSON_ASSERT(ranges.size() == 2 || ranges.size() == 4 || ranges.size() == 6);
07309         add(current);
07310
07311         for (auto range = ranges.begin(); range != ranges.end(); ++range)
07312         {
07313             get();
07314             if (JSON_HEDLEY_LIKELY(*range <= current && current <= *(++range))) //
      NOLINT(bugprone-inc-dec-in-conditions)
07315             {
07316                 add(current);
07317             }
07318             else
07319             {
07320                 error_message = "invalid string: ill-formed UTF-8 byte";
07321                 return false;
07322             }
07323         }
07324
07325         return true;
07326     }
07327
07343     token_type scan_string()
07344     {
07345         // reset token_buffer (ignore opening quote)
07346         reset();
07347
07348         // we entered the function by reading an open quote
07349         JSON_ASSERT(current == '\"');
07350
07351         while (true)
07352         {
07353             // get the next character
07354             switch (get())
07355             {
07356                 // end of file while parsing the string
07357                 case char_traits<char_type>::eof():
07358                 {
07359                     error_message = "invalid string: missing closing quote";
07360                     return token_type::parse_error;
07361                 }
07362
07363                 // closing quote
07364                 case '\"':
07365                 {
07366                     return token_type::value_string;
07367                 }
07368
07369                 // escapes
07370                 case '\\':
07371                 {
07372                     switch (get())
```

```
07373                     {
07374                         // quotation mark
07375                         case '\"':
07376                             add('\"');
07377                             break;
07378                         // reverse solidus
07379                         case '\\':
07380                             add('\\');
07381                             break;
07382                         // solidus
07383                         case '/':
07384                             add('/');
07385                             break;
07386                         // backspace
07387                         case 'b':
07388                             add('\b');
07389                             break;
07390                         // form feed
07391                         case 'f':
07392                             add('\f');
07393                             break;
07394                         // line feed
07395                         case 'n':
07396                             add('\n');
07397                             break;
07398                         // carriage return
07399                         case 'r':
07400                             add('\r');
07401                             break;
07402                         // tab
07403                         case 't':
07404                             add('\t');
07405                             break;
07406
07407                         // unicode escapes
07408                         case 'u':
07409                         {
07410                             const int codepoint1 = get_codepoint();
07411                             int codepoint = codepoint1; // start with codepoint1
07412
07413                             if (JSON_HEDLEY_UNLIKELY(codepoint1 == -1))
07414                             {
07415                                 error_message = "invalid string: '\\u' must be followed by 4 hex
      digits";
07416                                 return token_type::parse_error;
07417                             }
07418
07419                             // check if code point is a high surrogate
07420                             if (0xD800 <= codepoint1 && codepoint1 <= 0xDBFF)
07421                             {
07422                                 // expect next \uxxxx entry
07423                                 if (JSON_HEDLEY_LIKELY(get() == '\\' && get() == 'u'))
07424                                 {
07425                                     const int codepoint2 = get_codepoint();
07426
07427                                     if (JSON_HEDLEY_UNLIKELY(codepoint2 == -1))
07428                                     {
07429                                         error_message = "invalid string: '\\u' must be followed by 4
      hex digits";
07430                                         return token_type::parse_error;
07431                                     }
07432
07433                                     // check if codepoint2 is a low surrogate
07434                                     if (JSON_HEDLEY_LIKELY(0xDC00 <= codepoint2 && codepoint2 <=
      0xDFFF))
07435                                     {
07436                                         // overwrite codepoint
07437                                         codepoint = static_cast<int>(
07438                                                         // high surrogate occupies the most
      significant 22 bits
07439                                                         (static_cast<unsigned int>(codepoint1) << 10u)
07440                                                         // low surrogate occupies the least
      significant 15 bits
07441                                                         + static_cast<unsigned int>(codepoint2)
07442                                                         // there is still the 0xD800, 0xDC00, and
      0x10000 noise
07443                                                         // in the result, so we have to subtract with:
07444                                                         // (0xD800 << 10) + DC00 - 0x10000 = 0x35FDC00
07445                                                         - 0x35FDC00u);
07446                                     }
07447                                     else
07448                                     {
07449                                         error_message = "invalid string: surrogate U+D800..U+DBFF must
      be followed by U+DC00..U+DFFF";
07450                                         return token_type::parse_error;
07451                                     }
07452                                 }
```

```
07453                                 else
07454                                 {
07455                                     error_message = "invalid string: surrogate U+D800..U+DBFF must be
      followed by U+DC00..U+DFFF";
07456                                     return token_type::parse_error;
07457                                 }
07458                             }
07459                             else
07460                             {
07461                                 if (JSON_HEDLEY_UNLIKELY(0xDC00 <= codepoint1 && codepoint1 <=
      0xDFFF))
07462                                 {
07463                                     error_message = "invalid string: surrogate U+DC00..U+DFFF must
      follow U+D800..U+DBFF";
07464                                     return token_type::parse_error;
07465                                 }
07466                             }
07467 
07468                             // the result of the above calculation yields a proper codepoint
07469                             JSON_ASSERT(0x00 <= codepoint && codepoint <= 0x10FFFF);
07470 
07471                             // translate codepoint into bytes
07472                             if (codepoint < 0x80)
07473                             {
07474                                 // 1-byte characters: 0xxxxxxx (ASCII)
07475                                 add(static_cast<char_int_type>(codepoint));
07476                             }
07477                             else if (codepoint <= 0x7FF)
07478                             {
07479                                 // 2-byte characters: 110xxxxx 10xxxxxx
07480                                 add(static_cast<char_int_type>(0xC0u | (static_cast<unsigned
      int>(codepoint) » 6u)));
07481                                 add(static_cast<char_int_type>(0x80u | (static_cast<unsigned
      int>(codepoint) & 0x3Fu)));
07482                             }
07483                             else if (codepoint <= 0xFFFF)
07484                             {
07485                                 // 3-byte characters: 1110xxxx 10xxxxxx 10xxxxxx
07486                                 add(static_cast<char_int_type>(0xE0u | (static_cast<unsigned
      int>(codepoint) » 12u)));
07487                                 add(static_cast<char_int_type>(0x80u | ((static_cast<unsigned
      int>(codepoint) » 6u) & 0x3Fu)));
07488                                 add(static_cast<char_int_type>(0x80u | (static_cast<unsigned
      int>(codepoint) & 0x3Fu)));
07489                             }
07490                             else
07491                             {
07492                                 // 4-byte characters: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
07493                                 add(static_cast<char_int_type>(0xF0u | (static_cast<unsigned
      int>(codepoint) » 18u)));
07494                                 add(static_cast<char_int_type>(0x80u | ((static_cast<unsigned
      int>(codepoint) » 12u) & 0x3Fu)));
07495                                 add(static_cast<char_int_type>(0x80u | ((static_cast<unsigned
      int>(codepoint) » 6u) & 0x3Fu)));
07496                                 add(static_cast<char_int_type>(0x80u | (static_cast<unsigned
      int>(codepoint) & 0x3Fu)));
07497                             }
07498 
07499                             break;
07500                         }
07501 
07502                         // other characters after escape
07503                         default:
07504                             error_message = "invalid string: forbidden character after backslash";
07505                             return token_type::parse_error;
07506                     }
07507 
07508                     break;
07509                 }
07510 
07511                 // invalid control characters
07512                 case 0x00:
07513                 {
07514                     error_message = "invalid string: control character U+0000 (NUL) must be escaped to
      \\u0000";
07515                     return token_type::parse_error;
07516                 }
07517 
07518                 case 0x01:
07519                 {
07520                     error_message = "invalid string: control character U+0001 (SOH) must be escaped to
      \\u0001";
07521                     return token_type::parse_error;
07522                 }
07523 
07524                 case 0x02:
07525                 {
```

```
07526                     error_message = "invalid string: control character U+0002 (STX) must be escaped to
      \\u0002";
07527                     return token_type::parse_error;
07528                 }
07529
07530                 case 0x03:
07531                 {
07532                     error_message = "invalid string: control character U+0003 (ETX) must be escaped to
      \\u0003";
07533                     return token_type::parse_error;
07534                 }
07535
07536                 case 0x04:
07537                 {
07538                     error_message = "invalid string: control character U+0004 (EOT) must be escaped to
      \\u0004";
07539                     return token_type::parse_error;
07540                 }
07541
07542                 case 0x05:
07543                 {
07544                     error_message = "invalid string: control character U+0005 (ENQ) must be escaped to
      \\u0005";
07545                     return token_type::parse_error;
07546                 }
07547
07548                 case 0x06:
07549                 {
07550                     error_message = "invalid string: control character U+0006 (ACK) must be escaped to
      \\u0006";
07551                     return token_type::parse_error;
07552                 }
07553
07554                 case 0x07:
07555                 {
07556                     error_message = "invalid string: control character U+0007 (BEL) must be escaped to
      \\u0007";
07557                     return token_type::parse_error;
07558                 }
07559
07560                 case 0x08:
07561                 {
07562                     error_message = "invalid string: control character U+0008 (BS) must be escaped to
      \\u0008 or \\b";
07563                     return token_type::parse_error;
07564                 }
07565
07566                 case 0x09:
07567                 {
07568                     error_message = "invalid string: control character U+0009 (HT) must be escaped to
      \\u0009 or \\t";
07569                     return token_type::parse_error;
07570                 }
07571
07572                 case 0x0A:
07573                 {
07574                     error_message = "invalid string: control character U+000A (LF) must be escaped to
      \\u000A or \\n";
07575                     return token_type::parse_error;
07576                 }
07577
07578                 case 0x0B:
07579                 {
07580                     error_message = "invalid string: control character U+000B (VT) must be escaped to
      \\u000B";
07581                     return token_type::parse_error;
07582                 }
07583
07584                 case 0x0C:
07585                 {
07586                     error_message = "invalid string: control character U+000C (FF) must be escaped to
      \\u000C or \\f";
07587                     return token_type::parse_error;
07588                 }
07589
07590                 case 0x0D:
07591                 {
07592                     error_message = "invalid string: control character U+000D (CR) must be escaped to
      \\u000D or \\r";
07593                     return token_type::parse_error;
07594                 }
07595
07596                 case 0x0E:
07597                 {
07598                     error_message = "invalid string: control character U+000E (SO) must be escaped to
      \\u000E";
07599                     return token_type::parse_error;
```

```
07600                 }
07601
07602                 case 0x0F:
07603                 {
07604                     error_message = "invalid string: control character U+000F (SI) must be escaped to
      \\u000F";
07605                     return token_type::parse_error;
07606                 }
07607
07608                 case 0x10:
07609                 {
07610                     error_message = "invalid string: control character U+0010 (DLE) must be escaped to
      \\u0010";
07611                     return token_type::parse_error;
07612                 }
07613
07614                 case 0x11:
07615                 {
07616                     error_message = "invalid string: control character U+0011 (DC1) must be escaped to
      \\u0011";
07617                     return token_type::parse_error;
07618                 }
07619
07620                 case 0x12:
07621                 {
07622                     error_message = "invalid string: control character U+0012 (DC2) must be escaped to
      \\u0012";
07623                     return token_type::parse_error;
07624                 }
07625
07626                 case 0x13:
07627                 {
07628                     error_message = "invalid string: control character U+0013 (DC3) must be escaped to
      \\u0013";
07629                     return token_type::parse_error;
07630                 }
07631
07632                 case 0x14:
07633                 {
07634                     error_message = "invalid string: control character U+0014 (DC4) must be escaped to
      \\u0014";
07635                     return token_type::parse_error;
07636                 }
07637
07638                 case 0x15:
07639                 {
07640                     error_message = "invalid string: control character U+0015 (NAK) must be escaped to
      \\u0015";
07641                     return token_type::parse_error;
07642                 }
07643
07644                 case 0x16:
07645                 {
07646                     error_message = "invalid string: control character U+0016 (SYN) must be escaped to
      \\u0016";
07647                     return token_type::parse_error;
07648                 }
07649
07650                 case 0x17:
07651                 {
07652                     error_message = "invalid string: control character U+0017 (ETB) must be escaped to
      \\u0017";
07653                     return token_type::parse_error;
07654                 }
07655
07656                 case 0x18:
07657                 {
07658                     error_message = "invalid string: control character U+0018 (CAN) must be escaped to
      \\u0018";
07659                     return token_type::parse_error;
07660                 }
07661
07662                 case 0x19:
07663                 {
07664                     error_message = "invalid string: control character U+0019 (EM) must be escaped to
      \\u0019";
07665                     return token_type::parse_error;
07666                 }
07667
07668                 case 0x1A:
07669                 {
07670                     error_message = "invalid string: control character U+001A (SUB) must be escaped to
      \\u001A";
07671                     return token_type::parse_error;
07672                 }
07673
07674                 case 0x1B:
```

```
07675                 {
07676                     error_message = "invalid string: control character U+001B (ESC) must be escaped to
      \\u001B";
07677                     return token_type::parse_error;
07678                 }
07679
07680                 case 0x1C:
07681                 {
07682                     error_message = "invalid string: control character U+001C (FS) must be escaped to
      \\u001C";
07683                     return token_type::parse_error;
07684                 }
07685
07686                 case 0x1D:
07687                 {
07688                     error_message = "invalid string: control character U+001D (GS) must be escaped to
      \\u001D";
07689                     return token_type::parse_error;
07690                 }
07691
07692                 case 0x1E:
07693                 {
07694                     error_message = "invalid string: control character U+001E (RS) must be escaped to
      \\u001E";
07695                     return token_type::parse_error;
07696                 }
07697
07698                 case 0x1F:
07699                 {
07700                     error_message = "invalid string: control character U+001F (US) must be escaped to
      \\u001F";
07701                     return token_type::parse_error;
07702                 }
07703
07704                 // U+0020..U+007F (except U+0022 (quote) and U+005C (backspace))
07705                 case 0x20:
07706                 case 0x21:
07707                 case 0x23:
07708                 case 0x24:
07709                 case 0x25:
07710                 case 0x26:
07711                 case 0x27:
07712                 case 0x28:
07713                 case 0x29:
07714                 case 0x2A:
07715                 case 0x2B:
07716                 case 0x2C:
07717                 case 0x2D:
07718                 case 0x2E:
07719                 case 0x2F:
07720                 case 0x30:
07721                 case 0x31:
07722                 case 0x32:
07723                 case 0x33:
07724                 case 0x34:
07725                 case 0x35:
07726                 case 0x36:
07727                 case 0x37:
07728                 case 0x38:
07729                 case 0x39:
07730                 case 0x3A:
07731                 case 0x3B:
07732                 case 0x3C:
07733                 case 0x3D:
07734                 case 0x3E:
07735                 case 0x3F:
07736                 case 0x40:
07737                 case 0x41:
07738                 case 0x42:
07739                 case 0x43:
07740                 case 0x44:
07741                 case 0x45:
07742                 case 0x46:
07743                 case 0x47:
07744                 case 0x48:
07745                 case 0x49:
07746                 case 0x4A:
07747                 case 0x4B:
07748                 case 0x4C:
07749                 case 0x4D:
07750                 case 0x4E:
07751                 case 0x4F:
07752                 case 0x50:
07753                 case 0x51:
07754                 case 0x52:
07755                 case 0x53:
07756                 case 0x54:
```

```
07757                 case 0x55:
07758                 case 0x56:
07759                 case 0x57:
07760                 case 0x58:
07761                 case 0x59:
07762                 case 0x5A:
07763                 case 0x5B:
07764                 case 0x5D:
07765                 case 0x5E:
07766                 case 0x5F:
07767                 case 0x60:
07768                 case 0x61:
07769                 case 0x62:
07770                 case 0x63:
07771                 case 0x64:
07772                 case 0x65:
07773                 case 0x66:
07774                 case 0x67:
07775                 case 0x68:
07776                 case 0x69:
07777                 case 0x6A:
07778                 case 0x6B:
07779                 case 0x6C:
07780                 case 0x6D:
07781                 case 0x6E:
07782                 case 0x6F:
07783                 case 0x70:
07784                 case 0x71:
07785                 case 0x72:
07786                 case 0x73:
07787                 case 0x74:
07788                 case 0x75:
07789                 case 0x76:
07790                 case 0x77:
07791                 case 0x78:
07792                 case 0x79:
07793                 case 0x7A:
07794                 case 0x7B:
07795                 case 0x7C:
07796                 case 0x7D:
07797                 case 0x7E:
07798                 case 0x7F:
07799                 {
07800                     add(current);
07801                     break;
07802                 }
07803
07804                 // U+0080..U+07FF: bytes C2..DF 80..BF
07805                 case 0xC2:
07806                 case 0xC3:
07807                 case 0xC4:
07808                 case 0xC5:
07809                 case 0xC6:
07810                 case 0xC7:
07811                 case 0xC8:
07812                 case 0xC9:
07813                 case 0xCA:
07814                 case 0xCB:
07815                 case 0xCC:
07816                 case 0xCD:
07817                 case 0xCE:
07818                 case 0xCF:
07819                 case 0xD0:
07820                 case 0xD1:
07821                 case 0xD2:
07822                 case 0xD3:
07823                 case 0xD4:
07824                 case 0xD5:
07825                 case 0xD6:
07826                 case 0xD7:
07827                 case 0xD8:
07828                 case 0xD9:
07829                 case 0xDA:
07830                 case 0xDB:
07831                 case 0xDC:
07832                 case 0xDD:
07833                 case 0xDE:
07834                 case 0xDF:
07835                 {
07836                     if (JSON_HEDLEY_UNLIKELY(!next_byte_in_range({0x80, 0xBF})))
07837                     {
07838                         return token_type::parse_error;
07839                     }
07840                     break;
07841                 }
07842
07843                 // U+0800..U+0FFF: bytes E0 A0..BF 80..BF
```

```
07844                 case 0xE0:
07845                 {
07846                     if (JSON_HEDLEY_UNLIKELY(!(next_byte_in_range({0xA0, 0xBF, 0x80, 0xBF}))))
07847                     {
07848                         return token_type::parse_error;
07849                     }
07850                     break;
07851                 }
07852
07853                 // U+1000..U+CFFF: bytes E1..EC 80..BF 80..BF
07854                 // U+E000..U+FFFF: bytes EE..EF 80..BF 80..BF
07855                 case 0xE1:
07856                 case 0xE2:
07857                 case 0xE3:
07858                 case 0xE4:
07859                 case 0xE5:
07860                 case 0xE6:
07861                 case 0xE7:
07862                 case 0xE8:
07863                 case 0xE9:
07864                 case 0xEA:
07865                 case 0xEB:
07866                 case 0xEC:
07867                 case 0xEE:
07868                 case 0xEF:
07869                 {
07870                     if (JSON_HEDLEY_UNLIKELY(!(next_byte_in_range({0x80, 0xBF, 0x80, 0xBF}))))
07871                     {
07872                         return token_type::parse_error;
07873                     }
07874                     break;
07875                 }
07876
07877                 // U+D000..U+D7FF: bytes ED 80..9F 80..BF
07878                 case 0xED:
07879                 {
07880                     if (JSON_HEDLEY_UNLIKELY(!(next_byte_in_range({0x80, 0x9F, 0x80, 0xBF}))))
07881                     {
07882                         return token_type::parse_error;
07883                     }
07884                     break;
07885                 }
07886
07887                 // U+10000..U+3FFFF F0 90..BF 80..BF 80..BF
07888                 case 0xF0:
07889                 {
07890                     if (JSON_HEDLEY_UNLIKELY(!(next_byte_in_range({0x90, 0xBF, 0x80, 0xBF, 0x80,
      0xBF}))))
07891                     {
07892                         return token_type::parse_error;
07893                     }
07894                     break;
07895                 }
07896
07897                 // U+40000..U+FFFFF F1..F3 80..BF 80..BF 80..BF
07898                 case 0xF1:
07899                 case 0xF2:
07900                 case 0xF3:
07901                 {
07902                     if (JSON_HEDLEY_UNLIKELY(!(next_byte_in_range({0x80, 0xBF, 0x80, 0xBF, 0x80,
      0xBF}))))
07903                     {
07904                         return token_type::parse_error;
07905                     }
07906                     break;
07907                 }
07908
07909                 // U+100000..U+10FFFF F4 80..8F 80..BF 80..BF
07910                 case 0xF4:
07911                 {
07912                     if (JSON_HEDLEY_UNLIKELY(!(next_byte_in_range({0x80, 0x8F, 0x80, 0xBF, 0x80,
      0xBF}))))
07913                     {
07914                         return token_type::parse_error;
07915                     }
07916                     break;
07917                 }
07918
07919                 // the remaining bytes (80..C1 and F5..FF) are ill-formed
07920                 default:
07921                 {
07922                     error_message = "invalid string: ill-formed UTF-8 byte";
07923                     return token_type::parse_error;
07924                 }
07925             }
07926         }
07927     }
```

```
07928
07933     bool scan_comment()
07934     {
07935         switch (get())
07936         {
07937             // single-line comments skip input until a newline or EOF is read
07938             case '/':
07939             {
07940                 while (true)
07941                 {
07942                     switch (get())
07943                     {
07944                         case '\n':
07945                         case '\r':
07946                         case char_traits<char_type>::eof():
07947                         case '\0':
07948                             return true;
07949
07950                         default:
07951                             break;
07952                     }
07953                 }
07954             }
07955
07956             // multi-line comments skip input until */ is read
07957             case '*':
07958             {
07959                 while (true)
07960                 {
07961                     switch (get())
07962                     {
07963                         case char_traits<char_type>::eof():
07964                         case '\0':
07965                         {
07966                             error_message = "invalid comment; missing closing '*/'";
07967                             return false;
07968                         }
07969
07970                         case '*':
07971                         {
07972                             switch (get())
07973                             {
07974                                 case '/':
07975                                     return true;
07976
07977                                 default:
07978                                 {
07979                                     unget();
07980                                     continue;
07981                                 }
07982                             }
07983                         }
07984
07985                         default:
07986                             continue;
07987                     }
07988                 }
07989             }
07990
07991             // unexpected character after reading '/'
07992             default:
07993             {
07994                 error_message = "invalid comment; expecting '/' or '*' after '/'";
07995                 return false;
07996             }
07997         }
07998     }
07999
08000     JSON_HEDLEY_NON_NULL(2)
08001     static void strtof(float& f, const char* str, char** endptr) noexcept
08002     {
08003         f = std::strtof(str, endptr);
08004     }
08005
08006     JSON_HEDLEY_NON_NULL(2)
08007     static void strtof(double& f, const char* str, char** endptr) noexcept
08008     {
08009         f = std::strtod(str, endptr);
08010     }
08011
08012     JSON_HEDLEY_NON_NULL(2)
08013     static void strtof(long double& f, const char* str, char** endptr) noexcept
08014     {
08015         f = std::strtold(str, endptr);
08016     }
08017
08058     token_type scan_number()  // lgtm [cpp/use-of-goto] `goto` is used in this function to implement
```

```
      the number-parsing state machine described above. By design, any finite input will eventually reach
      the "done" state or return token_type::parse_error. In each intermediate state, 1 byte of the input is
      appended to the token_buffer vector, and only the already initialized variables token_buffer,
      number_type, and error_message are manipulated.
08059     {
08060         // reset token_buffer to store the number's bytes
08061         reset();
08062
08063         // the type of the parsed number; initially set to unsigned; will be
08064         // changed if minus sign, decimal point, or exponent is read
08065         token_type number_type = token_type::value_unsigned;
08066
08067         // state (init): we just found out we need to scan a number
08068         switch (current)
08069         {
08070             case '-':
08071             {
08072                 add(current);
08073                 goto scan_number_minus;
08074             }
08075
08076             case '0':
08077             {
08078                 add(current);
08079                 goto scan_number_zero;
08080             }
08081
08082             case '1':
08083             case '2':
08084             case '3':
08085             case '4':
08086             case '5':
08087             case '6':
08088             case '7':
08089             case '8':
08090             case '9':
08091             {
08092                 add(current);
08093                 goto scan_number_any1;
08094             }
08095
08096             // all other characters are rejected outside scan_number()
08097             default:            // LCOV_EXCL_LINE
08098                 JSON_ASSERT(false); // NOLINT(cert-dcl03-c,hicpp-static-assert,misc-static-assert)
      LCOV_EXCL_LINE
08099         }
08100
08101 scan_number_minus:
08102         // state: we just parsed a leading minus sign
08103         number_type = token_type::value_integer;
08104         switch (get())
08105         {
08106             case '0':
08107             {
08108                 add(current);
08109                 goto scan_number_zero;
08110             }
08111
08112             case '1':
08113             case '2':
08114             case '3':
08115             case '4':
08116             case '5':
08117             case '6':
08118             case '7':
08119             case '8':
08120             case '9':
08121             {
08122                 add(current);
08123                 goto scan_number_any1;
08124             }
08125
08126             default:
08127             {
08128                 error_message = "invalid number; expected digit after '-'";
08129                 return token_type::parse_error;
08130             }
08131         }
08132
08133 scan_number_zero:
08134         // state: we just parse a zero (maybe with a leading minus sign)
08135         switch (get())
08136         {
08137             case '.':
08138             {
08139                 add(decimal_point_char);
08140                 decimal_point_position = token_buffer.size() - 1;
```

```
08141                 goto scan_number_decimal1;
08142             }
08143
08144             case 'e':
08145             case 'E':
08146             {
08147                 add(current);
08148                 goto scan_number_exponent;
08149             }
08150
08151             default:
08152                 goto scan_number_done;
08153         }
08154
08155 scan_number_any1:
08156         // state: we just parsed a number 0-9 (maybe with a leading minus sign)
08157         switch (get())
08158         {
08159             case '0':
08160             case '1':
08161             case '2':
08162             case '3':
08163             case '4':
08164             case '5':
08165             case '6':
08166             case '7':
08167             case '8':
08168             case '9':
08169             {
08170                 add(current);
08171                 goto scan_number_any1;
08172             }
08173
08174             case '.':
08175             {
08176                 add(decimal_point_char);
08177                 decimal_point_position = token_buffer.size() - 1;
08178                 goto scan_number_decimal1;
08179             }
08180
08181             case 'e':
08182             case 'E':
08183             {
08184                 add(current);
08185                 goto scan_number_exponent;
08186             }
08187
08188             default:
08189                 goto scan_number_done;
08190         }
08191
08192 scan_number_decimal1:
08193         // state: we just parsed a decimal point
08194         number_type = token_type::value_float;
08195         switch (get())
08196         {
08197             case '0':
08198             case '1':
08199             case '2':
08200             case '3':
08201             case '4':
08202             case '5':
08203             case '6':
08204             case '7':
08205             case '8':
08206             case '9':
08207             {
08208                 add(current);
08209                 goto scan_number_decimal2;
08210             }
08211
08212             default:
08213             {
08214                 error_message = "invalid number; expected digit after '.'";
08215                 return token_type::parse_error;
08216             }
08217         }
08218
08219 scan_number_decimal2:
08220         // we just parsed at least one number after a decimal point
08221         switch (get())
08222         {
08223             case '0':
08224             case '1':
08225             case '2':
08226             case '3':
08227             case '4':
```

```
08228             case '5':
08229             case '6':
08230             case '7':
08231             case '8':
08232             case '9':
08233             {
08234                 add(current);
08235                 goto scan_number_decimal2;
08236             }
08237
08238             case 'e':
08239             case 'E':
08240             {
08241                 add(current);
08242                 goto scan_number_exponent;
08243             }
08244
08245             default:
08246                 goto scan_number_done;
08247         }
08248
08249 scan_number_exponent:
08250         // we just parsed an exponent
08251         number_type = token_type::value_float;
08252         switch (get())
08253         {
08254             case '+':
08255             case '-':
08256             {
08257                 add(current);
08258                 goto scan_number_sign;
08259             }
08260
08261             case '0':
08262             case '1':
08263             case '2':
08264             case '3':
08265             case '4':
08266             case '5':
08267             case '6':
08268             case '7':
08269             case '8':
08270             case '9':
08271             {
08272                 add(current);
08273                 goto scan_number_any2;
08274             }
08275
08276             default:
08277             {
08278                 error_message =
08279                     "invalid number; expected '+', '-', or digit after exponent";
08280                 return token_type::parse_error;
08281             }
08282         }
08283
08284 scan_number_sign:
08285         // we just parsed an exponent sign
08286         switch (get())
08287         {
08288             case '0':
08289             case '1':
08290             case '2':
08291             case '3':
08292             case '4':
08293             case '5':
08294             case '6':
08295             case '7':
08296             case '8':
08297             case '9':
08298             {
08299                 add(current);
08300                 goto scan_number_any2;
08301             }
08302
08303             default:
08304             {
08305                 error_message = "invalid number; expected digit after exponent sign";
08306                 return token_type::parse_error;
08307             }
08308         }
08309
08310 scan_number_any2:
08311         // we just parsed a number after the exponent or exponent sign
08312         switch (get())
08313         {
08314             case '0':
```

```
08315             case '1':
08316             case '2':
08317             case '3':
08318             case '4':
08319             case '5':
08320             case '6':
08321             case '7':
08322             case '8':
08323             case '9':
08324             {
08325                 add(current);
08326                 goto scan_number_any2;
08327             }
08328
08329             default:
08330                 goto scan_number_done;
08331         }
08332
08333 scan_number_done:
08334         // unget the character after the number (we only read it to know that
08335         // we are done scanning a number)
08336         unget();
08337
08338         char* endptr = nullptr; //
      NOLINT(misc-const-correctness,cppcoreguidelines-pro-type-vararg,hicpp-vararg)
08339         errno = 0;
08340
08341         // try to parse integers first and fall back to floats
08342         if (number_type == token_type::value_unsigned)
08343         {
08344             const auto x = std::strtoull(token_buffer.data(), &endptr, 10);
08345
08346             // we checked the number format before
08347             JSON_ASSERT(endptr == token_buffer.data() + token_buffer.size());
08348
08349             if (errno != ERANGE)
08350             {
08351                 value_unsigned = static_cast<number_unsigned_t>(x);
08352                 if (value_unsigned == x)
08353                 {
08354                     return token_type::value_unsigned;
08355                 }
08356             }
08357         }
08358         else if (number_type == token_type::value_integer)
08359         {
08360             const auto x = std::strtoll(token_buffer.data(), &endptr, 10);
08361
08362             // we checked the number format before
08363             JSON_ASSERT(endptr == token_buffer.data() + token_buffer.size());
08364
08365             if (errno != ERANGE)
08366             {
08367                 value_integer = static_cast<number_integer_t>(x);
08368                 if (value_integer == x)
08369                 {
08370                     return token_type::value_integer;
08371                 }
08372             }
08373         }
08374
08375         // this code is reached if we parse a floating-point number or if an
08376         // integer conversion above failed
08377         strtof(value_float, token_buffer.data(), &endptr);
08378
08379         // we checked the number format before
08380         JSON_ASSERT(endptr == token_buffer.data() + token_buffer.size());
08381
08382         return token_type::value_float;
08383     }
08384
08390     JSON_HEDLEY_NON_NULL(2)
08391     token_type scan_literal(const char_type* literal_text, const std::size_t length,
08392                             token_type return_type)
08393     {
08394         JSON_ASSERT(char_traits<char_type>::to_char_type(current) == literal_text[0]);
08395         for (std::size_t i = 1; i < length; ++i)
08396         {
08397             if (JSON_HEDLEY_UNLIKELY(char_traits<char_type>::to_char_type(get()) != literal_text[i]))
08398             {
08399                 error_message = "invalid literal";
08400                 return token_type::parse_error;
08401             }
08402         }
08403         return return_type;
08404     }
08405
```

```
08407     // input management
08409
08411     void reset() noexcept
08412     {
08413         token_buffer.clear();
08414         token_string.clear();
08415         decimal_point_position = std::string::npos;
08416         token_string.push_back(char_traits<char_type>::to_char_type(current));
08417     }
08418
08419     /*
08420     @brief get next character from the input
08421
08422     This function provides the interface to the used input adapter. It does
08423     not throw in case the input reached EOF, but returns a
08424     `char_traits<char>::eof()` in that case.  Stores the scanned characters
08425     for use in error messages.
08426
08427     @return character read from the input
08428     */
08429     char_int_type get()
08430     {
08431         ++position.chars_read_total;
08432         ++position.chars_read_current_line;
08433
08434         if (next_unget)
08435         {
08436             // only reset the next_unget variable and work with current
08437             next_unget = false;
08438         }
08439         else
08440         {
08441             current = ia.get_character();
08442         }
08443
08444         if (JSON_HEDLEY_LIKELY(current != char_traits<char_type>::eof()))
08445         {
08446             token_string.push_back(char_traits<char_type>::to_char_type(current));
08447         }
08448
08449         if (current == '\n')
08450         {
08451             ++position.lines_read;
08452             position.chars_read_current_line = 0;
08453         }
08454
08455         return current;
08456     }
08457
08466     void unget()
08467     {
08468         next_unget = true;
08469
08470         --position.chars_read_total;
08471
08472         // in case we "unget" a newline, we have to also decrement the lines_read
08473         if (position.chars_read_current_line == 0)
08474         {
08475             if (position.lines_read > 0)
08476             {
08477                 --position.lines_read;
08478             }
08479         }
08480         else
08481         {
08482             --position.chars_read_current_line;
08483         }
08484
08485         if (JSON_HEDLEY_LIKELY(current != char_traits<char_type>::eof()))
08486         {
08487             JSON_ASSERT(!token_string.empty());
08488             token_string.pop_back();
08489         }
08490     }
08491
08493     void add(char_int_type c)
08494     {
08495         token_buffer.push_back(static_cast<typename string_t::value_type>(c));
08496     }
08497
08498   public:
08500     // value getters
08502
08504     constexpr number_integer_t get_number_integer() const noexcept
08505     {
08506         return value_integer;
08507     }
```

```
08508
08510     constexpr number_unsigned_t get_number_unsigned() const noexcept
08511     {
08512         return value_unsigned;
08513     }
08514
08516     constexpr number_float_t get_number_float() const noexcept
08517     {
08518         return value_float;
08519     }
08520
08522     string_t& get_string()
08523     {
08524         // translate decimal points from locale back to '.' (#4084)
08525         if (decimal_point_char != '.' && decimal_point_position != std::string::npos)
08526         {
08527             token_buffer[decimal_point_position] = '.';
08528         }
08529         return token_buffer;
08530     }
08531
08533     // diagnostics
08535
08537     constexpr position_t get_position() const noexcept
08538     {
08539         return position;
08540     }
08541
08545     std::string get_token_string() const
08546     {
08547         // escape control characters
08548         std::string result;
08549         for (const auto c : token_string)
08550         {
08551             if (static_cast<unsigned char>(c) <= '\x1F')
08552             {
08553                 // escape control characters
08554                 std::array<char, 9> cs{{}};
08555                 static_cast<void>((std::snprintf)(cs.data(), cs.size(), "<U+%.4X>",
      static_cast<unsigned char>(c))); // NOLINT(cppcoreguidelines-pro-type-vararg,hicpp-vararg)
08556                 result += cs.data();
08557             }
08558             else
08559             {
08560                 // add character as is
08561                 result.push_back(static_cast<std::string::value_type>(c));
08562             }
08563         }
08564
08565         return result;
08566     }
08567
08569     JSON_HEDLEY_RETURNS_NON_NULL
08570     constexpr const char* get_error_message() const noexcept
08571     {
08572         return error_message;
08573     }
08574
08576     // actual scanner
08578
08583     bool skip_bom()
08584     {
08585         if (get() == 0xEF)
08586         {
08587             // check if we completely parse the BOM
08588             return get() == 0xBB && get() == 0xBF;
08589         }
08590
08591         // the first character is not the beginning of the BOM; unget it to
08592         // process is later
08593         unget();
08594         return true;
08595     }
08596
08597     void skip_whitespace()
08598     {
08599         do
08600         {
08601             get();
08602         }
08603         while (current == ' ' || current == '\t' || current == '\n' || current == '\r');
08604     }
08605
08606     token_type scan()
08607     {
08608         // initially, skip the BOM
08609         if (position.chars_read_total == 0 && !skip_bom())
```

```
08610         {
08611             error_message = "invalid BOM; must be 0xEF 0xBB 0xBF if given";
08612             return token_type::parse_error;
08613         }
08614 
08615         // read the next character and ignore whitespace
08616         skip_whitespace();
08617 
08618         // ignore comments
08619         while (ignore_comments && current == '/')
08620         {
08621             if (!scan_comment())
08622             {
08623                 return token_type::parse_error;
08624             }
08625 
08626             // skip following whitespace
08627             skip_whitespace();
08628         }
08629 
08630         switch (current)
08631         {
08632             // structural characters
08633             case '[':
08634                 return token_type::begin_array;
08635             case ']':
08636                 return token_type::end_array;
08637             case '{':
08638                 return token_type::begin_object;
08639             case '}':
08640                 return token_type::end_object;
08641             case ':':
08642                 return token_type::name_separator;
08643             case ',':
08644                 return token_type::value_separator;
08645 
08646             // literals
08647             case 't':
08648             {
08649                 std::array<char_type, 4> true_literal = {{static_cast<char_type>('t'),
      static_cast<char_type>('r'), static_cast<char_type>('u'), static_cast<char_type>('e')}};
08650                 return scan_literal(true_literal.data(), true_literal.size(),
      token_type::literal_true);
08651             }
08652             case 'f':
08653             {
08654                 std::array<char_type, 5> false_literal = {{static_cast<char_type>('f'),
      static_cast<char_type>('a'), static_cast<char_type>('l'), static_cast<char_type>('s'),
      static_cast<char_type>('e')}};
08655                 return scan_literal(false_literal.data(), false_literal.size(),
      token_type::literal_false);
08656             }
08657             case 'n':
08658             {
08659                 std::array<char_type, 4> null_literal = {{static_cast<char_type>('n'),
      static_cast<char_type>('u'), static_cast<char_type>('l'), static_cast<char_type>('l')}};
08660                 return scan_literal(null_literal.data(), null_literal.size(),
      token_type::literal_null);
08661             }
08662 
08663             // string
08664             case '\"':
08665                 return scan_string();
08666 
08667             // number
08668             case '-':
08669             case '0':
08670             case '1':
08671             case '2':
08672             case '3':
08673             case '4':
08674             case '5':
08675             case '6':
08676             case '7':
08677             case '8':
08678             case '9':
08679                 return scan_number();
08680 
08681             // end of input (the null byte is needed when parsing from
08682             // string literals)
08683             case '\0':
08684             case char_traits<char_type>::eof():
08685                 return token_type::end_of_input;
08686 
08687             // error
08688             default:
08689                 error_message = "invalid literal";
```

```
08690                 return token_type::parse_error;
08691             }
08692         }
08693
08694       private:
08696         InputAdapterType ia;
08697
08699         const bool ignore_comments = false;
08700
08702         char_int_type current = char_traits<char_type>::eof();
08703
08705         bool next_unget = false;
08706
08708         position_t position {};
08709
08711         std::vector<char_type> token_string {};
08712
08714         string_t token_buffer {};
08715
08717         const char* error_message = "";
08718
08719         // number values
08720         number_integer_t value_integer = 0;
08721         number_unsigned_t value_unsigned = 0;
08722         number_float_t value_float = 0;
08723
08725         const char_int_type decimal_point_char = '.';
08727         std::size_t decimal_point_position = std::string::npos;
08728 };
08729
08730 }  // namespace detail
08731 NLOHMANN_JSON_NAMESPACE_END
08732
08733 // #include <nlohmann/detail/macro_scope.hpp>
08734
08735 // #include <nlohmann/detail/string_concat.hpp>
08736
08737 NLOHMANN_JSON_NAMESPACE_BEGIN
08738
08747 template<typename BasicJsonType>
08748 struct json_sax
08749 {
08750     using number_integer_t = typename BasicJsonType::number_integer_t;
08751     using number_unsigned_t = typename BasicJsonType::number_unsigned_t;
08752     using number_float_t = typename BasicJsonType::number_float_t;
08753     using string_t = typename BasicJsonType::string_t;
08754     using binary_t = typename BasicJsonType::binary_t;
08755
08760     virtual bool null() = 0;
08761
08767     virtual bool boolean(bool val) = 0;
08768
08774     virtual bool number_integer(number_integer_t val) = 0;
08775
08781     virtual bool number_unsigned(number_unsigned_t val) = 0;
08782
08789     virtual bool number_float(number_float_t val, const string_t& s) = 0;
08790
08797     virtual bool string(string_t& val) = 0;
08798
08805     virtual bool binary(binary_t& val) = 0;
08806
08813     virtual bool start_object(std::size_t elements) = 0;
08814
08821     virtual bool key(string_t& val) = 0;
08822
08827     virtual bool end_object() = 0;
08828
08835     virtual bool start_array(std::size_t elements) = 0;
08836
08841     virtual bool end_array() = 0;
08842
08850     virtual bool parse_error(std::size_t position,
08851                              const std::string& last_token,
08852                              const detail::exception& ex) = 0;
08853
08854     json_sax() = default;
08855     json_sax(const json_sax&) = default;
08856     json_sax(json_sax&&) noexcept = default;
08857     json_sax& operator=(const json_sax&) = default;
08858     json_sax& operator=(json_sax&&) noexcept = default;
08859     virtual ~json_sax() = default;
08860 };
08861
08862 namespace detail
08863 {
08864 constexpr std::size_t unknown_size()
```

```
08865 {
08866     return (std::numeric_limits<std::size_t>::max)();
08867 }
08868
08882 template<typename BasicJsonType, typename InputAdapterType>
08883 class json_sax_dom_parser
08884 {
08885   public:
08886     using number_integer_t = typename BasicJsonType::number_integer_t;
08887     using number_unsigned_t = typename BasicJsonType::number_unsigned_t;
08888     using number_float_t = typename BasicJsonType::number_float_t;
08889     using string_t = typename BasicJsonType::string_t;
08890     using binary_t = typename BasicJsonType::binary_t;
08891     using lexer_t = lexer<BasicJsonType, InputAdapterType>;
08892
08898     explicit json_sax_dom_parser(BasicJsonType& r, const bool allow_exceptions_ = true, lexer_t*
      lexer_ = nullptr)
08899         : root(r), allow_exceptions(allow_exceptions_), m_lexer_ref(lexer_)
08900     {}
08901
08902     // make class move-only
08903     json_sax_dom_parser(const json_sax_dom_parser&) = delete;
08904     json_sax_dom_parser(json_sax_dom_parser&&) = default; //
      NOLINT(hicpp-noexcept-move,performance-noexcept-move-constructor)
08905     json_sax_dom_parser& operator=(const json_sax_dom_parser&) = delete;
08906     json_sax_dom_parser& operator=(json_sax_dom_parser&&) = default; //
      NOLINT(hicpp-noexcept-move,performance-noexcept-move-constructor)
08907     ~json_sax_dom_parser() = default;
08908
08909     bool null()
08910     {
08911         handle_value(nullptr);
08912         return true;
08913     }
08914
08915     bool boolean(bool val)
08916     {
08917         handle_value(val);
08918         return true;
08919     }
08920
08921     bool number_integer(number_integer_t val)
08922     {
08923         handle_value(val);
08924         return true;
08925     }
08926
08927     bool number_unsigned(number_unsigned_t val)
08928     {
08929         handle_value(val);
08930         return true;
08931     }
08932
08933     bool number_float(number_float_t val, const string_t& /*unused*/)
08934     {
08935         handle_value(val);
08936         return true;
08937     }
08938
08939     bool string(string_t& val)
08940     {
08941         handle_value(val);
08942         return true;
08943     }
08944
08945     bool binary(binary_t& val)
08946     {
08947         handle_value(std::move(val));
08948         return true;
08949     }
08950
08951     bool start_object(std::size_t len)
08952     {
08953         ref_stack.push_back(handle_value(BasicJsonType::value_t::object));
08954
08955 #if JSON_DIAGNOSTIC_POSITIONS
08956         // Manually set the start position of the object here.
08957         // Ensure this is after the call to handle_value to ensure correct start position.
08958         if (m_lexer_ref)
08959         {
08960             // Lexer has read the first character of the object, so
08961             // subtract 1 from the position to get the correct start position.
08962             ref_stack.back()->start_position = m_lexer_ref->get_position() - 1;
08963         }
08964 #endif
08965
08966         if (JSON_HEDLEY_UNLIKELY(len != detail::unknown_size() && len > ref_stack.back()->max_size()))
```

```
08967         {
08968             JSON_THROW(out_of_range::create(408, concat("excessive object size: ",
      std::to_string(len)), ref_stack.back()));
08969         }
08970
08971         return true;
08972     }
08973
08974     bool key(string_t& val)
08975     {
08976         JSON_ASSERT(!ref_stack.empty());
08977         JSON_ASSERT(ref_stack.back()->is_object());
08978
08979         // add null at the given key and store the reference for later
08980         object_element = &(ref_stack.back()->m_data.m_value.object->operator[](val));
08981         return true;
08982     }
08983
08984     bool end_object()
08985     {
08986         JSON_ASSERT(!ref_stack.empty());
08987         JSON_ASSERT(ref_stack.back()->is_object());
08988
08989 #if JSON_DIAGNOSTIC_POSITIONS
08990         if (m_lexer_ref)
08991         {
08992             // Lexer's position is past the closing brace, so set that as the end position.
08993             ref_stack.back()->end_position = m_lexer_ref->get_position();
08994         }
08995 #endif
08996
08997         ref_stack.back()->set_parents();
08998         ref_stack.pop_back();
08999         return true;
09000     }
09001
09002     bool start_array(std::size_t len)
09003     {
09004         ref_stack.push_back(handle_value(BasicJsonType::value_t::array));
09005
09006 #if JSON_DIAGNOSTIC_POSITIONS
09007         // Manually set the start position of the array here.
09008         // Ensure this is after the call to handle_value to ensure correct start position.
09009         if (m_lexer_ref)
09010         {
09011             ref_stack.back()->start_position = m_lexer_ref->get_position() - 1;
09012         }
09013 #endif
09014
09015         if (JSON_HEDLEY_UNLIKELY(len != detail::unknown_size() && len > ref_stack.back()->max_size()))
09016         {
09017             JSON_THROW(out_of_range::create(408, concat("excessive array size: ",
      std::to_string(len)), ref_stack.back()));
09018         }
09019
09020         return true;
09021     }
09022
09023     bool end_array()
09024     {
09025         JSON_ASSERT(!ref_stack.empty());
09026         JSON_ASSERT(ref_stack.back()->is_array());
09027
09028 #if JSON_DIAGNOSTIC_POSITIONS
09029         if (m_lexer_ref)
09030         {
09031             // Lexer's position is past the closing bracket, so set that as the end position.
09032             ref_stack.back()->end_position = m_lexer_ref->get_position();
09033         }
09034 #endif
09035
09036         ref_stack.back()->set_parents();
09037         ref_stack.pop_back();
09038         return true;
09039     }
09040
09041     template<class Exception>
09042     bool parse_error(std::size_t /*unused*/, const std::string& /*unused*/,
09043                      const Exception& ex)
09044     {
09045         errored = true;
09046         static_cast<void>(ex);
09047         if (allow_exceptions)
09048         {
09049             JSON_THROW(ex);
09050         }
09051         return false;
```

```
09052     }
09053
09054     constexpr bool is_errored() const
09055     {
09056         return errored;
09057     }
09058
09059   private:
09060
09061 #if JSON_DIAGNOSTIC_POSITIONS
09062     void handle_diagnostic_positions_for_json_value(BasicJsonType& v)
09063     {
09064         if (m_lexer_ref)
09065         {
09066             // Lexer has read past the current field value, so set the end position to the current
      position.
09067             // The start position will be set below based on the length of the string representation
09068             // of the value.
09069             v.end_position = m_lexer_ref->get_position();
09070
09071             switch (v.type())
09072             {
09073                 case value_t::boolean:
09074                 {
09075                     // 4 and 5 are the string length of "true" and "false"
09076                     v.start_position = v.end_position - (v.m_data.m_value.boolean ? 4 : 5);
09077                     break;
09078                 }
09079
09080                 case value_t::null:
09081                 {
09082                     // 4 is the string length of "null"
09083                     v.start_position = v.end_position - 4;
09084                     break;
09085                 }
09086
09087                 case value_t::string:
09088                 {
09089                     // include the length of the quotes, which is 2
09090                     v.start_position = v.end_position - v.m_data.m_value.string->size() - 2;
09091                     break;
09092                 }
09093
09094                 // As we handle the start and end positions for values created during parsing,
09095                 // we do not expect the following value type to be called. Regardless, set the
      positions
09096                 // in case this is created manually or through a different constructor. Exclude from
      lcov
09097                 // since the exact condition of this switch is esoteric.
09098                 // LCOV_EXCL_START
09099                 case value_t::discarded:
09100                 {
09101                     v.end_position = std::string::npos;
09102                     v.start_position = v.end_position;
09103                     break;
09104                 }
09105                 // LCOV_EXCL_STOP
09106                 case value_t::binary:
09107                 case value_t::number_integer:
09108                 case value_t::number_unsigned:
09109                 case value_t::number_float:
09110                 {
09111                     v.start_position = v.end_position - m_lexer_ref->get_string().size();
09112                     break;
09113                 }
09114                 case value_t::object:
09115                 case value_t::array:
09116                 {
09117                     // object and array are handled in start_object() and start_array() handlers
09118                     // skip setting the values here.
09119                     break;
09120                 }
09121                 default: // LCOV_EXCL_LINE
09122                     // Handle all possible types discretely, default handler should never be reached.
09123                     JSON_ASSERT(false); //
      NOLINT(cert-dcl03-c,hicpp-static-assert,misc-static-assert,-warnings-as-errors) LCOV_EXCL_LINE
09124             }
09125         }
09126     }
09127 #endif
09128
09135     template<typename Value>
09136     JSON_HEDLEY_RETURNS_NON_NULL
09137     BasicJsonType* handle_value(Value&& v)
09138     {
09139         if (ref_stack.empty())
09140         {
```

```
09141             root = BasicJsonType(std::forward<Value>(v));
09142
09143 #if JSON_DIAGNOSTIC_POSITIONS
09144             handle_diagnostic_positions_for_json_value(root);
09145 #endif
09146
09147             return &root;
09148         }
09149
09150         JSON_ASSERT(ref_stack.back()->is_array() || ref_stack.back()->is_object());
09151
09152         if (ref_stack.back()->is_array())
09153         {
09154             ref_stack.back()->m_data.m_value.array->emplace_back(std::forward<Value>(v));
09155
09156 #if JSON_DIAGNOSTIC_POSITIONS
09157
      handle_diagnostic_positions_for_json_value(ref_stack.back()->m_data.m_value.array->back());
09158 #endif
09159
09160             return &(ref_stack.back()->m_data.m_value.array->back());
09161         }
09162
09163         JSON_ASSERT(ref_stack.back()->is_object());
09164         JSON_ASSERT(object_element);
09165         *object_element = BasicJsonType(std::forward<Value>(v));
09166
09167 #if JSON_DIAGNOSTIC_POSITIONS
09168         handle_diagnostic_positions_for_json_value(*object_element);
09169 #endif
09170
09171         return object_element;
09172     }
09173
09175     BasicJsonType& root;
09177     std::vector<BasicJsonType*> ref_stack {};
09179     BasicJsonType* object_element = nullptr;
09181     bool errored = false;
09183     const bool allow_exceptions = true;
09185     lexer_t* m_lexer_ref = nullptr;
09186 };
09187
09188 template<typename BasicJsonType, typename InputAdapterType>
09189 class json_sax_dom_callback_parser
09190 {
09191   public:
09192     using number_integer_t = typename BasicJsonType::number_integer_t;
09193     using number_unsigned_t = typename BasicJsonType::number_unsigned_t;
09194     using number_float_t = typename BasicJsonType::number_float_t;
09195     using string_t = typename BasicJsonType::string_t;
09196     using binary_t = typename BasicJsonType::binary_t;
09197     using parser_callback_t = typename BasicJsonType::parser_callback_t;
09198     using parse_event_t = typename BasicJsonType::parse_event_t;
09199     using lexer_t = lexer<BasicJsonType, InputAdapterType>;
09200
09201     json_sax_dom_callback_parser(BasicJsonType& r,
09202                                  parser_callback_t cb,
09203                                  const bool allow_exceptions_ = true,
09204                                  lexer_t* lexer_ = nullptr)
09205         : root(r), callback(std::move(cb)), allow_exceptions(allow_exceptions_), m_lexer_ref(lexer_)
09206     {
09207         keep_stack.push_back(true);
09208     }
09209
09210     // make class move-only
09211     json_sax_dom_callback_parser(const json_sax_dom_callback_parser&) = delete;
09212     json_sax_dom_callback_parser(json_sax_dom_callback_parser&&) = default; //
      NOLINT(hicpp-noexcept-move,performance-noexcept-move-constructor)
09213     json_sax_dom_callback_parser& operator=(const json_sax_dom_callback_parser&) = delete;
09214     json_sax_dom_callback_parser& operator=(json_sax_dom_callback_parser&&) = default; //
      NOLINT(hicpp-noexcept-move,performance-noexcept-move-constructor)
09215     ~json_sax_dom_callback_parser() = default;
09216
09217     bool null()
09218     {
09219         handle_value(nullptr);
09220         return true;
09221     }
09222
09223     bool boolean(bool val)
09224     {
09225         handle_value(val);
09226         return true;
09227     }
09228
09229     bool number_integer(number_integer_t val)
09230     {
```

```
09231         handle_value(val);
09232         return true;
09233     }
09234
09235     bool number_unsigned(number_unsigned_t val)
09236     {
09237         handle_value(val);
09238         return true;
09239     }
09240
09241     bool number_float(number_float_t val, const string_t& /*unused*/)
09242     {
09243         handle_value(val);
09244         return true;
09245     }
09246
09247     bool string(string_t& val)
09248     {
09249         handle_value(val);
09250         return true;
09251     }
09252
09253     bool binary(binary_t& val)
09254     {
09255         handle_value(std::move(val));
09256         return true;
09257     }
09258
09259     bool start_object(std::size_t len)
09260     {
09261         // check callback for object start
09262         const bool keep = callback(static_cast<int>(ref_stack.size()), parse_event_t::object_start,
      discarded);
09263         keep_stack.push_back(keep);
09264
09265         auto val = handle_value(BasicJsonType::value_t::object, true);
09266         ref_stack.push_back(val.second);
09267
09268         if (ref_stack.back())
09269         {
09270
09271 #if JSON_DIAGNOSTIC_POSITIONS
09272             // Manually set the start position of the object here.
09273             // Ensure this is after the call to handle_value to ensure correct start position.
09274             if (m_lexer_ref)
09275             {
09276                 // Lexer has read the first character of the object, so
09277                 // subtract 1 from the position to get the correct start position.
09278                 ref_stack.back()->start_position = m_lexer_ref->get_position() - 1;
09279             }
09280 #endif
09281
09282             // check object limit
09283             if (JSON_HEDLEY_UNLIKELY(len != detail::unknown_size() && len >
      ref_stack.back()->max_size()))
09284             {
09285                 JSON_THROW(out_of_range::create(408, concat("excessive object size: ",
      std::to_string(len)), ref_stack.back()));
09286             }
09287         }
09288         return true;
09289     }
09290
09291     bool key(string_t& val)
09292     {
09293         BasicJsonType k = BasicJsonType(val);
09294
09295         // check callback for the key
09296         const bool keep = callback(static_cast<int>(ref_stack.size()), parse_event_t::key, k);
09297         key_keep_stack.push_back(keep);
09298
09299         // add discarded value at the given key and store the reference for later
09300         if (keep && ref_stack.back())
09301         {
09302             object_element = &(ref_stack.back()->m_data.m_value.object->operator[](val) = discarded);
09303         }
09304
09305         return true;
09306     }
09307
09308     bool end_object()
09309     {
09310         if (ref_stack.back())
09311         {
09312             if (!callback(static_cast<int>(ref_stack.size()) - 1, parse_event_t::object_end,
      *ref_stack.back()))
09313             {
```

```
09314                 // discard object
09315                 *ref_stack.back() = discarded;
09316
09317 #if JSON_DIAGNOSTIC_POSITIONS
09318                 // Set start/end positions for discarded object.
09319                 handle_diagnostic_positions_for_json_value(*ref_stack.back());
09320 #endif
09321             }
09322             else
09323             {
09324
09325 #if JSON_DIAGNOSTIC_POSITIONS
09326                 if (m_lexer_ref)
09327                 {
09328                     // Lexer's position is past the closing brace, so set that as the end position.
09329                     ref_stack.back()->end_position = m_lexer_ref->get_position();
09330                 }
09331 #endif
09332
09333                 ref_stack.back()->set_parents();
09334             }
09335         }
09336
09337         JSON_ASSERT(!ref_stack.empty());
09338         JSON_ASSERT(!keep_stack.empty());
09339         ref_stack.pop_back();
09340         keep_stack.pop_back();
09341
09342         if (!ref_stack.empty() && ref_stack.back() && ref_stack.back()->is_structured())
09343         {
09344             // remove discarded value
09345             for (auto it = ref_stack.back()->begin(); it != ref_stack.back()->end(); ++it)
09346             {
09347                 if (it->is_discarded())
09348                 {
09349                     ref_stack.back()->erase(it);
09350                     break;
09351                 }
09352             }
09353         }
09354
09355         return true;
09356     }
09357
09358     bool start_array(std::size_t len)
09359     {
09360         const bool keep = callback(static_cast<int>(ref_stack.size()), parse_event_t::array_start,
      discarded);
09361         keep_stack.push_back(keep);
09362
09363         auto val = handle_value(BasicJsonType::value_t::array, true);
09364         ref_stack.push_back(val.second);
09365
09366         if (ref_stack.back())
09367         {
09368
09369 #if JSON_DIAGNOSTIC_POSITIONS
09370             // Manually set the start position of the array here.
09371             // Ensure this is after the call to handle_value to ensure correct start position.
09372             if (m_lexer_ref)
09373             {
09374                 // Lexer has read the first character of the array, so
09375                 // subtract 1 from the position to get the correct start position.
09376                 ref_stack.back()->start_position = m_lexer_ref->get_position() - 1;
09377             }
09378 #endif
09379
09380             // check array limit
09381             if (JSON_HEDLEY_UNLIKELY(len != detail::unknown_size() && len >
      ref_stack.back()->max_size()))
09382             {
09383                 JSON_THROW(out_of_range::create(408, concat("excessive array size: ",
      std::to_string(len)), ref_stack.back()));
09384             }
09385         }
09386
09387         return true;
09388     }
09389
09390     bool end_array()
09391     {
09392         bool keep = true;
09393
09394         if (ref_stack.back())
09395         {
09396             keep = callback(static_cast<int>(ref_stack.size()) - 1, parse_event_t::array_end,
      *ref_stack.back());
```

```
09397             if (keep)
09398             {
09399
09400 #if JSON_DIAGNOSTIC_POSITIONS
09401                 if (m_lexer_ref)
09402                 {
09403                     // Lexer's position is past the closing bracket, so set that as the end position.
09404                     ref_stack.back()->end_position = m_lexer_ref->get_position();
09405                 }
09406 #endif
09407
09408                 ref_stack.back()->set_parents();
09409             }
09410             else
09411             {
09412                 // discard array
09413                 *ref_stack.back() = discarded;
09414
09415 #if JSON_DIAGNOSTIC_POSITIONS
09416                 // Set start/end positions for discarded array.
09417                 handle_diagnostic_positions_for_json_value(*ref_stack.back());
09418 #endif
09419             }
09420         }
09421
09422         JSON_ASSERT(!ref_stack.empty());
09423         JSON_ASSERT(!keep_stack.empty());
09424         ref_stack.pop_back();
09425         keep_stack.pop_back();
09426
09427         // remove discarded value
09428         if (!keep && !ref_stack.empty() && ref_stack.back()->is_array())
09429         {
09430             ref_stack.back()->m_data.m_value.array->pop_back();
09431         }
09432
09433         return true;
09434     }
09435
09436     template<class Exception>
09437     bool parse_error(std::size_t /*unused*/, const std::string& /*unused*/,
09438                      const Exception& ex)
09439     {
09440         errored = true;
09441         static_cast<void>(ex);
09442         if (allow_exceptions)
09443         {
09444             JSON_THROW(ex);
09445         }
09446         return false;
09447     }
09448
09449     constexpr bool is_errored() const
09450     {
09451         return errored;
09452     }
09453
09454   private:
09455
09456 #if JSON_DIAGNOSTIC_POSITIONS
09457     void handle_diagnostic_positions_for_json_value(BasicJsonType& v)
09458     {
09459         if (m_lexer_ref)
09460         {
09461             // Lexer has read past the current field value, so set the end position to the current
      position.
09462             // The start position will be set below based on the length of the string representation
09463             // of the value.
09464             v.end_position = m_lexer_ref->get_position();
09465
09466             switch (v.type())
09467             {
09468                 case value_t::boolean:
09469                 {
09470                     // 4 and 5 are the string length of "true" and "false"
09471                     v.start_position = v.end_position - (v.m_data.m_value.boolean ? 4 : 5);
09472                     break;
09473                 }
09474
09475                 case value_t::null:
09476                 {
09477                     // 4 is the string length of "null"
09478                     v.start_position = v.end_position - 4;
09479                     break;
09480                 }
09481
09482                 case value_t::string:
```

```
09483                 {
09484                     // include the length of the quotes, which is 2
09485                     v.start_position = v.end_position - v.m_data.m_value.string->size() - 2;
09486                     break;
09487                 }
09488
09489                 case value_t::discarded:
09490                 {
09491                     v.end_position = std::string::npos;
09492                     v.start_position = v.end_position;
09493                     break;
09494                 }
09495
09496                 case value_t::binary:
09497                 case value_t::number_integer:
09498                 case value_t::number_unsigned:
09499                 case value_t::number_float:
09500                 {
09501                     v.start_position = v.end_position - m_lexer_ref->get_string().size();
09502                     break;
09503                 }
09504
09505                 case value_t::object:
09506                 case value_t::array:
09507                 {
09508                     // object and array are handled in start_object() and start_array() handlers
09509                     // skip setting the values here.
09510                     break;
09511                 }
09512                 default: // LCOV_EXCL_LINE
09513                     // Handle all possible types discretely, default handler should never be reached.
09514                     JSON_ASSERT(false); //
      NOLINT(cert-dcl03-c,hicpp-static-assert,misc-static-assert,-warnings-as-errors) LCOV_EXCL_LINE
09515             }
09516         }
09517     }
09518 #endif
09519
09535     template<typename Value>
09536     std::pair<bool, BasicJsonType*> handle_value(Value&& v, const bool skip_callback = false)
09537     {
09538         JSON_ASSERT(!keep_stack.empty());
09539
09540         // do not handle this value if we know it would be added to a discarded
09541         // container
09542         if (!keep_stack.back())
09543         {
09544             return {false, nullptr};
09545         }
09546
09547         // create value
09548         auto value = BasicJsonType(std::forward<Value>(v));
09549
09550 #if JSON_DIAGNOSTIC_POSITIONS
09551         handle_diagnostic_positions_for_json_value(value);
09552 #endif
09553
09554         // check callback
09555         const bool keep = skip_callback || callback(static_cast<int>(ref_stack.size()),
      parse_event_t::value, value);
09556
09557         // do not handle this value if we just learnt it shall be discarded
09558         if (!keep)
09559         {
09560             return {false, nullptr};
09561         }
09562
09563         if (ref_stack.empty())
09564         {
09565             root = std::move(value);
09566             return {true, & root};
09567         }
09568
09569         // skip this value if we already decided to skip the parent
09570         // (https://github.com/nlohmann/json/issues/971#issuecomment-413678360)
09571         if (!ref_stack.back())
09572         {
09573             return {false, nullptr};
09574         }
09575
09576         // we now only expect arrays and objects
09577         JSON_ASSERT(ref_stack.back()->is_array() || ref_stack.back()->is_object());
09578
09579         // array
09580         if (ref_stack.back()->is_array())
09581         {
09582             ref_stack.back()->m_data.m_value.array->emplace_back(std::move(value));
```

```
09583             return {true, & (ref_stack.back()->m_data.m_value.array->back())};
09584         }
09585 
09586         // object
09587         JSON_ASSERT(ref_stack.back()->is_object());
09588         // check if we should store an element for the current key
09589         JSON_ASSERT(!key_keep_stack.empty());
09590         const bool store_element = key_keep_stack.back();
09591         key_keep_stack.pop_back();
09592 
09593         if (!store_element)
09594         {
09595             return {false, nullptr};
09596         }
09597 
09598         JSON_ASSERT(object_element);
09599         *object_element = std::move(value);
09600         return {true, object_element};
09601     }
09602 
09604     BasicJsonType& root;
09606     std::vector<BasicJsonType*> ref_stack {};
09608     std::vector<bool> keep_stack {}; // NOLINT(readability-redundant-member-init)
09610     std::vector<bool> key_keep_stack {}; // NOLINT(readability-redundant-member-init)
09612     BasicJsonType* object_element = nullptr;
09614     bool errored = false;
09616     const parser_callback_t callback = nullptr;
09618     const bool allow_exceptions = true;
09620     BasicJsonType discarded = BasicJsonType::value_t::discarded;
09622     lexer_t* m_lexer_ref = nullptr;
09623 };
09624 
09625 template<typename BasicJsonType>
09626 class json_sax_acceptor
09627 {
09628   public:
09629     using number_integer_t = typename BasicJsonType::number_integer_t;
09630     using number_unsigned_t = typename BasicJsonType::number_unsigned_t;
09631     using number_float_t = typename BasicJsonType::number_float_t;
09632     using string_t = typename BasicJsonType::string_t;
09633     using binary_t = typename BasicJsonType::binary_t;
09634 
09635     bool null()
09636     {
09637         return true;
09638     }
09639 
09640     bool boolean(bool /*unused*/)
09641     {
09642         return true;
09643     }
09644 
09645     bool number_integer(number_integer_t /*unused*/)
09646     {
09647         return true;
09648     }
09649 
09650     bool number_unsigned(number_unsigned_t /*unused*/)
09651     {
09652         return true;
09653     }
09654 
09655     bool number_float(number_float_t /*unused*/, const string_t& /*unused*/)
09656     {
09657         return true;
09658     }
09659 
09660     bool string(string_t& /*unused*/)
09661     {
09662         return true;
09663     }
09664 
09665     bool binary(binary_t& /*unused*/)
09666     {
09667         return true;
09668     }
09669 
09670     bool start_object(std::size_t /*unused*/ = detail::unknown_size())
09671     {
09672         return true;
09673     }
09674 
09675     bool key(string_t& /*unused*/)
09676     {
09677         return true;
09678     }
09679 
```

```
09680     bool end_object()
09681     {
09682         return true;
09683     }
09684
09685     bool start_array(std::size_t /*unused*/ = detail::unknown_size())
09686     {
09687         return true;
09688     }
09689
09690     bool end_array()
09691     {
09692         return true;
09693     }
09694
09695     bool parse_error(std::size_t /*unused*/, const std::string& /*unused*/, const detail::exception&
      /*unused*/)
09696     {
09697         return false;
09698     }
09699 };
09700
09701 }  // namespace detail
09702 NLOHMANN_JSON_NAMESPACE_END
09703
09704 // #include <nlohmann/detail/input/lexer.hpp>
09705
09706 // #include <nlohmann/detail/macro_scope.hpp>
09707
09708 // #include <nlohmann/detail/meta/is_sax.hpp>
09709 //     __ _____ _____ _____
09710 //  __|  |   __|     |   | |  JSON for Modern C++
09711 // |  |  |__   |  |  | | | |  version 3.12.0
09712 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
09713 //
09714 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
09715 // SPDX-License-Identifier: MIT
09716
09717
09718
09719 #include <cstdint> // size_t
09720 #include <utility> // declval
09721 #include <string> // string
09722
09723 // #include <nlohmann/detail/abi_macros.hpp>
09724
09725 // #include <nlohmann/detail/meta/detected.hpp>
09726
09727 // #include <nlohmann/detail/meta/type_traits.hpp>
09728
09729
09730 NLOHMANN_JSON_NAMESPACE_BEGIN
09731 namespace detail
09732 {
09733
09734 template<typename T>
09735 using null_function_t = decltype(std::declval<T&>().null());
09736
09737 template<typename T>
09738 using boolean_function_t =
09739     decltype(std::declval<T&>().boolean(std::declval<bool>()));
09740
09741 template<typename T, typename Integer>
09742 using number_integer_function_t =
09743     decltype(std::declval<T&>().number_integer(std::declval<Integer>()));
09744
09745 template<typename T, typename Unsigned>
09746 using number_unsigned_function_t =
09747     decltype(std::declval<T&>().number_unsigned(std::declval<Unsigned>()));
09748
09749 template<typename T, typename Float, typename String>
09750 using number_float_function_t = decltype(std::declval<T&>().number_float(
09751                                      std::declval<Float>(), std::declval<const String&>()));
09752
09753 template<typename T, typename String>
09754 using string_function_t =
09755     decltype(std::declval<T&>().string(std::declval<String&>()));
09756
09757 template<typename T, typename Binary>
09758 using binary_function_t =
09759     decltype(std::declval<T&>().binary(std::declval<Binary&>()));
09760
09761 template<typename T>
09762 using start_object_function_t =
09763     decltype(std::declval<T&>().start_object(std::declval<std::size_t>()));
09764
09765 template<typename T, typename String>
```

```
09766 using key_function_t =
09767     decltype(std::declval<T&>().key(std::declval<String&>()));
09768
09769 template<typename T>
09770 using end_object_function_t = decltype(std::declval<T&>().end_object());
09771
09772 template<typename T>
09773 using start_array_function_t =
09774     decltype(std::declval<T&>().start_array(std::declval<std::size_t>()));
09775
09776 template<typename T>
09777 using end_array_function_t = decltype(std::declval<T&>().end_array());
09778
09779 template<typename T, typename Exception>
09780 using parse_error_function_t = decltype(std::declval<T&>().parse_error(
09781         std::declval<std::size_t>(), std::declval<const std::string&>(),
09782         std::declval<const Exception&>()));
09783
09784 template<typename SAX, typename BasicJsonType>
09785 struct is_sax
09786 {
09787   private:
09788     static_assert(is_basic_json<BasicJsonType>::value,
09789                   "BasicJsonType must be of type basic_json<...>");
09790
09791     using number_integer_t = typename BasicJsonType::number_integer_t;
09792     using number_unsigned_t = typename BasicJsonType::number_unsigned_t;
09793     using number_float_t = typename BasicJsonType::number_float_t;
09794     using string_t = typename BasicJsonType::string_t;
09795     using binary_t = typename BasicJsonType::binary_t;
09796     using exception_t = typename BasicJsonType::exception;
09797
09798   public:
09799     static constexpr bool value =
09800         is_detected_exact<bool, null_function_t, SAX>::value &&
09801         is_detected_exact<bool, boolean_function_t, SAX>::value &&
09802         is_detected_exact<bool, number_integer_function_t, SAX, number_integer_t>::value &&
09803         is_detected_exact<bool, number_unsigned_function_t, SAX, number_unsigned_t>::value &&
09804         is_detected_exact<bool, number_float_function_t, SAX, number_float_t, string_t>::value &&
09805         is_detected_exact<bool, string_function_t, SAX, string_t>::value &&
09806         is_detected_exact<bool, binary_function_t, SAX, binary_t>::value &&
09807         is_detected_exact<bool, start_object_function_t, SAX>::value &&
09808         is_detected_exact<bool, key_function_t, SAX, string_t>::value &&
09809         is_detected_exact<bool, end_object_function_t, SAX>::value &&
09810         is_detected_exact<bool, start_array_function_t, SAX>::value &&
09811         is_detected_exact<bool, end_array_function_t, SAX>::value &&
09812         is_detected_exact<bool, parse_error_function_t, SAX, exception_t>::value;
09813 };
09814
09815 template<typename SAX, typename BasicJsonType>
09816 struct is_sax_static_asserts
09817 {
09818   private:
09819     static_assert(is_basic_json<BasicJsonType>::value,
09820                   "BasicJsonType must be of type basic_json<...>");
09821
09822     using number_integer_t = typename BasicJsonType::number_integer_t;
09823     using number_unsigned_t = typename BasicJsonType::number_unsigned_t;
09824     using number_float_t = typename BasicJsonType::number_float_t;
09825     using string_t = typename BasicJsonType::string_t;
09826     using binary_t = typename BasicJsonType::binary_t;
09827     using exception_t = typename BasicJsonType::exception;
09828
09829   public:
09830     static_assert(is_detected_exact<bool, null_function_t, SAX>::value,
09831                   "Missing/invalid function: bool null()");
09832     static_assert(is_detected_exact<bool, boolean_function_t, SAX>::value,
09833                   "Missing/invalid function: bool boolean(bool)");
09834     static_assert(is_detected_exact<bool, boolean_function_t, SAX>::value,
09835                   "Missing/invalid function: bool boolean(bool)");
09836     static_assert(
09837         is_detected_exact<bool, number_integer_function_t, SAX,
09838         number_integer_t>::value,
09839         "Missing/invalid function: bool number_integer(number_integer_t)");
09840     static_assert(
09841         is_detected_exact<bool, number_unsigned_function_t, SAX,
09842         number_unsigned_t>::value,
09843         "Missing/invalid function: bool number_unsigned(number_unsigned_t)");
09844     static_assert(is_detected_exact<bool, number_float_function_t, SAX,
09845                   number_float_t, string_t>::value,
09846                   "Missing/invalid function: bool number_float(number_float_t, const string_t&)");
09847     static_assert(
09848         is_detected_exact<bool, string_function_t, SAX, string_t>::value,
09849         "Missing/invalid function: bool string(string_t&)");
09850     static_assert(
09851         is_detected_exact<bool, binary_function_t, SAX, binary_t>::value,
09852         "Missing/invalid function: bool binary(binary_t&)");
```

```
09853     static_assert(is_detected_exact<bool, start_object_function_t, SAX>::value,
09854                   "Missing/invalid function: bool start_object(std::size_t)");
09855     static_assert(is_detected_exact<bool, key_function_t, SAX, string_t>::value,
09856                   "Missing/invalid function: bool key(string_t&)");
09857     static_assert(is_detected_exact<bool, end_object_function_t, SAX>::value,
09858                   "Missing/invalid function: bool end_object()");
09859     static_assert(is_detected_exact<bool, start_array_function_t, SAX>::value,
09860                   "Missing/invalid function: bool start_array(std::size_t)");
09861     static_assert(is_detected_exact<bool, end_array_function_t, SAX>::value,
09862                   "Missing/invalid function: bool end_array()");
09863     static_assert(
09864         is_detected_exact<bool, parse_error_function_t, SAX, exception_t>::value,
09865         "Missing/invalid function: bool parse_error(std::size_t, const "
09866         "std::string&, const exception&)");
09867 };
09868
09869 }  // namespace detail
09870 NLOHMANN_JSON_NAMESPACE_END
09871
09872 // #include <nlohmann/detail/meta/type_traits.hpp>
09873
09874 // #include <nlohmann/detail/string_concat.hpp>
09875
09876 // #include <nlohmann/detail/value_t.hpp>
09877
09878
09879 NLOHMANN_JSON_NAMESPACE_BEGIN
09880 namespace detail
09881 {
09882
09884 enum class cbor_tag_handler_t
09885 {
09886     error,
09887     ignore,
09888     store
09889 };
09890
09898 inline bool little_endianness(int num = 1) noexcept
09899 {
09900     return *reinterpret_cast<char*>(&num) == 1;
09901 }
09902
09904 // binary reader //
09906
09910 template<typename BasicJsonType, typename InputAdapterType, typename SAX =
      json_sax_dom_parser<BasicJsonType, InputAdapterType>>
09911 class binary_reader
09912 {
09913     using number_integer_t = typename BasicJsonType::number_integer_t;
09914     using number_unsigned_t = typename BasicJsonType::number_unsigned_t;
09915     using number_float_t = typename BasicJsonType::number_float_t;
09916     using string_t = typename BasicJsonType::string_t;
09917     using binary_t = typename BasicJsonType::binary_t;
09918     using json_sax_t = SAX;
09919     using char_type = typename InputAdapterType::char_type;
09920     using char_int_type = typename char_traits<char_type>::int_type;
09921
09922   public:
09928     explicit binary_reader(InputAdapterType&& adapter, const input_format_t format =
      input_format_t::json) noexcept : ia(std::move(adapter)), input_format(format)
09929     {
09930         (void)detail::is_sax_static_asserts<SAX, BasicJsonType> {};
09931     }
09932
09933     // make class move-only
09934     binary_reader(const binary_reader&) = delete;
09935     binary_reader(binary_reader&&) = default; //
      NOLINT(hicpp-noexcept-move,performance-noexcept-move-constructor)
09936     binary_reader& operator=(const binary_reader&) = delete;
09937     binary_reader& operator=(binary_reader&&) = default; //
      NOLINT(hicpp-noexcept-move,performance-noexcept-move-constructor)
09938     ~binary_reader() = default;
09939
09948     JSON_HEDLEY_NON_NULL(3)
09949     bool sax_parse(const input_format_t format,
09950                    json_sax_t* sax_,
09951                    const bool strict = true,
09952                    const cbor_tag_handler_t tag_handler = cbor_tag_handler_t::error)
09953     {
09954         sax = sax_;
09955         bool result = false;
09956
09957         switch (format)
09958         {
09959             case input_format_t::bson:
09960                 result = parse_bson_internal();
09961                 break;
```

```
09962
09963             case input_format_t::cbor:
09964                 result = parse_cbor_internal(true, tag_handler);
09965                 break;
09966
09967             case input_format_t::msgpack:
09968                 result = parse_msgpack_internal();
09969                 break;
09970
09971             case input_format_t::ubjson:
09972             case input_format_t::bjdata:
09973                 result = parse_ubjson_internal();
09974                 break;
09975
09976             case input_format_t::json: // LCOV_EXCL_LINE
09977             default:            // LCOV_EXCL_LINE
09978                 JSON_ASSERT(false); // NOLINT(cert-dcl03-c,hicpp-static-assert,misc-static-assert)
      LCOV_EXCL_LINE
09979         }
09980
09981         // strict mode: next byte must be EOF
09982         if (result && strict)
09983         {
09984             if (input_format == input_format_t::ubjson || input_format == input_format_t::bjdata)
09985             {
09986                 get_ignore_noop();
09987             }
09988             else
09989             {
09990                 get();
09991             }
09992
09993             if (JSON_HEDLEY_UNLIKELY(current != char_traits<char_type>::eof()))
09994             {
09995                 return sax->parse_error(chars_read, get_token_string(), parse_error::create(110,
      chars_read,
09996                                         exception_message(input_format, concat("expected end of input;
      last byte: 0x", get_token_string()), "value"), nullptr));
09997             }
09998         }
09999
10000         return result;
10001     }
10002
10003   private:
10005     // BSON //
10007
10012     bool parse_bson_internal()
10013     {
10014         std::int32_t document_size{};
10015         get_number<std::int32_t, true>(input_format_t::bson, document_size);
10016
10017         if (JSON_HEDLEY_UNLIKELY(!sax->start_object(detail::unknown_size())))
10018         {
10019             return false;
10020         }
10021
10022         if (JSON_HEDLEY_UNLIKELY(!parse_bson_element_list(/*is_array*/false)))
10023         {
10024             return false;
10025         }
10026
10027         return sax->end_object();
10028     }
10029
10037     bool get_bson_cstr(string_t& result)
10038     {
10039         auto out = std::back_inserter(result);
10040         while (true)
10041         {
10042             get();
10043             if (JSON_HEDLEY_UNLIKELY(!unexpect_eof(input_format_t::bson, "cstring")))
10044             {
10045                 return false;
10046             }
10047             if (current == 0x00)
10048             {
10049                 return true;
10050             }
10051             *out++ = static_cast<typename string_t::value_type>(current);
10052         }
10053     }
10054
10066     template<typename NumberType>
10067     bool get_bson_string(const NumberType len, string_t& result)
10068     {
10069         if (JSON_HEDLEY_UNLIKELY(len < 1))
```

```
10070         {
10071             auto last_token = get_token_string();
10072             return sax->parse_error(chars_read, last_token, parse_error::create(112, chars_read,
10073                                     exception_message(input_format_t::bson, concat("string length must
      be at least 1, is ", std::to_string(len)), "string"), nullptr));
10074         }
10075 
10076         return get_string(input_format_t::bson, len - static_cast<NumberType>(1), result) && get() !=
      char_traits<char_type>::eof();
10077     }
10078 
10088     template<typename NumberType>
10089     bool get_bson_binary(const NumberType len, binary_t& result)
10090     {
10091         if (JSON_HEDLEY_UNLIKELY(len < 0))
10092         {
10093             auto last_token = get_token_string();
10094             return sax->parse_error(chars_read, last_token, parse_error::create(112, chars_read,
10095                                     exception_message(input_format_t::bson, concat("byte array length
      cannot be negative, is ", std::to_string(len)), "binary"), nullptr));
10096         }
10097 
10098         // All BSON binary values have a subtype
10099         std::uint8_t subtype{};
10100         get_number<std::uint8_t>(input_format_t::bson, subtype);
10101         result.set_subtype(subtype);
10102 
10103         return get_binary(input_format_t::bson, len, result);
10104     }
10105 
10116     bool parse_bson_element_internal(const char_int_type element_type,
10117                                      const std::size_t element_type_parse_position)
10118     {
10119         switch (element_type)
10120         {
10121             case 0x01: // double
10122             {
10123                 double number{};
10124                 return get_number<double, true>(input_format_t::bson, number) &&
      sax->number_float(static_cast<number_float_t>(number), "");
10125             }
10126 
10127             case 0x02: // string
10128             {
10129                 std::int32_t len{};
10130                 string_t value;
10131                 return get_number<std::int32_t, true>(input_format_t::bson, len) &&
      get_bson_string(len, value) && sax->string(value);
10132             }
10133 
10134             case 0x03: // object
10135             {
10136                 return parse_bson_internal();
10137             }
10138 
10139             case 0x04: // array
10140             {
10141                 return parse_bson_array();
10142             }
10143 
10144             case 0x05: // binary
10145             {
10146                 std::int32_t len{};
10147                 binary_t value;
10148                 return get_number<std::int32_t, true>(input_format_t::bson, len) &&
      get_bson_binary(len, value) && sax->binary(value);
10149             }
10150 
10151             case 0x08: // boolean
10152             {
10153                 return sax->boolean(get() != 0);
10154             }
10155 
10156             case 0x0A: // null
10157             {
10158                 return sax->null();
10159             }
10160 
10161             case 0x10: // int32
10162             {
10163                 std::int32_t value{};
10164                 return get_number<std::int32_t, true>(input_format_t::bson, value) &&
      sax->number_integer(value);
10165             }
10166 
10167             case 0x12: // int64
10168             {
```

```
10169                 std::int64_t value{};
10170                 return get_number<std::int64_t, true>(input_format_t::bson, value) &&
      sax->number_integer(value);
10171             }
10172
10173             case 0x11: // uint64
10174             {
10175                 std::uint64_t value{};
10176                 return get_number<std::uint64_t, true>(input_format_t::bson, value) &&
      sax->number_unsigned(value);
10177             }
10178
10179             default: // anything else is not supported (yet)
10180             {
10181                 std::array<char, 3> cr{{}};
10182                 static_cast<void>((std::snprintf)(cr.data(), cr.size(), "%.2hhX", static_cast<unsigned
      char>(element_type))); // NOLINT(cppcoreguidelines-pro-type-vararg,hicpp-vararg)
10183                 const std::string cr_str{cr.data()};
10184                 return sax->parse_error(element_type_parse_position, cr_str,
10185                                         parse_error::create(114, element_type_parse_position,
      concat("Unsupported BSON record type 0x", cr_str), nullptr));
10186             }
10187         }
10188     }
10189
10202     bool parse_bson_element_list(const bool is_array)
10203     {
10204         string_t key;
10205
10206         while (auto element_type = get())
10207         {
10208             if (JSON_HEDLEY_UNLIKELY(!unexpect_eof(input_format_t::bson, "element list")))
10209             {
10210                 return false;
10211             }
10212
10213             const std::size_t element_type_parse_position = chars_read;
10214             if (JSON_HEDLEY_UNLIKELY(!get_bson_cstr(key)))
10215             {
10216                 return false;
10217             }
10218
10219             if (!is_array && !sax->key(key))
10220             {
10221                 return false;
10222             }
10223
10224             if (JSON_HEDLEY_UNLIKELY(!parse_bson_element_internal(element_type,
      element_type_parse_position)))
10225             {
10226                 return false;
10227             }
10228
10229             // get_bson_cstr only appends
10230             key.clear();
10231         }
10232
10233         return true;
10234     }
10235
10240     bool parse_bson_array()
10241     {
10242         std::int32_t document_size{};
10243         get_number<std::int32_t, true>(input_format_t::bson, document_size);
10244
10245         if (JSON_HEDLEY_UNLIKELY(!sax->start_array(detail::unknown_size())))
10246         {
10247             return false;
10248         }
10249
10250         if (JSON_HEDLEY_UNLIKELY(!parse_bson_element_list(/*is_array*/true)))
10251         {
10252             return false;
10253         }
10254
10255         return sax->end_array();
10256     }
10257
10259     // CBOR //
10261
10270
10271     template<typename NumberType>
10272     bool get_cbor_negative_integer()
10273     {
10274         NumberType number{};
10275         if (JSON_HEDLEY_UNLIKELY(!get_number(input_format_t::cbor, number)))
10276         {
```

```
10277             return false;
10278         }
10279         const auto max_val = static_cast<NumberType>((std::numeric_limits<number_integer_t>::max)());
10280         if (number > max_val)
10281         {
10282             return sax->parse_error(chars_read, get_token_string(),
10283                                     parse_error::create(112, chars_read,
10284                                             exception_message(input_format_t::cbor, "negative integer
      overflow", "value"), nullptr));
10285         }
10286         return sax->number_integer(static_cast<number_integer_t>(-1) -
      static_cast<number_integer_t>(number));
10287     }
10288
10289     bool parse_cbor_internal(const bool get_char,
10290                              const cbor_tag_handler_t tag_handler)
10291     {
10292         switch (get_char ? get() : current)
10293         {
10294             // EOF
10295             case char_traits<char_type>::eof():
10296                 return unexpect_eof(input_format_t::cbor, "value");
10297
10298             // Integer 0x00..0x17 (0..23)
10299             case 0x00:
10300             case 0x01:
10301             case 0x02:
10302             case 0x03:
10303             case 0x04:
10304             case 0x05:
10305             case 0x06:
10306             case 0x07:
10307             case 0x08:
10308             case 0x09:
10309             case 0x0A:
10310             case 0x0B:
10311             case 0x0C:
10312             case 0x0D:
10313             case 0x0E:
10314             case 0x0F:
10315             case 0x10:
10316             case 0x11:
10317             case 0x12:
10318             case 0x13:
10319             case 0x14:
10320             case 0x15:
10321             case 0x16:
10322             case 0x17:
10323                 return sax->number_unsigned(static_cast<number_unsigned_t>(current));
10324
10325             case 0x18: // Unsigned integer (one-byte uint8_t follows)
10326             {
10327                 std::uint8_t number{};
10328                 return get_number(input_format_t::cbor, number) && sax->number_unsigned(number);
10329             }
10330
10331             case 0x19: // Unsigned integer (two-byte uint16_t follows)
10332             {
10333                 std::uint16_t number{};
10334                 return get_number(input_format_t::cbor, number) && sax->number_unsigned(number);
10335             }
10336
10337             case 0x1A: // Unsigned integer (four-byte uint32_t follows)
10338             {
10339                 std::uint32_t number{};
10340                 return get_number(input_format_t::cbor, number) && sax->number_unsigned(number);
10341             }
10342
10343             case 0x1B: // Unsigned integer (eight-byte uint64_t follows)
10344             {
10345                 std::uint64_t number{};
10346                 return get_number(input_format_t::cbor, number) && sax->number_unsigned(number);
10347             }
10348
10349             // Negative integer -1-0x00..-1-0x17 (-1..-24)
10350             case 0x20:
10351             case 0x21:
10352             case 0x22:
10353             case 0x23:
10354             case 0x24:
10355             case 0x25:
10356             case 0x26:
10357             case 0x27:
10358             case 0x28:
10359             case 0x29:
10360             case 0x2A:
10361             case 0x2B:
```

```
10362             case 0x2C:
10363             case 0x2D:
10364             case 0x2E:
10365             case 0x2F:
10366             case 0x30:
10367             case 0x31:
10368             case 0x32:
10369             case 0x33:
10370             case 0x34:
10371             case 0x35:
10372             case 0x36:
10373             case 0x37:
10374                 return sax->number_integer(static_cast<std::int8_t>(0x20 - 1 - current));
10375
10376             case 0x38: // Negative integer (one-byte uint8_t follows)
10377                 return get_cbor_negative_integer<std::uint8_t>();
10378
10379             case 0x39: // Negative integer -1-n (two-byte uint16_t follows)
10380                 return get_cbor_negative_integer<std::uint16_t>();
10381
10382             case 0x3A: // Negative integer -1-n (four-byte uint32_t follows)
10383                 return get_cbor_negative_integer<std::uint32_t>();
10384
10385             case 0x3B: // Negative integer -1-n (eight-byte uint64_t follows)
10386                 return get_cbor_negative_integer<std::uint64_t>();
10387
10388             // Binary data (0x00..0x17 bytes follow)
10389             case 0x40:
10390             case 0x41:
10391             case 0x42:
10392             case 0x43:
10393             case 0x44:
10394             case 0x45:
10395             case 0x46:
10396             case 0x47:
10397             case 0x48:
10398             case 0x49:
10399             case 0x4A:
10400             case 0x4B:
10401             case 0x4C:
10402             case 0x4D:
10403             case 0x4E:
10404             case 0x4F:
10405             case 0x50:
10406             case 0x51:
10407             case 0x52:
10408             case 0x53:
10409             case 0x54:
10410             case 0x55:
10411             case 0x56:
10412             case 0x57:
10413             case 0x58: // Binary data (one-byte uint8_t for n follows)
10414             case 0x59: // Binary data (two-byte uint16_t for n follow)
10415             case 0x5A: // Binary data (four-byte uint32_t for n follow)
10416             case 0x5B: // Binary data (eight-byte uint64_t for n follow)
10417             case 0x5F: // Binary data (indefinite length)
10418             {
10419                 binary_t b;
10420                 return get_cbor_binary(b) && sax->binary(b);
10421             }
10422
10423             // UTF-8 string (0x00..0x17 bytes follow)
10424             case 0x60:
10425             case 0x61:
10426             case 0x62:
10427             case 0x63:
10428             case 0x64:
10429             case 0x65:
10430             case 0x66:
10431             case 0x67:
10432             case 0x68:
10433             case 0x69:
10434             case 0x6A:
10435             case 0x6B:
10436             case 0x6C:
10437             case 0x6D:
10438             case 0x6E:
10439             case 0x6F:
10440             case 0x70:
10441             case 0x71:
10442             case 0x72:
10443             case 0x73:
10444             case 0x74:
10445             case 0x75:
10446             case 0x76:
10447             case 0x77:
10448             case 0x78: // UTF-8 string (one-byte uint8_t for n follows)
```

```
10449             case 0x79: // UTF-8 string (two-byte uint16_t for n follow)
10450             case 0x7A: // UTF-8 string (four-byte uint32_t for n follow)
10451             case 0x7B: // UTF-8 string (eight-byte uint64_t for n follow)
10452             case 0x7F: // UTF-8 string (indefinite length)
10453             {
10454                 string_t s;
10455                 return get_cbor_string(s) && sax->string(s);
10456             }
10457
10458             // array (0x00..0x17 data items follow)
10459             case 0x80:
10460             case 0x81:
10461             case 0x82:
10462             case 0x83:
10463             case 0x84:
10464             case 0x85:
10465             case 0x86:
10466             case 0x87:
10467             case 0x88:
10468             case 0x89:
10469             case 0x8A:
10470             case 0x8B:
10471             case 0x8C:
10472             case 0x8D:
10473             case 0x8E:
10474             case 0x8F:
10475             case 0x90:
10476             case 0x91:
10477             case 0x92:
10478             case 0x93:
10479             case 0x94:
10480             case 0x95:
10481             case 0x96:
10482             case 0x97:
10483                 return get_cbor_array(
10484                         conditional_static_cast<std::size_t>(static_cast<unsigned int>(current) &
      0x1Fu), tag_handler);
10485
10486             case 0x98: // array (one-byte uint8_t for n follows)
10487             {
10488                 std::uint8_t len{};
10489                 return get_number(input_format_t::cbor, len) &&
      get_cbor_array(static_cast<std::size_t>(len), tag_handler);
10490             }
10491
10492             case 0x99: // array (two-byte uint16_t for n follow)
10493             {
10494                 std::uint16_t len{};
10495                 return get_number(input_format_t::cbor, len) &&
      get_cbor_array(static_cast<std::size_t>(len), tag_handler);
10496             }
10497
10498             case 0x9A: // array (four-byte uint32_t for n follow)
10499             {
10500                 std::uint32_t len{};
10501                 return get_number(input_format_t::cbor, len) &&
      get_cbor_array(conditional_static_cast<std::size_t>(len), tag_handler);
10502             }
10503
10504             case 0x9B: // array (eight-byte uint64_t for n follow)
10505             {
10506                 std::uint64_t len{};
10507                 return get_number(input_format_t::cbor, len) &&
      get_cbor_array(conditional_static_cast<std::size_t>(len), tag_handler);
10508             }
10509
10510             case 0x9F: // array (indefinite length)
10511                 return get_cbor_array(detail::unknown_size(), tag_handler);
10512
10513             // map (0x00..0x17 pairs of data items follow)
10514             case 0xA0:
10515             case 0xA1:
10516             case 0xA2:
10517             case 0xA3:
10518             case 0xA4:
10519             case 0xA5:
10520             case 0xA6:
10521             case 0xA7:
10522             case 0xA8:
10523             case 0xA9:
10524             case 0xAA:
10525             case 0xAB:
10526             case 0xAC:
10527             case 0xAD:
10528             case 0xAE:
10529             case 0xAF:
10530             case 0xB0:
```

```
10531             case 0xB1:
10532             case 0xB2:
10533             case 0xB3:
10534             case 0xB4:
10535             case 0xB5:
10536             case 0xB6:
10537             case 0xB7:
10538                 return get_cbor_object(conditional_static_cast<std::size_t>(static_cast<unsigned
      int>(current) & 0x1Fu), tag_handler);
10539
10540             case 0xB8: // map (one-byte uint8_t for n follows)
10541             {
10542                 std::uint8_t len{};
10543                 return get_number(input_format_t::cbor, len) &&
      get_cbor_object(static_cast<std::size_t>(len), tag_handler);
10544             }
10545
10546             case 0xB9: // map (two-byte uint16_t for n follow)
10547             {
10548                 std::uint16_t len{};
10549                 return get_number(input_format_t::cbor, len) &&
      get_cbor_object(static_cast<std::size_t>(len), tag_handler);
10550             }
10551
10552             case 0xBA: // map (four-byte uint32_t for n follow)
10553             {
10554                 std::uint32_t len{};
10555                 return get_number(input_format_t::cbor, len) &&
      get_cbor_object(conditional_static_cast<std::size_t>(len), tag_handler);
10556             }
10557
10558             case 0xBB: // map (eight-byte uint64_t for n follow)
10559             {
10560                 std::uint64_t len{};
10561                 return get_number(input_format_t::cbor, len) &&
      get_cbor_object(conditional_static_cast<std::size_t>(len), tag_handler);
10562             }
10563
10564             case 0xBF: // map (indefinite length)
10565                 return get_cbor_object(detail::unknown_size(), tag_handler);
10566
10567             case 0xC6: // tagged item
10568             case 0xC7:
10569             case 0xC8:
10570             case 0xC9:
10571             case 0xCA:
10572             case 0xCB:
10573             case 0xCC:
10574             case 0xCD:
10575             case 0xCE:
10576             case 0xCF:
10577             case 0xD0:
10578             case 0xD1:
10579             case 0xD2:
10580             case 0xD3:
10581             case 0xD4:
10582             case 0xD8: // tagged item (1 byte follows)
10583             case 0xD9: // tagged item (2 bytes follow)
10584             case 0xDA: // tagged item (4 bytes follow)
10585             case 0xDB: // tagged item (8 bytes follow)
10586             {
10587                 switch (tag_handler)
10588                 {
10589                     case cbor_tag_handler_t::error:
10590                     {
10591                         auto last_token = get_token_string();
10592                         return sax->parse_error(chars_read, last_token, parse_error::create(112,
      chars_read,
10593                                                 exception_message(input_format_t::cbor,
      concat("invalid byte: 0x", last_token), "value"), nullptr));
10594                     }
10595
10596                     case cbor_tag_handler_t::ignore:
10597                     {
10598                         // ignore binary subtype
10599                         switch (current)
10600                         {
10601                             case 0xD8:
10602                             {
10603                                 std::uint8_t subtype_to_ignore{};
10604                                 get_number(input_format_t::cbor, subtype_to_ignore);
10605                                 break;
10606                             }
10607                             case 0xD9:
10608                             {
10609                                 std::uint16_t subtype_to_ignore{};
10610                                 get_number(input_format_t::cbor, subtype_to_ignore);
```

```
10611                                 break;
10612                             }
10613                             case 0xDA:
10614                             {
10615                                 std::uint32_t subtype_to_ignore{};
10616                                 get_number(input_format_t::cbor, subtype_to_ignore);
10617                                 break;
10618                             }
10619                             case 0xDB:
10620                             {
10621                                 std::uint64_t subtype_to_ignore{};
10622                                 get_number(input_format_t::cbor, subtype_to_ignore);
10623                                 break;
10624                             }
10625                             default:
10626                                 break;
10627                         }
10628                         return parse_cbor_internal(true, tag_handler);
10629                     }
10630 
10631                     case cbor_tag_handler_t::store:
10632                     {
10633                         binary_t b;
10634                         // use binary subtype and store in a binary container
10635                         switch (current)
10636                         {
10637                             case 0xD8:
10638                             {
10639                                 std::uint8_t subtype{};
10640                                 get_number(input_format_t::cbor, subtype);
10641                                 b.set_subtype(detail::conditional_static_cast<typename
     binary_t::subtype_type>(subtype));
10642                                 break;
10643                             }
10644                             case 0xD9:
10645                             {
10646                                 std::uint16_t subtype{};
10647                                 get_number(input_format_t::cbor, subtype);
10648                                 b.set_subtype(detail::conditional_static_cast<typename
     binary_t::subtype_type>(subtype));
10649                                 break;
10650                             }
10651                             case 0xDA:
10652                             {
10653                                 std::uint32_t subtype{};
10654                                 get_number(input_format_t::cbor, subtype);
10655                                 b.set_subtype(detail::conditional_static_cast<typename
     binary_t::subtype_type>(subtype));
10656                                 break;
10657                             }
10658                             case 0xDB:
10659                             {
10660                                 std::uint64_t subtype{};
10661                                 get_number(input_format_t::cbor, subtype);
10662                                 b.set_subtype(detail::conditional_static_cast<typename
     binary_t::subtype_type>(subtype));
10663                                 break;
10664                             }
10665                             default:
10666                                 return parse_cbor_internal(true, tag_handler);
10667                         }
10668                         get();
10669                         return get_cbor_binary(b) && sax->binary(b);
10670                     }
10671 
10672                     default:                // LCOV_EXCL_LINE
10673                         JSON_ASSERT(false); //
     NOLINT(cert-dcl03-c,hicpp-static-assert,misc-static-assert) LCOV_EXCL_LINE
10674                         return false;       // LCOV_EXCL_LINE
10675                 }
10676             }
10677 
10678             case 0xF4: // false
10679                 return sax->boolean(false);
10680 
10681             case 0xF5: // true
10682                 return sax->boolean(true);
10683 
10684             case 0xF6: // null
10685                 return sax->null();
10686 
10687             case 0xF9: // Half-Precision Float (two-byte IEEE 754)
10688             {
10689                 const auto byte1_raw = get();
10690                 if (JSON_HEDLEY_UNLIKELY(!unexpect_eof(input_format_t::cbor, "number")))
10691                 {
10692                     return false;
```

```
10693                 }
10694                 const auto byte2_raw = get();
10695                 if (JSON_HEDLEY_UNLIKELY(!unexpect_eof(input_format_t::cbor, "number")))
10696                 {
10697                     return false;
10698                 }
10699
10700                 const auto byte1 = static_cast<unsigned char>(byte1_raw);
10701                 const auto byte2 = static_cast<unsigned char>(byte2_raw);
10702
10703                 // Code from RFC 7049, Appendix D, Figure 3:
10704                 // As half-precision floating-point numbers were only added
10705                 // to IEEE 754 in 2008, today's programming platforms often
10706                 // still only have limited support for them. It is very
10707                 // easy to include at least decoding support for them even
10708                 // without such support. An example of a small decoder for
10709                 // half-precision floating-point numbers in the C language
10710                 // is shown in Fig. 3.
10711                 const auto half = static_cast<unsigned int>((byte1 « 8u) + byte2);
10712                 const double val = [&half]
10713                 {
10714                     const int exp = (half » 10u) & 0x1Fu;
10715                     const unsigned int mant = half & 0x3FFu;
10716                     JSON_ASSERT(0 <= exp&& exp <= 32);
10717                     JSON_ASSERT(mant <= 1024);
10718                     switch (exp)
10719                     {
10720                         case 0:
10721                             return std::ldexp(mant, -24);
10722                         case 31:
10723                             return (mant == 0)
10724                             ? std::numeric_limits<double>::infinity()
10725                             : std::numeric_limits<double>::quiet_NaN();
10726                         default:
10727                             return std::ldexp(mant + 1024, exp - 25);
10728                     }
10729                 }();
10730                 return sax->number_float((half & 0x8000u) != 0
10731                                          ? static_cast<number_float_t>(-val)
10732                                          : static_cast<number_float_t>(val), "");
10733             }
10734
10735             case 0xFA: // Single-Precision Float (four-byte IEEE 754)
10736             {
10737                 float number{};
10738                 return get_number(input_format_t::cbor, number) &&
     sax->number_float(static_cast<number_float_t>(number), "");
10739             }
10740
10741             case 0xFB: // Double-Precision Float (eight-byte IEEE 754)
10742             {
10743                 double number{};
10744                 return get_number(input_format_t::cbor, number) &&
     sax->number_float(static_cast<number_float_t>(number), "");
10745             }
10746
10747             default: // anything else (0xFF is handled inside the other types)
10748             {
10749                 auto last_token = get_token_string();
10750                 return sax->parse_error(chars_read, last_token, parse_error::create(112, chars_read,
10751                                         exception_message(input_format_t::cbor, concat("invalid byte:
     0x", last_token), "value"), nullptr));
10752             }
10753         }
10754     }
10755
10767     bool get_cbor_string(string_t& result)
10768     {
10769         if (JSON_HEDLEY_UNLIKELY(!unexpect_eof(input_format_t::cbor, "string")))
10770         {
10771             return false;
10772         }
10773
10774         switch (current)
10775         {
10776             // UTF-8 string (0x00..0x17 bytes follow)
10777             case 0x60:
10778             case 0x61:
10779             case 0x62:
10780             case 0x63:
10781             case 0x64:
10782             case 0x65:
10783             case 0x66:
10784             case 0x67:
10785             case 0x68:
10786             case 0x69:
10787             case 0x6A:
```

```
10788             case 0x6B:
10789             case 0x6C:
10790             case 0x6D:
10791             case 0x6E:
10792             case 0x6F:
10793             case 0x70:
10794             case 0x71:
10795             case 0x72:
10796             case 0x73:
10797             case 0x74:
10798             case 0x75:
10799             case 0x76:
10800             case 0x77:
10801             {
10802                 return get_string(input_format_t::cbor, static_cast<unsigned int>(current) & 0x1Fu,
      result);
10803             }
10804
10805             case 0x78: // UTF-8 string (one-byte uint8_t for n follows)
10806             {
10807                 std::uint8_t len{};
10808                 return get_number(input_format_t::cbor, len) && get_string(input_format_t::cbor, len,
      result);
10809             }
10810
10811             case 0x79: // UTF-8 string (two-byte uint16_t for n follow)
10812             {
10813                 std::uint16_t len{};
10814                 return get_number(input_format_t::cbor, len) && get_string(input_format_t::cbor, len,
      result);
10815             }
10816
10817             case 0x7A: // UTF-8 string (four-byte uint32_t for n follow)
10818             {
10819                 std::uint32_t len{};
10820                 return get_number(input_format_t::cbor, len) && get_string(input_format_t::cbor, len,
      result);
10821             }
10822
10823             case 0x7B: // UTF-8 string (eight-byte uint64_t for n follow)
10824             {
10825                 std::uint64_t len{};
10826                 return get_number(input_format_t::cbor, len) && get_string(input_format_t::cbor, len,
      result);
10827             }
10828
10829             case 0x7F: // UTF-8 string (indefinite length)
10830             {
10831                 while (get() != 0xFF)
10832                 {
10833                     string_t chunk;
10834                     if (!get_cbor_string(chunk))
10835                     {
10836                         return false;
10837                     }
10838                     result.append(chunk);
10839                 }
10840                 return true;
10841             }
10842
10843             default:
10844             {
10845                 auto last_token = get_token_string();
10846                 return sax->parse_error(chars_read, last_token, parse_error::create(113, chars_read,
10847                                         exception_message(input_format_t::cbor, concat("expected
      length specification (0x60-0x7B) or indefinite string type (0x7F); last byte: 0x", last_token),
      "string"), nullptr));
10848             }
10849         }
10850     }
10851
10863     bool get_cbor_binary(binary_t& result)
10864     {
10865         if (JSON_HEDLEY_UNLIKELY(!unexpect_eof(input_format_t::cbor, "binary")))
10866         {
10867             return false;
10868         }
10869
10870         switch (current)
10871         {
10872             // Binary data (0x00..0x17 bytes follow)
10873             case 0x40:
10874             case 0x41:
10875             case 0x42:
10876             case 0x43:
10877             case 0x44:
10878             case 0x45:
```

```
10879             case 0x46:
10880             case 0x47:
10881             case 0x48:
10882             case 0x49:
10883             case 0x4A:
10884             case 0x4B:
10885             case 0x4C:
10886             case 0x4D:
10887             case 0x4E:
10888             case 0x4F:
10889             case 0x50:
10890             case 0x51:
10891             case 0x52:
10892             case 0x53:
10893             case 0x54:
10894             case 0x55:
10895             case 0x56:
10896             case 0x57:
10897             {
10898                 return get_binary(input_format_t::cbor, static_cast<unsigned int>(current) & 0x1Fu,
      result);
10899             }
10900
10901             case 0x58: // Binary data (one-byte uint8_t for n follows)
10902             {
10903                 std::uint8_t len{};
10904                 return get_number(input_format_t::cbor, len) &&
10905                        get_binary(input_format_t::cbor, len, result);
10906             }
10907
10908             case 0x59: // Binary data (two-byte uint16_t for n follow)
10909             {
10910                 std::uint16_t len{};
10911                 return get_number(input_format_t::cbor, len) &&
10912                        get_binary(input_format_t::cbor, len, result);
10913             }
10914
10915             case 0x5A: // Binary data (four-byte uint32_t for n follow)
10916             {
10917                 std::uint32_t len{};
10918                 return get_number(input_format_t::cbor, len) &&
10919                        get_binary(input_format_t::cbor, len, result);
10920             }
10921
10922             case 0x5B: // Binary data (eight-byte uint64_t for n follow)
10923             {
10924                 std::uint64_t len{};
10925                 return get_number(input_format_t::cbor, len) &&
10926                        get_binary(input_format_t::cbor, len, result);
10927             }
10928
10929             case 0x5F: // Binary data (indefinite length)
10930             {
10931                 while (get() != 0xFF)
10932                 {
10933                     binary_t chunk;
10934                     if (!get_cbor_binary(chunk))
10935                     {
10936                         return false;
10937                     }
10938                     result.insert(result.end(), chunk.begin(), chunk.end());
10939                 }
10940                 return true;
10941             }
10942
10943             default:
10944             {
10945                 auto last_token = get_token_string();
10946                 return sax->parse_error(chars_read, last_token, parse_error::create(113, chars_read,
10947                                         exception_message(input_format_t::cbor, concat("expected
      length specification (0x40-0x5B) or indefinite binary array type (0x5F); last byte: 0x", last_token),
      "binary"), nullptr));
10948             }
10949         }
10950     }
10951
10958     bool get_cbor_array(const std::size_t len,
10959                         const cbor_tag_handler_t tag_handler)
10960     {
10961         if (JSON_HEDLEY_UNLIKELY(!sax->start_array(len)))
10962         {
10963             return false;
10964         }
10965
10966         if (len != detail::unknown_size())
10967         {
10968             for (std::size_t i = 0; i < len; ++i)
```

```
10969             {
10970                 if (JSON_HEDLEY_UNLIKELY(!parse_cbor_internal(true, tag_handler)))
10971                 {
10972                     return false;
10973                 }
10974             }
10975         }
10976         else
10977         {
10978             while (get() != 0xFF)
10979             {
10980                 if (JSON_HEDLEY_UNLIKELY(!parse_cbor_internal(false, tag_handler)))
10981                 {
10982                     return false;
10983                 }
10984             }
10985         }
10986 
10987         return sax->end_array();
10988     }
10989 
10996     bool get_cbor_object(const std::size_t len,
10997                          const cbor_tag_handler_t tag_handler)
10998     {
10999         if (JSON_HEDLEY_UNLIKELY(!sax->start_object(len)))
11000         {
11001             return false;
11002         }
11003 
11004         if (len != 0)
11005         {
11006             string_t key;
11007             if (len != detail::unknown_size())
11008             {
11009                 for (std::size_t i = 0; i < len; ++i)
11010                 {
11011                     get();
11012                     if (JSON_HEDLEY_UNLIKELY(!get_cbor_string(key) || !sax->key(key)))
11013                     {
11014                         return false;
11015                     }
11016 
11017                     if (JSON_HEDLEY_UNLIKELY(!parse_cbor_internal(true, tag_handler)))
11018                     {
11019                         return false;
11020                     }
11021                     key.clear();
11022                 }
11023             }
11024             else
11025             {
11026                 while (get() != 0xFF)
11027                 {
11028                     if (JSON_HEDLEY_UNLIKELY(!get_cbor_string(key) || !sax->key(key)))
11029                     {
11030                         return false;
11031                     }
11032 
11033                     if (JSON_HEDLEY_UNLIKELY(!parse_cbor_internal(true, tag_handler)))
11034                     {
11035                         return false;
11036                     }
11037                     key.clear();
11038                 }
11039             }
11040         }
11041 
11042         return sax->end_object();
11043     }
11044 
11046     // MsgPack //
11048 
11052     bool parse_msgpack_internal()
11053     {
11054         switch (get())
11055         {
11056             // EOF
11057             case char_traits<char_type>::eof():
11058                 return unexpect_eof(input_format_t::msgpack, "value");
11059 
11060             // positive fixint
11061             case 0x00:
11062             case 0x01:
11063             case 0x02:
11064             case 0x03:
11065             case 0x04:
11066             case 0x05:
```

```
11067             case 0x06:
11068             case 0x07:
11069             case 0x08:
11070             case 0x09:
11071             case 0x0A:
11072             case 0x0B:
11073             case 0x0C:
11074             case 0x0D:
11075             case 0x0E:
11076             case 0x0F:
11077             case 0x10:
11078             case 0x11:
11079             case 0x12:
11080             case 0x13:
11081             case 0x14:
11082             case 0x15:
11083             case 0x16:
11084             case 0x17:
11085             case 0x18:
11086             case 0x19:
11087             case 0x1A:
11088             case 0x1B:
11089             case 0x1C:
11090             case 0x1D:
11091             case 0x1E:
11092             case 0x1F:
11093             case 0x20:
11094             case 0x21:
11095             case 0x22:
11096             case 0x23:
11097             case 0x24:
11098             case 0x25:
11099             case 0x26:
11100             case 0x27:
11101             case 0x28:
11102             case 0x29:
11103             case 0x2A:
11104             case 0x2B:
11105             case 0x2C:
11106             case 0x2D:
11107             case 0x2E:
11108             case 0x2F:
11109             case 0x30:
11110             case 0x31:
11111             case 0x32:
11112             case 0x33:
11113             case 0x34:
11114             case 0x35:
11115             case 0x36:
11116             case 0x37:
11117             case 0x38:
11118             case 0x39:
11119             case 0x3A:
11120             case 0x3B:
11121             case 0x3C:
11122             case 0x3D:
11123             case 0x3E:
11124             case 0x3F:
11125             case 0x40:
11126             case 0x41:
11127             case 0x42:
11128             case 0x43:
11129             case 0x44:
11130             case 0x45:
11131             case 0x46:
11132             case 0x47:
11133             case 0x48:
11134             case 0x49:
11135             case 0x4A:
11136             case 0x4B:
11137             case 0x4C:
11138             case 0x4D:
11139             case 0x4E:
11140             case 0x4F:
11141             case 0x50:
11142             case 0x51:
11143             case 0x52:
11144             case 0x53:
11145             case 0x54:
11146             case 0x55:
11147             case 0x56:
11148             case 0x57:
11149             case 0x58:
11150             case 0x59:
11151             case 0x5A:
11152             case 0x5B:
11153             case 0x5C:
```

```
11154             case 0x5D:
11155             case 0x5E:
11156             case 0x5F:
11157             case 0x60:
11158             case 0x61:
11159             case 0x62:
11160             case 0x63:
11161             case 0x64:
11162             case 0x65:
11163             case 0x66:
11164             case 0x67:
11165             case 0x68:
11166             case 0x69:
11167             case 0x6A:
11168             case 0x6B:
11169             case 0x6C:
11170             case 0x6D:
11171             case 0x6E:
11172             case 0x6F:
11173             case 0x70:
11174             case 0x71:
11175             case 0x72:
11176             case 0x73:
11177             case 0x74:
11178             case 0x75:
11179             case 0x76:
11180             case 0x77:
11181             case 0x78:
11182             case 0x79:
11183             case 0x7A:
11184             case 0x7B:
11185             case 0x7C:
11186             case 0x7D:
11187             case 0x7E:
11188             case 0x7F:
11189                 return sax->number_unsigned(static_cast<number_unsigned_t>(current));
11190
11191             // fixmap
11192             case 0x80:
11193             case 0x81:
11194             case 0x82:
11195             case 0x83:
11196             case 0x84:
11197             case 0x85:
11198             case 0x86:
11199             case 0x87:
11200             case 0x88:
11201             case 0x89:
11202             case 0x8A:
11203             case 0x8B:
11204             case 0x8C:
11205             case 0x8D:
11206             case 0x8E:
11207             case 0x8F:
11208                 return get_msgpack_object(conditional_static_cast<std::size_t>(static_cast<unsigned
      int>(current) & 0x0Fu));
11209
11210             // fixarray
11211             case 0x90:
11212             case 0x91:
11213             case 0x92:
11214             case 0x93:
11215             case 0x94:
11216             case 0x95:
11217             case 0x96:
11218             case 0x97:
11219             case 0x98:
11220             case 0x99:
11221             case 0x9A:
11222             case 0x9B:
11223             case 0x9C:
11224             case 0x9D:
11225             case 0x9E:
11226             case 0x9F:
11227                 return get_msgpack_array(conditional_static_cast<std::size_t>(static_cast<unsigned
      int>(current) & 0x0Fu));
11228
11229             // fixstr
11230             case 0xA0:
11231             case 0xA1:
11232             case 0xA2:
11233             case 0xA3:
11234             case 0xA4:
11235             case 0xA5:
11236             case 0xA6:
11237             case 0xA7:
11238             case 0xA8:
```

```
11239             case 0xA9:
11240             case 0xAA:
11241             case 0xAB:
11242             case 0xAC:
11243             case 0xAD:
11244             case 0xAE:
11245             case 0xAF:
11246             case 0xB0:
11247             case 0xB1:
11248             case 0xB2:
11249             case 0xB3:
11250             case 0xB4:
11251             case 0xB5:
11252             case 0xB6:
11253             case 0xB7:
11254             case 0xB8:
11255             case 0xB9:
11256             case 0xBA:
11257             case 0xBB:
11258             case 0xBC:
11259             case 0xBD:
11260             case 0xBE:
11261             case 0xBF:
11262             case 0xD9: // str 8
11263             case 0xDA: // str 16
11264             case 0xDB: // str 32
11265             {
11266                 string_t s;
11267                 return get_msgpack_string(s) && sax->string(s);
11268             }
11269
11270             case 0xC0: // nil
11271                 return sax->null();
11272
11273             case 0xC2: // false
11274                 return sax->boolean(false);
11275
11276             case 0xC3: // true
11277                 return sax->boolean(true);
11278
11279             case 0xC4: // bin 8
11280             case 0xC5: // bin 16
11281             case 0xC6: // bin 32
11282             case 0xC7: // ext 8
11283             case 0xC8: // ext 16
11284             case 0xC9: // ext 32
11285             case 0xD4: // fixext 1
11286             case 0xD5: // fixext 2
11287             case 0xD6: // fixext 4
11288             case 0xD7: // fixext 8
11289             case 0xD8: // fixext 16
11290             {
11291                 binary_t b;
11292                 return get_msgpack_binary(b) && sax->binary(b);
11293             }
11294
11295             case 0xCA: // float 32
11296             {
11297                 float number{};
11298                 return get_number(input_format_t::msgpack, number) &&
      sax->number_float(static_cast<number_float_t>(number), "");
11299             }
11300
11301             case 0xCB: // float 64
11302             {
11303                 double number{};
11304                 return get_number(input_format_t::msgpack, number) &&
      sax->number_float(static_cast<number_float_t>(number), "");
11305             }
11306
11307             case 0xCC: // uint 8
11308             {
11309                 std::uint8_t number{};
11310                 return get_number(input_format_t::msgpack, number) && sax->number_unsigned(number);
11311             }
11312
11313             case 0xCD: // uint 16
11314             {
11315                 std::uint16_t number{};
11316                 return get_number(input_format_t::msgpack, number) && sax->number_unsigned(number);
11317             }
11318
11319             case 0xCE: // uint 32
11320             {
11321                 std::uint32_t number{};
11322                 return get_number(input_format_t::msgpack, number) && sax->number_unsigned(number);
11323             }
```

```
11324
11325             case 0xCF: // uint 64
11326             {
11327                 std::uint64_t number{};
11328                 return get_number(input_format_t::msgpack, number) && sax->number_unsigned(number);
11329             }
11330
11331             case 0xD0: // int 8
11332             {
11333                 std::int8_t number{};
11334                 return get_number(input_format_t::msgpack, number) && sax->number_integer(number);
11335             }
11336
11337             case 0xD1: // int 16
11338             {
11339                 std::int16_t number{};
11340                 return get_number(input_format_t::msgpack, number) && sax->number_integer(number);
11341             }
11342
11343             case 0xD2: // int 32
11344             {
11345                 std::int32_t number{};
11346                 return get_number(input_format_t::msgpack, number) && sax->number_integer(number);
11347             }
11348
11349             case 0xD3: // int 64
11350             {
11351                 std::int64_t number{};
11352                 return get_number(input_format_t::msgpack, number) && sax->number_integer(number);
11353             }
11354
11355             case 0xDC: // array 16
11356             {
11357                 std::uint16_t len{};
11358                 return get_number(input_format_t::msgpack, len) &&
      get_msgpack_array(static_cast<std::size_t>(len));
11359             }
11360
11361             case 0xDD: // array 32
11362             {
11363                 std::uint32_t len{};
11364                 return get_number(input_format_t::msgpack, len) &&
      get_msgpack_array(conditional_static_cast<std::size_t>(len));
11365             }
11366
11367             case 0xDE: // map 16
11368             {
11369                 std::uint16_t len{};
11370                 return get_number(input_format_t::msgpack, len) &&
      get_msgpack_object(static_cast<std::size_t>(len));
11371             }
11372
11373             case 0xDF: // map 32
11374             {
11375                 std::uint32_t len{};
11376                 return get_number(input_format_t::msgpack, len) &&
      get_msgpack_object(conditional_static_cast<std::size_t>(len));
11377             }
11378
11379             // negative fixint
11380             case 0xE0:
11381             case 0xE1:
11382             case 0xE2:
11383             case 0xE3:
11384             case 0xE4:
11385             case 0xE5:
11386             case 0xE6:
11387             case 0xE7:
11388             case 0xE8:
11389             case 0xE9:
11390             case 0xEA:
11391             case 0xEB:
11392             case 0xEC:
11393             case 0xED:
11394             case 0xEE:
11395             case 0xEF:
11396             case 0xF0:
11397             case 0xF1:
11398             case 0xF2:
11399             case 0xF3:
11400             case 0xF4:
11401             case 0xF5:
11402             case 0xF6:
11403             case 0xF7:
11404             case 0xF8:
11405             case 0xF9:
11406             case 0xFA:
```

```
11407             case 0xFB:
11408             case 0xFC:
11409             case 0xFD:
11410             case 0xFE:
11411             case 0xFF:
11412                 return sax->number_integer(static_cast<std::int8_t>(current));
11413
11414             default: // anything else
11415             {
11416                 auto last_token = get_token_string();
11417                 return sax->parse_error(chars_read, last_token, parse_error::create(112, chars_read,
11418                                         exception_message(input_format_t::msgpack, concat("invalid
      byte: 0x", last_token), "value"), nullptr));
11419             }
11420         }
11421     }
11422
11433     bool get_msgpack_string(string_t& result)
11434     {
11435         if (JSON_HEDLEY_UNLIKELY(!unexpect_eof(input_format_t::msgpack, "string")))
11436         {
11437             return false;
11438         }
11439
11440         switch (current)
11441         {
11442             // fixstr
11443             case 0xA0:
11444             case 0xA1:
11445             case 0xA2:
11446             case 0xA3:
11447             case 0xA4:
11448             case 0xA5:
11449             case 0xA6:
11450             case 0xA7:
11451             case 0xA8:
11452             case 0xA9:
11453             case 0xAA:
11454             case 0xAB:
11455             case 0xAC:
11456             case 0xAD:
11457             case 0xAE:
11458             case 0xAF:
11459             case 0xB0:
11460             case 0xB1:
11461             case 0xB2:
11462             case 0xB3:
11463             case 0xB4:
11464             case 0xB5:
11465             case 0xB6:
11466             case 0xB7:
11467             case 0xB8:
11468             case 0xB9:
11469             case 0xBA:
11470             case 0xBB:
11471             case 0xBC:
11472             case 0xBD:
11473             case 0xBE:
11474             case 0xBF:
11475             {
11476                 return get_string(input_format_t::msgpack, static_cast<unsigned int>(current) & 0x1Fu,
      result);
11477             }
11478
11479             case 0xD9: // str 8
11480             {
11481                 std::uint8_t len{};
11482                 return get_number(input_format_t::msgpack, len) && get_string(input_format_t::msgpack,
      len, result);
11483             }
11484
11485             case 0xDA: // str 16
11486             {
11487                 std::uint16_t len{};
11488                 return get_number(input_format_t::msgpack, len) && get_string(input_format_t::msgpack,
      len, result);
11489             }
11490
11491             case 0xDB: // str 32
11492             {
11493                 std::uint32_t len{};
11494                 return get_number(input_format_t::msgpack, len) && get_string(input_format_t::msgpack,
      len, result);
11495             }
11496
11497             default:
11498             {
```

```
11499                 auto last_token = get_token_string();
11500                 return sax->parse_error(chars_read, last_token, parse_error::create(113, chars_read,
11501                                         exception_message(input_format_t::msgpack, concat("expected
      length specification (0xA0-0xBF, 0xD9-0xDB); last byte: 0x", last_token), "string"), nullptr));
11502             }
11503         }
11504     }
11505
11516     bool get_msgpack_binary(binary_t& result)
11517     {
11518         // helper function to set the subtype
11519         auto assign_and_return_true = [&result](std::int8_t subtype)
11520         {
11521             result.set_subtype(static_cast<std::uint8_t>(subtype));
11522             return true;
11523         };
11524
11525         switch (current)
11526         {
11527             case 0xC4: // bin 8
11528             {
11529                 std::uint8_t len{};
11530                 return get_number(input_format_t::msgpack, len) &&
11531                        get_binary(input_format_t::msgpack, len, result);
11532             }
11533
11534             case 0xC5: // bin 16
11535             {
11536                 std::uint16_t len{};
11537                 return get_number(input_format_t::msgpack, len) &&
11538                        get_binary(input_format_t::msgpack, len, result);
11539             }
11540
11541             case 0xC6: // bin 32
11542             {
11543                 std::uint32_t len{};
11544                 return get_number(input_format_t::msgpack, len) &&
11545                        get_binary(input_format_t::msgpack, len, result);
11546             }
11547
11548             case 0xC7: // ext 8
11549             {
11550                 std::uint8_t len{};
11551                 std::int8_t subtype{};
11552                 return get_number(input_format_t::msgpack, len) &&
11553                        get_number(input_format_t::msgpack, subtype) &&
11554                        get_binary(input_format_t::msgpack, len, result) &&
11555                        assign_and_return_true(subtype);
11556             }
11557
11558             case 0xC8: // ext 16
11559             {
11560                 std::uint16_t len{};
11561                 std::int8_t subtype{};
11562                 return get_number(input_format_t::msgpack, len) &&
11563                        get_number(input_format_t::msgpack, subtype) &&
11564                        get_binary(input_format_t::msgpack, len, result) &&
11565                        assign_and_return_true(subtype);
11566             }
11567
11568             case 0xC9: // ext 32
11569             {
11570                 std::uint32_t len{};
11571                 std::int8_t subtype{};
11572                 return get_number(input_format_t::msgpack, len) &&
11573                        get_number(input_format_t::msgpack, subtype) &&
11574                        get_binary(input_format_t::msgpack, len, result) &&
11575                        assign_and_return_true(subtype);
11576             }
11577
11578             case 0xD4: // fixext 1
11579             {
11580                 std::int8_t subtype{};
11581                 return get_number(input_format_t::msgpack, subtype) &&
11582                        get_binary(input_format_t::msgpack, 1, result) &&
11583                        assign_and_return_true(subtype);
11584             }
11585
11586             case 0xD5: // fixext 2
11587             {
11588                 std::int8_t subtype{};
11589                 return get_number(input_format_t::msgpack, subtype) &&
11590                        get_binary(input_format_t::msgpack, 2, result) &&
11591                        assign_and_return_true(subtype);
11592             }
11593
11594             case 0xD6: // fixext 4
```

```
11595             {
11596                 std::int8_t subtype{};
11597                 return get_number(input_format_t::msgpack, subtype) &&
11598                        get_binary(input_format_t::msgpack, 4, result) &&
11599                        assign_and_return_true(subtype);
11600             }
11601
11602             case 0xD7: // fixext 8
11603             {
11604                 std::int8_t subtype{};
11605                 return get_number(input_format_t::msgpack, subtype) &&
11606                        get_binary(input_format_t::msgpack, 8, result) &&
11607                        assign_and_return_true(subtype);
11608             }
11609
11610             case 0xD8: // fixext 16
11611             {
11612                 std::int8_t subtype{};
11613                 return get_number(input_format_t::msgpack, subtype) &&
11614                        get_binary(input_format_t::msgpack, 16, result) &&
11615                        assign_and_return_true(subtype);
11616             }
11617
11618             default:           // LCOV_EXCL_LINE
11619                 return false;  // LCOV_EXCL_LINE
11620         }
11621     }
11622
11627     bool get_msgpack_array(const std::size_t len)
11628     {
11629         if (JSON_HEDLEY_UNLIKELY(!sax->start_array(len)))
11630         {
11631             return false;
11632         }
11633
11634         for (std::size_t i = 0; i < len; ++i)
11635         {
11636             if (JSON_HEDLEY_UNLIKELY(!parse_msgpack_internal()))
11637             {
11638                 return false;
11639             }
11640         }
11641
11642         return sax->end_array();
11643     }
11644
11649     bool get_msgpack_object(const std::size_t len)
11650     {
11651         if (JSON_HEDLEY_UNLIKELY(!sax->start_object(len)))
11652         {
11653             return false;
11654         }
11655
11656         string_t key;
11657         for (std::size_t i = 0; i < len; ++i)
11658         {
11659             get();
11660             if (JSON_HEDLEY_UNLIKELY(!get_msgpack_string(key) || !sax->key(key)))
11661             {
11662                 return false;
11663             }
11664
11665             if (JSON_HEDLEY_UNLIKELY(!parse_msgpack_internal()))
11666             {
11667                 return false;
11668             }
11669             key.clear();
11670         }
11671
11672         return sax->end_object();
11673     }
11674
11676     // UBJSON //
11678
11686     bool parse_ubjson_internal(const bool get_char = true)
11687     {
11688         return get_ubjson_value(get_char ? get_ignore_noop() : current);
11689     }
11690
11705     bool get_ubjson_string(string_t& result, const bool get_char = true)
11706     {
11707         if (get_char)
11708         {
11709             get();  // TODO(niels): may we ignore N here?
11710         }
11711
11712         if (JSON_HEDLEY_UNLIKELY(!unexpect_eof(input_format, "value")))
```

```
11713         {
11714             return false;
11715         }
11716 
11717         switch (current)
11718         {
11719             case 'U':
11720             {
11721                 std::uint8_t len{};
11722                 return get_number(input_format, len) && get_string(input_format, len, result);
11723             }
11724 
11725             case 'i':
11726             {
11727                 std::int8_t len{};
11728                 return get_number(input_format, len) && get_string(input_format, len, result);
11729             }
11730 
11731             case 'I':
11732             {
11733                 std::int16_t len{};
11734                 return get_number(input_format, len) && get_string(input_format, len, result);
11735             }
11736 
11737             case 'l':
11738             {
11739                 std::int32_t len{};
11740                 return get_number(input_format, len) && get_string(input_format, len, result);
11741             }
11742 
11743             case 'L':
11744             {
11745                 std::int64_t len{};
11746                 return get_number(input_format, len) && get_string(input_format, len, result);
11747             }
11748 
11749             case 'u':
11750             {
11751                 if (input_format != input_format_t::bjdata)
11752                 {
11753                     break;
11754                 }
11755                 std::uint16_t len{};
11756                 return get_number(input_format, len) && get_string(input_format, len, result);
11757             }
11758 
11759             case 'm':
11760             {
11761                 if (input_format != input_format_t::bjdata)
11762                 {
11763                     break;
11764                 }
11765                 std::uint32_t len{};
11766                 return get_number(input_format, len) && get_string(input_format, len, result);
11767             }
11768 
11769             case 'M':
11770             {
11771                 if (input_format != input_format_t::bjdata)
11772                 {
11773                     break;
11774                 }
11775                 std::uint64_t len{};
11776                 return get_number(input_format, len) && get_string(input_format, len, result);
11777             }
11778 
11779             default:
11780                 break;
11781         }
11782         auto last_token = get_token_string();
11783         std::string message;
11784 
11785         if (input_format != input_format_t::bjdata)
11786         {
11787             message = "expected length type specification (U, i, I, l, L); last byte: 0x" +
      last_token;
11788         }
11789         else
11790         {
11791             message = "expected length type specification (U, i, u, I, m, l, M, L); last byte: 0x" +
      last_token;
11792         }
11793         return sax->parse_error(chars_read, last_token, parse_error::create(113, chars_read,
      exception_message(input_format, message, "string"), nullptr));
11794     }
11795 
11800     bool get_ubjson_ndarray_size(std::vector<size_t>& dim)
```

```
11801     {
11802         std::pair<std::size_t, char_int_type> size_and_type;
11803         size_t dimlen = 0;
11804         bool no_ndarray = true;
11805 
11806         if (JSON_HEDLEY_UNLIKELY(!get_ubjson_size_type(size_and_type, no_ndarray)))
11807         {
11808             return false;
11809         }
11810 
11811         if (size_and_type.first != npos)
11812         {
11813             if (size_and_type.second != 0)
11814             {
11815                 if (size_and_type.second != 'N')
11816                 {
11817                     for (std::size_t i = 0; i < size_and_type.first; ++i)
11818                     {
11819                         if (JSON_HEDLEY_UNLIKELY(!get_ubjson_size_value(dimlen, no_ndarray,
      size_and_type.second)))
11820                         {
11821                             return false;
11822                         }
11823                         dim.push_back(dimlen);
11824                     }
11825                 }
11826             }
11827             else
11828             {
11829                 for (std::size_t i = 0; i < size_and_type.first; ++i)
11830                 {
11831                     if (JSON_HEDLEY_UNLIKELY(!get_ubjson_size_value(dimlen, no_ndarray)))
11832                     {
11833                         return false;
11834                     }
11835                     dim.push_back(dimlen);
11836                 }
11837             }
11838         }
11839         else
11840         {
11841             while (current != ']')
11842             {
11843                 if (JSON_HEDLEY_UNLIKELY(!get_ubjson_size_value(dimlen, no_ndarray, current)))
11844                 {
11845                     return false;
11846                 }
11847                 dim.push_back(dimlen);
11848                 get_ignore_noop();
11849             }
11850         }
11851         return true;
11852     }
11853 
11865     bool get_ubjson_size_value(std::size_t& result, bool& is_ndarray, char_int_type prefix = 0)
11866     {
11867         if (prefix == 0)
11868         {
11869             prefix = get_ignore_noop();
11870         }
11871 
11872         switch (prefix)
11873         {
11874             case 'U':
11875             {
11876                 std::uint8_t number{};
11877                 if (JSON_HEDLEY_UNLIKELY(!get_number(input_format, number)))
11878                 {
11879                     return false;
11880                 }
11881                 result = static_cast<std::size_t>(number);
11882                 return true;
11883             }
11884 
11885             case 'i':
11886             {
11887                 std::int8_t number{};
11888                 if (JSON_HEDLEY_UNLIKELY(!get_number(input_format, number)))
11889                 {
11890                     return false;
11891                 }
11892                 if (number < 0)
11893                 {
11894                     return sax->parse_error(chars_read, get_token_string(), parse_error::create(113,
      chars_read,
11895                                             exception_message(input_format, "count in an optimized
      container must be positive", "size"), nullptr));
```

```
11896                 }
11897                 result = static_cast<std::size_t>(number); //
      NOLINT(bugprone-signed-char-misuse,cert-str34-c): number is not a char
11898                 return true;
11899             }
11900
11901             case 'I':
11902             {
11903                 std::int16_t number{};
11904                 if (JSON_HEDLEY_UNLIKELY(!get_number(input_format, number)))
11905                 {
11906                     return false;
11907                 }
11908                 if (number < 0)
11909                 {
11910                     return sax->parse_error(chars_read, get_token_string(), parse_error::create(113,
      chars_read,
11911                                             exception_message(input_format, "count in an optimized
      container must be positive", "size"), nullptr));
11912                 }
11913                 result = static_cast<std::size_t>(number);
11914                 return true;
11915             }
11916
11917             case 'l':
11918             {
11919                 std::int32_t number{};
11920                 if (JSON_HEDLEY_UNLIKELY(!get_number(input_format, number)))
11921                 {
11922                     return false;
11923                 }
11924                 if (number < 0)
11925                 {
11926                     return sax->parse_error(chars_read, get_token_string(), parse_error::create(113,
      chars_read,
11927                                             exception_message(input_format, "count in an optimized
      container must be positive", "size"), nullptr));
11928                 }
11929                 result = static_cast<std::size_t>(number);
11930                 return true;
11931             }
11932
11933             case 'L':
11934             {
11935                 std::int64_t number{};
11936                 if (JSON_HEDLEY_UNLIKELY(!get_number(input_format, number)))
11937                 {
11938                     return false;
11939                 }
11940                 if (number < 0)
11941                 {
11942                     return sax->parse_error(chars_read, get_token_string(), parse_error::create(113,
      chars_read,
11943                                             exception_message(input_format, "count in an optimized
      container must be positive", "size"), nullptr));
11944                 }
11945                 if (!value_in_range_of<std::size_t>(number))
11946                 {
11947                     return sax->parse_error(chars_read, get_token_string(), out_of_range::create(408,
11948                                             exception_message(input_format, "integer value overflow",
      "size"), nullptr));
11949                 }
11950                 result = static_cast<std::size_t>(number);
11951                 return true;
11952             }
11953
11954             case 'u':
11955             {
11956                 if (input_format != input_format_t::bjdata)
11957                 {
11958                     break;
11959                 }
11960                 std::uint16_t number{};
11961                 if (JSON_HEDLEY_UNLIKELY(!get_number(input_format, number)))
11962                 {
11963                     return false;
11964                 }
11965                 result = static_cast<std::size_t>(number);
11966                 return true;
11967             }
11968
11969             case 'm':
11970             {
11971                 if (input_format != input_format_t::bjdata)
11972                 {
11973                     break;
11974                 }
```

```
11975                 std::uint32_t number{};
11976                 if (JSON_HEDLEY_UNLIKELY(!get_number(input_format, number)))
11977                 {
11978                     return false;
11979                 }
11980                 result = conditional_static_cast<std::size_t>(number);
11981                 return true;
11982             }
11983
11984             case 'M':
11985             {
11986                 if (input_format != input_format_t::bjdata)
11987                 {
11988                     break;
11989                 }
11990                 std::uint64_t number{};
11991                 if (JSON_HEDLEY_UNLIKELY(!get_number(input_format, number)))
11992                 {
11993                     return false;
11994                 }
11995                 if (!value_in_range_of<std::size_t>(number))
11996                 {
11997                     return sax->parse_error(chars_read, get_token_string(), out_of_range::create(408,
11998                                             exception_message(input_format, "integer value overflow",
      "size"), nullptr));
11999                 }
12000                 result = detail::conditional_static_cast<std::size_t>(number);
12001                 return true;
12002             }
12003
12004             case '[':
12005             {
12006                 if (input_format != input_format_t::bjdata)
12007                 {
12008                     break;
12009                 }
12010                 if (is_ndarray) // ndarray dimensional vector can only contain integers and cannot
      embed another array
12011                 {
12012                     return sax->parse_error(chars_read, get_token_string(), parse_error::create(113,
      chars_read, exception_message(input_format, "ndarray dimensional vector is not allowed", "size"),
      nullptr));
12013                 }
12014                 std::vector<size_t> dim;
12015                 if (JSON_HEDLEY_UNLIKELY(!get_ubjson_ndarray_size(dim)))
12016                 {
12017                     return false;
12018                 }
12019                 if (dim.size() == 1 || (dim.size() == 2 && dim.at(0) == 1)) // return normal array
      size if 1D row vector
12020                 {
12021                     result = dim.at(dim.size() - 1);
12022                     return true;
12023                 }
12024                 if (!dim.empty())  // if ndarray, convert to an object in JData annotated array format
12025                 {
12026                     for (auto i : dim) // test if any dimension in an ndarray is 0, if so, return a 1D
      empty container
12027                     {
12028                         if ( i == 0 )
12029                         {
12030                             result = 0;
12031                             return true;
12032                         }
12033                     }
12034
12035                     string_t key = "_ArraySize_";
12036                     if (JSON_HEDLEY_UNLIKELY(!sax->start_object(3) || !sax->key(key) ||
      !sax->start_array(dim.size())))
12037                     {
12038                         return false;
12039                     }
12040                     result = 1;
12041                     for (auto i : dim)
12042                     {
12043                         // Pre-multiplication overflow check: if i > 0 and result > SIZE_MAX/i, then
      result*i would overflow.
12044                         // This check must happen before multiplication since overflow detection after
      the fact is unreliable
12045                         // as modular arithmetic can produce any value, not just 0 or SIZE_MAX.
12046                         if (JSON_HEDLEY_UNLIKELY(i > 0 && result >
      (std::numeric_limits<std::size_t>::max)() / i))
12047                         {
12048                             return sax->parse_error(chars_read, get_token_string(),
      out_of_range::create(408, exception_message(input_format, "excessive ndarray size caused overflow",
      "size"), nullptr));
12049                         }
```

```
12050                         result *= i;
12051                         // Additional post-multiplication check to catch any edge cases the pre-check
      might miss
12052                         if (result == 0 || result == npos)
12053                         {
12054                             return sax->parse_error(chars_read, get_token_string(),
      out_of_range::create(408, exception_message(input_format, "excessive ndarray size caused overflow",
      "size"), nullptr));
12055                         }
12056                         if
      (JSON_HEDLEY_UNLIKELY(!sax->number_unsigned(static_cast<number_unsigned_t>(i))))
12057                         {
12058                             return false;
12059                         }
12060                     }
12061                     is_ndarray = true;
12062                     return sax->end_array();
12063                 }
12064                 result = 0;
12065                 return true;
12066             }
12067
12068             default:
12069                 break;
12070         }
12071         auto last_token = get_token_string();
12072         std::string message;
12073
12074         if (input_format != input_format_t::bjdata)
12075         {
12076             message = "expected length type specification (U, i, I, l, L) after '#'; last byte: 0x" +
      last_token;
12077         }
12078         else
12079         {
12080             message = "expected length type specification (U, i, u, I, m, l, M, L) after '#'; last
      byte: 0x" + last_token;
12081         }
12082         return sax->parse_error(chars_read, last_token, parse_error::create(113, chars_read,
      exception_message(input_format, message, "size"), nullptr));
12083     }
12084
12096     bool get_ubjson_size_type(std::pair<std::size_t, char_int_type>& result, bool inside_ndarray =
      false)
12097     {
12098         result.first = npos; // size
12099         result.second = 0; // type
12100         bool is_ndarray = false;
12101
12102         get_ignore_noop();
12103
12104         if (current == '$')
12105         {
12106             result.second = get();  // must not ignore 'N', because 'N' maybe the type
12107             if (input_format == input_format_t::bjdata
12108                     && JSON_HEDLEY_UNLIKELY(std::binary_search(bjd_optimized_type_markers.begin(),
      bjd_optimized_type_markers.end(), result.second)))
12109             {
12110                 auto last_token = get_token_string();
12111                 return sax->parse_error(chars_read, last_token, parse_error::create(112, chars_read,
12112                                         exception_message(input_format, concat("marker 0x",
      last_token, " is not a permitted optimized array type"), "type"), nullptr));
12113             }
12114
12115             if (JSON_HEDLEY_UNLIKELY(!unexpect_eof(input_format, "type")))
12116             {
12117                 return false;
12118             }
12119
12120             get_ignore_noop();
12121             if (JSON_HEDLEY_UNLIKELY(current != '#'))
12122             {
12123                 if (JSON_HEDLEY_UNLIKELY(!unexpect_eof(input_format, "value")))
12124                 {
12125                     return false;
12126                 }
12127                 auto last_token = get_token_string();
12128                 return sax->parse_error(chars_read, last_token, parse_error::create(112, chars_read,
12129                                         exception_message(input_format, concat("expected '#' after
      type information; last byte: 0x", last_token), "size"), nullptr));
12130             }
12131
12132             const bool is_error = get_ubjson_size_value(result.first, is_ndarray);
12133             if (input_format == input_format_t::bjdata && is_ndarray)
12134             {
12135                 if (inside_ndarray)
12136                 {
```

```
12137                     return sax->parse_error(chars_read, get_token_string(), parse_error::create(112,
      chars_read,
12138                                                 exception_message(input_format, "ndarray can not be
      recursive", "size"), nullptr));
12139                 }
12140                 result.second |= (1 << 8); // use bit 8 to indicate ndarray, all UBJSON and BJData
      markers should be ASCII letters
12141             }
12142             return is_error;
12143         }
12144
12145         if (current == '#')
12146         {
12147             const bool is_error = get_ubjson_size_value(result.first, is_ndarray);
12148             if (input_format == input_format_t::bjdata && is_ndarray)
12149             {
12150                 return sax->parse_error(chars_read, get_token_string(), parse_error::create(112,
      chars_read,
12151                                             exception_message(input_format, "ndarray requires both type
      and size", "size"), nullptr));
12152             }
12153             return is_error;
12154         }
12155
12156         return true;
12157     }
12158
12163     bool get_ubjson_value(const char_int_type prefix)
12164     {
12165         switch (prefix)
12166         {
12167             case char_traits<char_type>::eof():  // EOF
12168                 return unexpect_eof(input_format, "value");
12169
12170             case 'T':  // true
12171                 return sax->boolean(true);
12172             case 'F':  // false
12173                 return sax->boolean(false);
12174
12175             case 'Z':  // null
12176                 return sax->null();
12177
12178             case 'B':  // byte
12179             {
12180                 if (input_format != input_format_t::bjdata)
12181                 {
12182                     break;
12183                 }
12184                 std::uint8_t number{};
12185                 return get_number(input_format, number) && sax->number_unsigned(number);
12186             }
12187
12188             case 'U':
12189             {
12190                 std::uint8_t number{};
12191                 return get_number(input_format, number) && sax->number_unsigned(number);
12192             }
12193
12194             case 'i':
12195             {
12196                 std::int8_t number{};
12197                 return get_number(input_format, number) && sax->number_integer(number);
12198             }
12199
12200             case 'I':
12201             {
12202                 std::int16_t number{};
12203                 return get_number(input_format, number) && sax->number_integer(number);
12204             }
12205
12206             case 'l':
12207             {
12208                 std::int32_t number{};
12209                 return get_number(input_format, number) && sax->number_integer(number);
12210             }
12211
12212             case 'L':
12213             {
12214                 std::int64_t number{};
12215                 return get_number(input_format, number) && sax->number_integer(number);
12216             }
12217
12218             case 'u':
12219             {
12220                 if (input_format != input_format_t::bjdata)
12221                 {
12222                     break;
```

```
12223                 }
12224                 std::uint16_t number{};
12225                 return get_number(input_format, number) && sax->number_unsigned(number);
12226             }
12227
12228             case 'm':
12229             {
12230                 if (input_format != input_format_t::bjdata)
12231                 {
12232                     break;
12233                 }
12234                 std::uint32_t number{};
12235                 return get_number(input_format, number) && sax->number_unsigned(number);
12236             }
12237
12238             case 'M':
12239             {
12240                 if (input_format != input_format_t::bjdata)
12241                 {
12242                     break;
12243                 }
12244                 std::uint64_t number{};
12245                 return get_number(input_format, number) && sax->number_unsigned(number);
12246             }
12247
12248             case 'h':
12249             {
12250                 if (input_format != input_format_t::bjdata)
12251                 {
12252                     break;
12253                 }
12254                 const auto byte1_raw = get();
12255                 if (JSON_HEDLEY_UNLIKELY(!unexpect_eof(input_format, "number")))
12256                 {
12257                     return false;
12258                 }
12259                 const auto byte2_raw = get();
12260                 if (JSON_HEDLEY_UNLIKELY(!unexpect_eof(input_format, "number")))
12261                 {
12262                     return false;
12263                 }
12264
12265                 const auto byte1 = static_cast<unsigned char>(byte1_raw);
12266                 const auto byte2 = static_cast<unsigned char>(byte2_raw);
12267
12268                 // Code from RFC 7049, Appendix D, Figure 3:
12269                 // As half-precision floating-point numbers were only added
12270                 // to IEEE 754 in 2008, today's programming platforms often
12271                 // still only have limited support for them. It is very
12272                 // easy to include at least decoding support for them even
12273                 // without such support. An example of a small decoder for
12274                 // half-precision floating-point numbers in the C language
12275                 // is shown in Fig. 3.
12276                 const auto half = static_cast<unsigned int>((byte2 « 8u) + byte1);
12277                 const double val = [&half]
12278                 {
12279                     const int exp = (half » 10u) & 0x1Fu;
12280                     const unsigned int mant = half & 0x3FFu;
12281                     JSON_ASSERT(0 <= exp&& exp <= 32);
12282                     JSON_ASSERT(mant <= 1024);
12283                     switch (exp)
12284                     {
12285                         case 0:
12286                             return std::ldexp(mant, -24);
12287                         case 31:
12288                             return (mant == 0)
12289                             ? std::numeric_limits<double>::infinity()
12290                             : std::numeric_limits<double>::quiet_NaN();
12291                         default:
12292                             return std::ldexp(mant + 1024, exp - 25);
12293                     }
12294                 }();
12295                 return sax->number_float((half & 0x8000u) != 0
12296                                          ? static_cast<number_float_t>(-val)
12297                                          : static_cast<number_float_t>(val), "");
12298             }
12299
12300             case 'd':
12301             {
12302                 float number{};
12303                 return get_number(input_format, number) &&
    sax->number_float(static_cast<number_float_t>(number), "");
12304             }
12305
12306             case 'D':
12307             {
12308                 double number{};
```

```
12309                 return get_number(input_format, number) &&
      sax->number_float(static_cast<number_float_t>(number), "");
12310             }
12311
12312             case 'H':
12313             {
12314                 return get_ubjson_high_precision_number();
12315             }
12316
12317             case 'C':  // char
12318             {
12319                 get();
12320                 if (JSON_HEDLEY_UNLIKELY(!unexpect_eof(input_format, "char")))
12321                 {
12322                     return false;
12323                 }
12324                 if (JSON_HEDLEY_UNLIKELY(current > 127))
12325                 {
12326                     auto last_token = get_token_string();
12327                     return sax->parse_error(chars_read, last_token, parse_error::create(113,
      chars_read,
12328                                             exception_message(input_format, concat("byte after 'C'
      must be in range 0x00..0x7F; last byte: 0x", last_token), "char"), nullptr));
12329                 }
12330                 string_t s(1, static_cast<typename string_t::value_type>(current));
12331                 return sax->string(s);
12332             }
12333
12334             case 'S':  // string
12335             {
12336                 string_t s;
12337                 return get_ubjson_string(s) && sax->string(s);
12338             }
12339
12340             case '[':  // array
12341                 return get_ubjson_array();
12342
12343             case '{':  // object
12344                 return get_ubjson_object();
12345
12346             default: // anything else
12347                 break;
12348         }
12349         auto last_token = get_token_string();
12350         return sax->parse_error(chars_read, last_token, parse_error::create(112, chars_read,
      exception_message(input_format, "invalid byte: 0x" + last_token, "value"), nullptr));
12351     }
12352
12356     bool get_ubjson_array()
12357     {
12358         std::pair<std::size_t, char_int_type> size_and_type;
12359         if (JSON_HEDLEY_UNLIKELY(!get_ubjson_size_type(size_and_type)))
12360         {
12361             return false;
12362         }
12363
12364         // if bit-8 of size_and_type.second is set to 1, encode bjdata ndarray as an object in JData
      annotated array format (https://github.com/NeuroJSON/jdata):
12365         // {"_ArrayType_" : "typeid", "_ArraySize_" : [n1, n2, ...], "_ArrayData_" : [v1, v2, ...]}
12366
12367         if (input_format == input_format_t::bjdata && size_and_type.first != npos &&
      (size_and_type.second & (1 << 8)) != 0)
12368         {
12369             size_and_type.second &= ~(static_cast<char_int_type>(1) << 8);  // use bit 8 to indicate
      ndarray, here we remove the bit to restore the type marker
12370             auto it = std::lower_bound(bjd_types_map.begin(), bjd_types_map.end(),
      size_and_type.second, [](const bjd_type & p, char_int_type t)
12371             {
12372                 return p.first < t;
12373             });
12374             string_t key = "_ArrayType_";
12375             if (JSON_HEDLEY_UNLIKELY(it == bjd_types_map.end() || it->first != size_and_type.second))
12376             {
12377                 auto last_token = get_token_string();
12378                 return sax->parse_error(chars_read, last_token, parse_error::create(112, chars_read,
12379                                         exception_message(input_format, "invalid byte: 0x" +
      last_token, "type"), nullptr));
12380             }
12381
12382             string_t type = it->second; // sax->string() takes a reference
12383             if (JSON_HEDLEY_UNLIKELY(!sax->key(key) || !sax->string(type)))
12384             {
12385                 return false;
12386             }
12387
12388             if (size_and_type.second == 'C' || size_and_type.second == 'B')
12389             {
```

```
12390                 size_and_type.second = 'U';
12391             }
12392
12393             key = "_ArrayData_";
12394             if (JSON_HEDLEY_UNLIKELY(!sax->key(key) || !sax->start_array(size_and_type.first) ))
12395             {
12396                 return false;
12397             }
12398
12399             for (std::size_t i = 0; i < size_and_type.first; ++i)
12400             {
12401                 if (JSON_HEDLEY_UNLIKELY(!get_ubjson_value(size_and_type.second)))
12402                 {
12403                     return false;
12404                 }
12405             }
12406
12407             return (sax->end_array() && sax->end_object());
12408         }
12409
12410         // If BJData type marker is 'B' decode as binary
12411         if (input_format == input_format_t::bjdata && size_and_type.first != npos &&
      size_and_type.second == 'B')
12412         {
12413             binary_t result;
12414             return get_binary(input_format, size_and_type.first, result) && sax->binary(result);
12415         }
12416
12417         if (size_and_type.first != npos)
12418         {
12419             if (JSON_HEDLEY_UNLIKELY(!sax->start_array(size_and_type.first)))
12420             {
12421                 return false;
12422             }
12423
12424             if (size_and_type.second != 0)
12425             {
12426                 if (size_and_type.second != 'N')
12427                 {
12428                     for (std::size_t i = 0; i < size_and_type.first; ++i)
12429                     {
12430                         if (JSON_HEDLEY_UNLIKELY(!get_ubjson_value(size_and_type.second)))
12431                         {
12432                             return false;
12433                         }
12434                     }
12435                 }
12436             }
12437             else
12438             {
12439                 for (std::size_t i = 0; i < size_and_type.first; ++i)
12440                 {
12441                     if (JSON_HEDLEY_UNLIKELY(!parse_ubjson_internal()))
12442                     {
12443                         return false;
12444                     }
12445                 }
12446             }
12447         }
12448         else
12449         {
12450             if (JSON_HEDLEY_UNLIKELY(!sax->start_array(detail::unknown_size())))
12451             {
12452                 return false;
12453             }
12454
12455             while (current != ']')
12456             {
12457                 if (JSON_HEDLEY_UNLIKELY(!parse_ubjson_internal(false)))
12458                 {
12459                     return false;
12460                 }
12461                 get_ignore_noop();
12462             }
12463         }
12464
12465         return sax->end_array();
12466     }
12467
12471     bool get_ubjson_object()
12472     {
12473         std::pair<std::size_t, char_int_type> size_and_type;
12474         if (JSON_HEDLEY_UNLIKELY(!get_ubjson_size_type(size_and_type)))
12475         {
12476             return false;
12477         }
12478
```

```
12479         // do not accept ND-array size in objects in BJData
12480         if (input_format == input_format_t::bjdata && size_and_type.first != npos &&
      (size_and_type.second & (1 << 8)) != 0)
12481         {
12482             auto last_token = get_token_string();
12483             return sax->parse_error(chars_read, last_token, parse_error::create(112, chars_read,
12484                                     exception_message(input_format, "BJData object does not support
      ND-array size in optimized format", "object"), nullptr));
12485         }
12486
12487         string_t key;
12488         if (size_and_type.first != npos)
12489         {
12490             if (JSON_HEDLEY_UNLIKELY(!sax->start_object(size_and_type.first)))
12491             {
12492                 return false;
12493             }
12494
12495             if (size_and_type.second != 0)
12496             {
12497                 for (std::size_t i = 0; i < size_and_type.first; ++i)
12498                 {
12499                     if (JSON_HEDLEY_UNLIKELY(!get_ubjson_string(key) || !sax->key(key)))
12500                     {
12501                         return false;
12502                     }
12503                     if (JSON_HEDLEY_UNLIKELY(!get_ubjson_value(size_and_type.second)))
12504                     {
12505                         return false;
12506                     }
12507                     key.clear();
12508                 }
12509             }
12510             else
12511             {
12512                 for (std::size_t i = 0; i < size_and_type.first; ++i)
12513                 {
12514                     if (JSON_HEDLEY_UNLIKELY(!get_ubjson_string(key) || !sax->key(key)))
12515                     {
12516                         return false;
12517                     }
12518                     if (JSON_HEDLEY_UNLIKELY(!parse_ubjson_internal()))
12519                     {
12520                         return false;
12521                     }
12522                     key.clear();
12523                 }
12524             }
12525         }
12526         else
12527         {
12528             if (JSON_HEDLEY_UNLIKELY(!sax->start_object(detail::unknown_size())))
12529             {
12530                 return false;
12531             }
12532
12533             while (current != '}')
12534             {
12535                 if (JSON_HEDLEY_UNLIKELY(!get_ubjson_string(key, false) || !sax->key(key)))
12536                 {
12537                     return false;
12538                 }
12539                 if (JSON_HEDLEY_UNLIKELY(!parse_ubjson_internal()))
12540                 {
12541                     return false;
12542                 }
12543                 get_ignore_noop();
12544                 key.clear();
12545             }
12546         }
12547
12548         return sax->end_object();
12549     }
12550
12551     // Note, no reader for UBJSON binary types is implemented because they do
12552     // not exist
12553
12554     bool get_ubjson_high_precision_number()
12555     {
12556         // get the size of the following number string
12557         std::size_t size{};
12558         bool no_ndarray = true;
12559         auto res = get_ubjson_size_value(size, no_ndarray);
12560         if (JSON_HEDLEY_UNLIKELY(!res))
12561         {
12562             return res;
12563         }
```

```
12564
12565         // get number string
12566         std::vector<char> number_vector;
12567         for (std::size_t i = 0; i < size; ++i)
12568         {
12569             get();
12570             if (JSON_HEDLEY_UNLIKELY(!unexpect_eof(input_format, "number")))
12571             {
12572                 return false;
12573             }
12574             number_vector.push_back(static_cast<char>(current));
12575         }
12576
12577         // parse number string
12578         using ia_type = decltype(detail::input_adapter(number_vector));
12579         auto number_lexer = detail::lexer<BasicJsonType,
      ia_type>(detail::input_adapter(number_vector), false);
12580         const auto result_number = number_lexer.scan();
12581         const auto number_string = number_lexer.get_token_string();
12582         const auto result_remainder = number_lexer.scan();
12583
12584         using token_type = typename detail::lexer_base<BasicJsonType>::token_type;
12585
12586         if (JSON_HEDLEY_UNLIKELY(result_remainder != token_type::end_of_input))
12587         {
12588             return sax->parse_error(chars_read, number_string, parse_error::create(115, chars_read,
12589                                     exception_message(input_format, concat("invalid number text: ",
      number_lexer.get_token_string()), "high-precision number"), nullptr));
12590         }
12591
12592         switch (result_number)
12593         {
12594             case token_type::value_integer:
12595                 return sax->number_integer(number_lexer.get_number_integer());
12596             case token_type::value_unsigned:
12597                 return sax->number_unsigned(number_lexer.get_number_unsigned());
12598             case token_type::value_float:
12599                 return sax->number_float(number_lexer.get_number_float(), std::move(number_string));
12600             case token_type::uninitialized:
12601             case token_type::literal_true:
12602             case token_type::literal_false:
12603             case token_type::literal_null:
12604             case token_type::value_string:
12605             case token_type::begin_array:
12606             case token_type::begin_object:
12607             case token_type::end_array:
12608             case token_type::end_object:
12609             case token_type::name_separator:
12610             case token_type::value_separator:
12611             case token_type::parse_error:
12612             case token_type::end_of_input:
12613             case token_type::literal_or_value:
12614             default:
12615                 return sax->parse_error(chars_read, number_string, parse_error::create(115,
      chars_read,
12616                                         exception_message(input_format, concat("invalid number text:
      ", number_lexer.get_token_string()), "high-precision number"), nullptr));
12617         }
12618     }
12619
12621     // Utility functions //
12623
12633     char_int_type get()
12634     {
12635         ++chars_read;
12636         return current = ia.get_character();
12637     }
12638
12647     template<class T>
12648     bool get_to(T& dest, const input_format_t format, const char* context)
12649     {
12650         auto new_chars_read = ia.get_elements(&dest);
12651         chars_read += new_chars_read;
12652         if (JSON_HEDLEY_UNLIKELY(new_chars_read < sizeof(T)))
12653         {
12654             // in case of failure, advance position by 1 to report the failing location
12655             ++chars_read;
12656             sax->parse_error(chars_read, "<end of file>", parse_error::create(110, chars_read,
      exception_message(format, "unexpected end of input", context), nullptr));
12657             return false;
12658         }
12659         return true;
12660     }
12661
12665     char_int_type get_ignore_noop()
12666     {
12667         do
```

```
12668         {
12669             get();
12670         }
12671         while (current == 'N');
12672 
12673         return current;
12674     }
12675 
12676     template<class NumberType>
12677     static void byte_swap(NumberType& number)
12678     {
12679         constexpr std::size_t sz = sizeof(number);
12680 #ifdef __cpp_lib_byteswap
12681         if constexpr (sz == 1)
12682         {
12683             return;
12684         }
12685         else if constexpr(std::is_integral_v<NumberType>)
12686         {
12687             number = std::byteswap(number);
12688             return;
12689         }
12690         else
12691         {
12692 #endif
12693             auto* ptr = reinterpret_cast<std::uint8_t*>(&number);
12694             for (std::size_t i = 0; i < sz / 2; ++i)
12695             {
12696                 std::swap(ptr[i], ptr[sz - i - 1]);
12697             }
12698 #ifdef __cpp_lib_byteswap
12699         }
12700 #endif
12701     }
12702 
12703     /*
12704     @brief read a number from the input
12705 
12706     @tparam NumberType the type of the number
12707     @param[in] format   the current format (for diagnostics)
12708     @param[out] result  number of type @a NumberType
12709 
12710     @return whether conversion completed
12711 
12712     @note This function needs to respect the system's endianness, because
12713           bytes in CBOR, MessagePack, and UBJSON are stored in network order
12714           (big endian) and therefore need reordering on little endian systems.
12715           On the other hand, BSON and BJData use little endian and should reorder
12716           on big endian systems.
12717     */
12718     template<typename NumberType, bool InputIsLittleEndian = false>
12719     bool get_number(const input_format_t format, NumberType& result)
12720     {
12721         // read in the original format
12722 
12723         if (JSON_HEDLEY_UNLIKELY(!get_to(result, format, "number")))
12724         {
12725             return false;
12726         }
12727         if (is_little_endian != (InputIsLittleEndian || format == input_format_t::bjdata))
12728         {
12729             byte_swap(result);
12730         }
12731         return true;
12732     }
12733 
12748     template<typename NumberType>
12749     bool get_string(const input_format_t format,
12750                     const NumberType len,
12751                     string_t& result)
12752     {
12753         bool success = true;
12754         for (NumberType i = 0; i < len; i++)
12755         {
12756             get();
12757             if (JSON_HEDLEY_UNLIKELY(!unexpect_eof(format, "string")))
12758             {
12759                 success = false;
12760                 break;
12761             }
12762             result.push_back(static_cast<typename string_t::value_type>(current));
12763         }
12764         return success;
12765     }
12766 
12781     template<typename NumberType>
12782     bool get_binary(const input_format_t format,
```

```
12783                     const NumberType len,
12784                     binary_t& result)
12785     {
12786         bool success = true;
12787         for (NumberType i = 0; i < len; i++)
12788         {
12789             get();
12790             if (JSON_HEDLEY_UNLIKELY(!unexpect_eof(format, "binary")))
12791             {
12792                 success = false;
12793                 break;
12794             }
12795             result.push_back(static_cast<typename binary_t::value_type>(current));
12796         }
12797         return success;
12798     }
12799
12805     JSON_HEDLEY_NON_NULL(3)
12806     bool unexpect_eof(const input_format_t format, const char* context) const
12807     {
12808         if (JSON_HEDLEY_UNLIKELY(current == char_traits<char_type>::eof()))
12809         {
12810             return sax->parse_error(chars_read, "<end of file>",
12811                                     parse_error::create(110, chars_read, exception_message(format,
      "unexpected end of input", context), nullptr));
12812         }
12813         return true;
12814     }
12815
12819     std::string get_token_string() const
12820     {
12821         std::array<char, 3> cr{{}};
12822         static_cast<void>((std::snprintf)(cr.data(), cr.size(), "%.2hhX", static_cast<unsigned
      char>(current))); // NOLINT(cppcoreguidelines-pro-type-vararg,hicpp-vararg)
12823         return std::string{cr.data()};
12824     }
12825
12832     std::string exception_message(const input_format_t format,
12833                                   const std::string& detail,
12834                                   const std::string& context) const
12835     {
12836         std::string error_msg = "syntax error while parsing ";
12837
12838         switch (format)
12839         {
12840             case input_format_t::cbor:
12841                 error_msg += "CBOR";
12842                 break;
12843
12844             case input_format_t::msgpack:
12845                 error_msg += "MessagePack";
12846                 break;
12847
12848             case input_format_t::ubjson:
12849                 error_msg += "UBJSON";
12850                 break;
12851
12852             case input_format_t::bson:
12853                 error_msg += "BSON";
12854                 break;
12855
12856             case input_format_t::bjdata:
12857                 error_msg += "BJData";
12858                 break;
12859
12860             case input_format_t::json: // LCOV_EXCL_LINE
12861             default:                   // LCOV_EXCL_LINE
12862                 JSON_ASSERT(false); // NOLINT(cert-dcl03-c,hicpp-static-assert,misc-static-assert)
      LCOV_EXCL_LINE
12863         }
12864
12865         return concat(error_msg, ' ', context, ": ", detail);
12866     }
12867
12868   private:
12869     static JSON_INLINE_VARIABLE constexpr std::size_t npos = detail::unknown_size();
12870
12872     InputAdapterType ia;
12873
12875     char_int_type current = char_traits<char_type>::eof();
12876
12878     std::size_t chars_read = 0;
12879
12881     const bool is_little_endian = little_endianness();
12882
12884     const input_format_t input_format = input_format_t::json;
12885
```

```
12887     json_sax_t* sax = nullptr;
12888
12889     // excluded markers in bjdata optimized type
12890 #define JSON_BINARY_READER_MAKE_BJD_OPTIMIZED_TYPE_MARKERS_ \
12891     make_array<char_int_type>('F', 'H', 'N', 'S', 'T', 'Z', '[', '{')
12892
12893 #define JSON_BINARY_READER_MAKE_BJD_TYPES_MAP_ \
12894     make_array<bjd_type>(                      \
12895     bjd_type{'B', "byte"},                     \
12896     bjd_type{'C', "char"},                     \
12897     bjd_type{'D', "double"},                   \
12898     bjd_type{'I', "int16"},                    \
12899     bjd_type{'L', "int64"},                    \
12900     bjd_type{'M', "uint64"},                   \
12901     bjd_type{'U', "uint8"},                    \
12902     bjd_type{'d', "single"},                   \
12903     bjd_type{'i', "int8"},                     \
12904     bjd_type{'l', "int32"},                    \
12905     bjd_type{'m', "uint32"},                   \
12906     bjd_type{'u', "uint16"})
12907
12908   JSON_PRIVATE_UNLESS_TESTED:
12909     // lookup tables
12910     // NOLINTNEXTLINE(cppcoreguidelines-non-private-member-variables-in-classes)
12911     const decltype(JSON_BINARY_READER_MAKE_BJD_OPTIMIZED_TYPE_MARKERS_) bjd_optimized_type_markers =
12912         JSON_BINARY_READER_MAKE_BJD_OPTIMIZED_TYPE_MARKERS_;
12913
12914     using bjd_type = std::pair<char_int_type, string_t>;
12915     // NOLINTNEXTLINE(cppcoreguidelines-non-private-member-variables-in-classes)
12916     const decltype(JSON_BINARY_READER_MAKE_BJD_TYPES_MAP_) bjd_types_map =
12917         JSON_BINARY_READER_MAKE_BJD_TYPES_MAP_;
12918
12919 #undef JSON_BINARY_READER_MAKE_BJD_OPTIMIZED_TYPE_MARKERS_
12920 #undef JSON_BINARY_READER_MAKE_BJD_TYPES_MAP_
12921 };
12922
12923 #ifndef JSON_HAS_CPP_17
12924     template<typename BasicJsonType, typename InputAdapterType, typename SAX>
12925     constexpr std::size_t binary_reader<BasicJsonType, InputAdapterType, SAX>::npos;
12926 #endif
12927
12928 }  // namespace detail
12929 NLOHMANN_JSON_NAMESPACE_END
12930
12931 // #include <nlohmann/detail/input/input_adapters.hpp>
12932
12933 // #include <nlohmann/detail/input/lexer.hpp>
12934
12935 // #include <nlohmann/detail/input/parser.hpp>
12936 //     __ _____ _____ _____
12937 //  __|  |   __|     |   | |  JSON for Modern C++
12938 // |  |  |__   |  |  | | | |  version 3.12.0
12939 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
12940 //
12941 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
12942 // SPDX-License-Identifier: MIT
12943
12944
12945
12946 #include <cmath> // isfinite
12947 #include <cstdint> // uint8_t
12948 #include <functional> // function
12949 #include <string> // string
12950 #include <utility> // move
12951 #include <vector> // vector
12952
12953 // #include <nlohmann/detail/exceptions.hpp>
12954
12955 // #include <nlohmann/detail/input/input_adapters.hpp>
12956
12957 // #include <nlohmann/detail/input/json_sax.hpp>
12958
12959 // #include <nlohmann/detail/input/lexer.hpp>
12960
12961 // #include <nlohmann/detail/macro_scope.hpp>
12962
12963 // #include <nlohmann/detail/meta/is_sax.hpp>
12964
12965 // #include <nlohmann/detail/string_concat.hpp>
12966
12967 // #include <nlohmann/detail/value_t.hpp>
12968
12969
12970 NLOHMANN_JSON_NAMESPACE_BEGIN
12971 namespace detail
12972 {
12974 // parser //
```

```
12976
12977 enum class parse_event_t : std::uint8_t
12978 {
12980     object_start,
12982     object_end,
12984     array_start,
12986     array_end,
12988     key,
12990     value
12991 };
12992
12993 template<typename BasicJsonType>
12994 using parser_callback_t =
12995     std::function<bool(int /*depth*/, parse_event_t /*event*/, BasicJsonType& /*parsed*/)>;
12996
13002 template<typename BasicJsonType, typename InputAdapterType>
13003 class parser
13004 {
13005     using number_integer_t = typename BasicJsonType::number_integer_t;
13006     using number_unsigned_t = typename BasicJsonType::number_unsigned_t;
13007     using number_float_t = typename BasicJsonType::number_float_t;
13008     using string_t = typename BasicJsonType::string_t;
13009     using lexer_t = lexer<BasicJsonType, InputAdapterType>;
13010     using token_type = typename lexer_t::token_type;
13011
13012   public:
13014     explicit parser(InputAdapterType&& adapter,
13015                     parser_callback_t<BasicJsonType> cb = nullptr,
13016                     const bool allow_exceptions_ = true,
13017                     const bool ignore_comments = false,
13018                     const bool ignore_trailing_commas_ = false)
13019         : callback(std::move(cb))
13020         , m_lexer(std::move(adapter), ignore_comments)
13021         , allow_exceptions(allow_exceptions_)
13022         , ignore_trailing_commas(ignore_trailing_commas_)
13023     {
13024         // read first token
13025         get_token();
13026     }
13027
13038     void parse(const bool strict, BasicJsonType& result)
13039     {
13040         if (callback)
13041         {
13042             json_sax_dom_callback_parser<BasicJsonType, InputAdapterType> sdp(result, callback,
      allow_exceptions, &m_lexer);
13043             sax_parse_internal(&sdp);
13044
13045             // in strict mode, input must be completely read
13046             if (strict && (get_token() != token_type::end_of_input))
13047             {
13048                 sdp.parse_error(m_lexer.get_position(),
13049                                 m_lexer.get_token_string(),
13050                                 parse_error::create(101, m_lexer.get_position(),
13051                                                     exception_message(token_type::end_of_input,
      "value"), nullptr));
13052             }
13053
13054             // in case of an error, return a discarded value
13055             if (sdp.is_errored())
13056             {
13057                 result = value_t::discarded;
13058                 return;
13059             }
13060
13061             // set top-level value to null if it was discarded by the callback
13062             // function
13063             if (result.is_discarded())
13064             {
13065                 result = nullptr;
13066             }
13067         }
13068         else
13069         {
13070             json_sax_dom_parser<BasicJsonType, InputAdapterType> sdp(result, allow_exceptions,
      &m_lexer);
13071             sax_parse_internal(&sdp);
13072
13073             // in strict mode, input must be completely read
13074             if (strict && (get_token() != token_type::end_of_input))
13075             {
13076                 sdp.parse_error(m_lexer.get_position(),
13077                                 m_lexer.get_token_string(),
13078                                 parse_error::create(101, m_lexer.get_position(),
      exception_message(token_type::end_of_input, "value"), nullptr));
13079             }
13080
```

```
13081             // in case of an error, return a discarded value
13082             if (sdp.is_errored())
13083             {
13084                 result = value_t::discarded;
13085                 return;
13086             }
13087         }
13088 
13089         result.assert_invariant();
13090     }
13091 
13098     bool accept(const bool strict = true)
13099     {
13100         json_sax_acceptor<BasicJsonType> sax_acceptor;
13101         return sax_parse(&sax_acceptor, strict);
13102     }
13103 
13104     template<typename SAX>
13105     JSON_HEDLEY_NON_NULL(2)
13106     bool sax_parse(SAX* sax, const bool strict = true)
13107     {
13108         (void)detail::is_sax_static_asserts<SAX, BasicJsonType> {};
13109         const bool result = sax_parse_internal(sax);
13110 
13111         // strict mode: next byte must be EOF
13112         if (result && strict && (get_token() != token_type::end_of_input))
13113         {
13114             return sax->parse_error(m_lexer.get_position(),
13115                                     m_lexer.get_token_string(),
13116                                     parse_error::create(101, m_lexer.get_position(),
      exception_message(token_type::end_of_input, "value"), nullptr));
13117         }
13118 
13119         return result;
13120     }
13121 
13122   private:
13123     template<typename SAX>
13124     JSON_HEDLEY_NON_NULL(2)
13125     bool sax_parse_internal(SAX* sax)
13126     {
13127         // stack to remember the hierarchy of structured values we are parsing
13128         // true = array; false = object
13129         std::vector<bool> states;
13130         // value to avoid a goto (see comment where set to true)
13131         bool skip_to_state_evaluation = false;
13132 
13133         while (true)
13134         {
13135             if (!skip_to_state_evaluation)
13136             {
13137                 // invariant: get_token() was called before each iteration
13138                 switch (last_token)
13139                 {
13140                     case token_type::begin_object:
13141                     {
13142                         if (JSON_HEDLEY_UNLIKELY(!sax->start_object(detail::unknown_size())))
13143                         {
13144                             return false;
13145                         }
13146 
13147                         // closing } -> we are done
13148                         if (get_token() == token_type::end_object)
13149                         {
13150                             if (JSON_HEDLEY_UNLIKELY(!sax->end_object()))
13151                             {
13152                                 return false;
13153                             }
13154                             break;
13155                         }
13156 
13157                         // parse key
13158                         if (JSON_HEDLEY_UNLIKELY(last_token != token_type::value_string))
13159                         {
13160                             return sax->parse_error(m_lexer.get_position(),
13161                                                     m_lexer.get_token_string(),
13162                                                     parse_error::create(101, m_lexer.get_position(),
      exception_message(token_type::value_string, "object key"), nullptr));
13163                         }
13164                         if (JSON_HEDLEY_UNLIKELY(!sax->key(m_lexer.get_string())))
13165                         {
13166                             return false;
13167                         }
13168 
13169                         // parse separator (:)
13170                         if (JSON_HEDLEY_UNLIKELY(get_token() != token_type::name_separator))
13171                         {
```

```
13172                             return sax->parse_error(m_lexer.get_position(),
13173                                                     m_lexer.get_token_string(),
13174                                                     parse_error::create(101, m_lexer.get_position(),
      exception_message(token_type::name_separator, "object separator"), nullptr));
13175                         }
13176
13177                         // remember we are now inside an object
13178                         states.push_back(false);
13179
13180                         // parse values
13181                         get_token();
13182                         continue;
13183                     }
13184
13185                     case token_type::begin_array:
13186                     {
13187                         if (JSON_HEDLEY_UNLIKELY(!sax->start_array(detail::unknown_size())))
13188                         {
13189                             return false;
13190                         }
13191
13192                         // closing ] -> we are done
13193                         if (get_token() == token_type::end_array)
13194                         {
13195                             if (JSON_HEDLEY_UNLIKELY(!sax->end_array()))
13196                             {
13197                                 return false;
13198                             }
13199                             break;
13200                         }
13201
13202                         // remember we are now inside an array
13203                         states.push_back(true);
13204
13205                         // parse values (no need to call get_token)
13206                         continue;
13207                     }
13208
13209                     case token_type::value_float:
13210                     {
13211                         const auto res = m_lexer.get_number_float();
13212
13213                         if (JSON_HEDLEY_UNLIKELY(!std::isfinite(res)))
13214                         {
13215                             return sax->parse_error(m_lexer.get_position(),
13216                                                     m_lexer.get_token_string(),
13217                                                     out_of_range::create(406, concat("number overflow
      parsing '", m_lexer.get_token_string(), '\''), nullptr));
13218                         }
13219
13220                         if (JSON_HEDLEY_UNLIKELY(!sax->number_float(res, m_lexer.get_string())))
13221                         {
13222                             return false;
13223                         }
13224
13225                         break;
13226                     }
13227
13228                     case token_type::literal_false:
13229                     {
13230                         if (JSON_HEDLEY_UNLIKELY(!sax->boolean(false)))
13231                         {
13232                             return false;
13233                         }
13234                         break;
13235                     }
13236
13237                     case token_type::literal_null:
13238                     {
13239                         if (JSON_HEDLEY_UNLIKELY(!sax->null()))
13240                         {
13241                             return false;
13242                         }
13243                         break;
13244                     }
13245
13246                     case token_type::literal_true:
13247                     {
13248                         if (JSON_HEDLEY_UNLIKELY(!sax->boolean(true)))
13249                         {
13250                             return false;
13251                         }
13252                         break;
13253                     }
13254
13255                     case token_type::value_integer:
13256                     {
```

```
13257                         if (JSON_HEDLEY_UNLIKELY(!sax->number_integer(m_lexer.get_number_integer())))
13258                         {
13259                             return false;
13260                         }
13261                         break;
13262                     }
13263
13264                     case token_type::value_string:
13265                     {
13266                         if (JSON_HEDLEY_UNLIKELY(!sax->string(m_lexer.get_string())))
13267                         {
13268                             return false;
13269                         }
13270                         break;
13271                     }
13272
13273                     case token_type::value_unsigned:
13274                     {
13275                         if
      (JSON_HEDLEY_UNLIKELY(!sax->number_unsigned(m_lexer.get_number_unsigned())))
13276                         {
13277                             return false;
13278                         }
13279                         break;
13280                     }
13281
13282                     case token_type::parse_error:
13283                     {
13284                         // using "uninitialized" to avoid an "expected" message
13285                         return sax->parse_error(m_lexer.get_position(),
13286                                                 m_lexer.get_token_string(),
13287                                                 parse_error::create(101, m_lexer.get_position(),
      exception_message(token_type::uninitialized, "value"), nullptr));
13288                     }
13289                     case token_type::end_of_input:
13290                     {
13291                         if (JSON_HEDLEY_UNLIKELY(m_lexer.get_position().chars_read_total == 1))
13292                         {
13293                             return sax->parse_error(m_lexer.get_position(),
13294                                                     m_lexer.get_token_string(),
13295                                                     parse_error::create(101, m_lexer.get_position(),
13296                                                             "attempting to parse an empty input; check
      that your input string or stream contains the expected JSON", nullptr));
13297                         }
13298
13299                         return sax->parse_error(m_lexer.get_position(),
13300                                                 m_lexer.get_token_string(),
13301                                                 parse_error::create(101, m_lexer.get_position(),
      exception_message(token_type::literal_or_value, "value"), nullptr));
13302                     }
13303                     case token_type::uninitialized:
13304                     case token_type::end_array:
13305                     case token_type::end_object:
13306                     case token_type::name_separator:
13307                     case token_type::value_separator:
13308                     case token_type::literal_or_value:
13309                     default: // the last token was unexpected
13310                     {
13311                         return sax->parse_error(m_lexer.get_position(),
13312                                                 m_lexer.get_token_string(),
13313                                                 parse_error::create(101, m_lexer.get_position(),
      exception_message(token_type::literal_or_value, "value"), nullptr));
13314                     }
13315                 }
13316             }
13317             else
13318             {
13319                 skip_to_state_evaluation = false;
13320             }
13321
13322             // we reached this line after we successfully parsed a value
13323             if (states.empty())
13324             {
13325                 // empty stack: we reached the end of the hierarchy: done
13326                 return true;
13327             }
13328
13329             if (states.back())  // array
13330             {
13331                 // comma -> next value
13332                 // or end of array (ignore_trailing_commas = true)
13333                 if (get_token() == token_type::value_separator)
13334                 {
13335                     // parse a new value
13336                     get_token();
13337
13338                     // if ignore_trailing_commas and last_token is ], we can continue to "closing ]"
```

```
13339                     if (!(ignore_trailing_commas && last_token == token_type::end_array))
13340                     {
13341                         continue;
13342                     }
13343                 }
13344 
13345                 // closing ]
13346                 if (JSON_HEDLEY_LIKELY(last_token == token_type::end_array))
13347                 {
13348                     if (JSON_HEDLEY_UNLIKELY(!sax->end_array()))
13349                     {
13350                         return false;
13351                     }
13352 
13353                     // We are done with this array. Before we can parse a
13354                     // new value, we need to evaluate the new state first.
13355                     // By setting skip_to_state_evaluation to false, we
13356                     // are effectively jumping to the beginning of this if.
13357                     JSON_ASSERT(!states.empty());
13358                     states.pop_back();
13359                     skip_to_state_evaluation = true;
13360                     continue;
13361                 }
13362 
13363                 return sax->parse_error(m_lexer.get_position(),
13364                                         m_lexer.get_token_string(),
13365                                         parse_error::create(101, m_lexer.get_position(),
    exception_message(token_type::end_array, "array"), nullptr));
13366             }
13367 
13368             // states.back() is false -> object
13369 
13370             // comma -> next value
13371             // or end of object (ignore_trailing_commas = true)
13372             if (get_token() == token_type::value_separator)
13373             {
13374                 get_token();
13375 
13376                 // if ignore_trailing_commas and last_token is }, we can continue to "closing }"
13377                 if (!(ignore_trailing_commas && last_token == token_type::end_object))
13378                 {
13379                     // parse key
13380                     if (JSON_HEDLEY_UNLIKELY(last_token != token_type::value_string))
13381                     {
13382                         return sax->parse_error(m_lexer.get_position(),
13383                                                 m_lexer.get_token_string(),
13384                                                 parse_error::create(101, m_lexer.get_position(),
    exception_message(token_type::value_string, "object key"), nullptr));
13385                     }
13386 
13387                     if (JSON_HEDLEY_UNLIKELY(!sax->key(m_lexer.get_string())))
13388                     {
13389                         return false;
13390                     }
13391 
13392                     // parse separator (:)
13393                     if (JSON_HEDLEY_UNLIKELY(get_token() != token_type::name_separator))
13394                     {
13395                         return sax->parse_error(m_lexer.get_position(),
13396                                                 m_lexer.get_token_string(),
13397                                                 parse_error::create(101, m_lexer.get_position(),
    exception_message(token_type::name_separator, "object separator"), nullptr));
13398                     }
13399 
13400                     // parse values
13401                     get_token();
13402                     continue;
13403                 }
13404             }
13405 
13406             // closing }
13407             if (JSON_HEDLEY_LIKELY(last_token == token_type::end_object))
13408             {
13409                 if (JSON_HEDLEY_UNLIKELY(!sax->end_object()))
13410                 {
13411                     return false;
13412                 }
13413 
13414                 // We are done with this object. Before we can parse a
13415                 // new value, we need to evaluate the new state first.
13416                 // By setting skip_to_state_evaluation to false, we
13417                 // are effectively jumping to the beginning of this if.
13418                 JSON_ASSERT(!states.empty());
13419                 states.pop_back();
13420                 skip_to_state_evaluation = true;
13421                 continue;
13422             }
```

```
13423
13424             return sax->parse_error(m_lexer.get_position(),
13425                                     m_lexer.get_token_string(),
13426                                     parse_error::create(101, m_lexer.get_position(),
    exception_message(token_type::end_object, "object"), nullptr));
13427         }
13428     }
13429
13431     token_type get_token()
13432     {
13433         return last_token = m_lexer.scan();
13434     }
13435
13436     std::string exception_message(const token_type expected, const std::string& context)
13437     {
13438         std::string error_msg = "syntax error ";
13439
13440         if (!context.empty())
13441         {
13442             error_msg += concat("while parsing ", context, ' ');
13443         }
13444
13445         error_msg += "- ";
13446
13447         if (last_token == token_type::parse_error)
13448         {
13449             error_msg += concat(m_lexer.get_error_message(), "; last read: '",
13450                                 m_lexer.get_token_string(), '\'');
13451         }
13452         else
13453         {
13454             error_msg += concat("unexpected ", lexer_t::token_type_name(last_token));
13455         }
13456
13457         if (expected != token_type::uninitialized)
13458         {
13459             error_msg += concat("; expected ", lexer_t::token_type_name(expected));
13460         }
13461
13462         return error_msg;
13463     }
13464
13465   private:
13467     const parser_callback_t<BasicJsonType> callback = nullptr;
13469     token_type last_token = token_type::uninitialized;
13471     lexer_t m_lexer;
13473     const bool allow_exceptions = true;
13475     const bool ignore_trailing_commas = false;
13476 };
13477
13478 }  // namespace detail
13479 NLOHMANN_JSON_NAMESPACE_END
13480
13481 // #include <nlohmann/detail/iterators/internal_iterator.hpp>
13482 //     __ _____ _____ _____
13483 //  __|  |   __|     |   | |  JSON for Modern C++
13484 // |  |  |__   |  |  | | | |  version 3.12.0
13485 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
13486 //
13487 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
13488 // SPDX-License-Identifier: MIT
13489
13490
13491
13492 // #include <nlohmann/detail/abi_macros.hpp>
13493
13494 // #include <nlohmann/detail/iterators/primitive_iterator.hpp>
13495 //     __ _____ _____ _____
13496 //  __|  |   __|     |   | |  JSON for Modern C++
13497 // |  |  |__   |  |  | | | |  version 3.12.0
13498 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
13499 //
13500 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
13501 // SPDX-License-Identifier: MIT
13502
13503
13504
13505 #include <cstddef> // ptrdiff_t
13506 #include <limits>  // numeric_limits
13507
13508 // #include <nlohmann/detail/macro_scope.hpp>
13509
13510
13511 NLOHMANN_JSON_NAMESPACE_BEGIN
13512 namespace detail
13513 {
13514
```

```
13515 /*
13516 @brief an iterator for primitive JSON types
13517
13518 This class models an iterator for primitive JSON types (boolean, number,
13519 string). Its only purpose is to allow the iterator/const_iterator classes
13520 to "iterate" over primitive values. Internally, the iterator is modeled by
13521 a `difference_type` variable. Value begin_value (`0`) models the begin and
13522 end_value (`1`) models past the end.
13523 */
13524 class primitive_iterator_t
13525 {
13526   private:
13527     using difference_type = std::ptrdiff_t;
13528     static constexpr difference_type begin_value = 0;
13529     static constexpr difference_type end_value = begin_value + 1;
13530
13531   JSON_PRIVATE_UNLESS_TESTED:
13533     difference_type m_it = (std::numeric_limits<std::ptrdiff_t>::min)();
13534
13535   public:
13536     constexpr difference_type get_value() const noexcept
13537     {
13538         return m_it;
13539     }
13540
13542     void set_begin() noexcept
13543     {
13544         m_it = begin_value;
13545     }
13546
13548     void set_end() noexcept
13549     {
13550         m_it = end_value;
13551     }
13552
13554     constexpr bool is_begin() const noexcept
13555     {
13556         return m_it == begin_value;
13557     }
13558
13560     constexpr bool is_end() const noexcept
13561     {
13562         return m_it == end_value;
13563     }
13564
13565     friend constexpr bool operator==(primitive_iterator_t lhs, primitive_iterator_t rhs) noexcept
13566     {
13567         return lhs.m_it == rhs.m_it;
13568     }
13569
13570     friend constexpr bool operator<(primitive_iterator_t lhs, primitive_iterator_t rhs) noexcept
13571     {
13572         return lhs.m_it < rhs.m_it;
13573     }
13574
13575     primitive_iterator_t operator+(difference_type n) noexcept
13576     {
13577         auto result = *this;
13578         result += n;
13579         return result;
13580     }
13581
13582     friend constexpr difference_type operator-(primitive_iterator_t lhs, primitive_iterator_t rhs)
      noexcept
13583     {
13584         return lhs.m_it - rhs.m_it;
13585     }
13586
13587     primitive_iterator_t& operator++() noexcept
13588     {
13589         ++m_it;
13590         return *this;
13591     }
13592
13593     primitive_iterator_t operator++(int)& noexcept // NOLINT(cert-dcl21-cpp)
13594     {
13595         auto result = *this;
13596         ++m_it;
13597         return result;
13598     }
13599
13600     primitive_iterator_t& operator--() noexcept
13601     {
13602         --m_it;
13603         return *this;
13604     }
13605
```

```
13606     primitive_iterator_t operator--(int)& noexcept // NOLINT(cert-dcl21-cpp)
13607     {
13608         auto result = *this;
13609         --m_it;
13610         return result;
13611     }
13612
13613     primitive_iterator_t& operator+=(difference_type n) noexcept
13614     {
13615         m_it += n;
13616         return *this;
13617     }
13618
13619     primitive_iterator_t& operator-=(difference_type n) noexcept
13620     {
13621         m_it -= n;
13622         return *this;
13623     }
13624 };
13625
13626 }  // namespace detail
13627 NLOHMANN_JSON_NAMESPACE_END
13628
13629
13630 NLOHMANN_JSON_NAMESPACE_BEGIN
13631 namespace detail
13632 {
13633
13640 template<typename BasicJsonType> struct internal_iterator
13641 {
13643     typename BasicJsonType::object_t::iterator object_iterator {};
13645     typename BasicJsonType::array_t::iterator array_iterator {};
13647     primitive_iterator_t primitive_iterator {};
13648 };
13649
13650 }  // namespace detail
13651 NLOHMANN_JSON_NAMESPACE_END
13652
13653 // #include <nlohmann/detail/iterators/iter_impl.hpp>
13654 //     __ _____ _____ _____
13655 //  __|  |   __|     |   | |  JSON for Modern C++
13656 // |  |  |__   |  |  | | | |  version 3.12.0
13657 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
13658 //
13659 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
13660 // SPDX-License-Identifier: MIT
13661
13662
13663
13664 #include <iterator> // iterator, random_access_iterator_tag, bidirectional_iterator_tag, advance, next
13665 #include <type_traits> // conditional, is_const, remove_const
13666
13667 // #include <nlohmann/detail/exceptions.hpp>
13668
13669 // #include <nlohmann/detail/iterators/internal_iterator.hpp>
13670
13671 // #include <nlohmann/detail/iterators/primitive_iterator.hpp>
13672
13673 // #include <nlohmann/detail/macro_scope.hpp>
13674
13675 // #include <nlohmann/detail/meta/cpp_future.hpp>
13676
13677 // #include <nlohmann/detail/meta/type_traits.hpp>
13678
13679 // #include <nlohmann/detail/value_t.hpp>
13680
13681
13682 NLOHMANN_JSON_NAMESPACE_BEGIN
13683 namespace detail
13684 {
13685
13686 // forward declare to be able to friend it later on
13687 template<typename IteratorType> class iteration_proxy;
13688 template<typename IteratorType> class iteration_proxy_value;
13689
13706 template<typename BasicJsonType>
13707 class iter_impl // NOLINT(cppcoreguidelines-special-member-functions,hicpp-special-member-functions)
13708 {
13710     using other_iter_impl = iter_impl<typename std::conditional<std::is_const<BasicJsonType>::value,
    typename std::remove_const<BasicJsonType>::type, const BasicJsonType>::type>;
13712     friend other_iter_impl;
13713     friend BasicJsonType;
13714     friend iteration_proxy<iter_impl>;
13715     friend iteration_proxy_value<iter_impl>;
13716
13717     using object_t = typename BasicJsonType::object_t;
13718     using array_t = typename BasicJsonType::array_t;
```

```
13719     // make sure BasicJsonType is basic_json or const basic_json
13720     static_assert(is_basic_json<typename std::remove_const<BasicJsonType>::type>::value,
13721                   "iter_impl only accepts (const) basic_json");
13722     // superficial check for the LegacyBidirectionalIterator named requirement
13723     static_assert(std::is_base_of<std::bidirectional_iterator_tag,
      std::bidirectional_iterator_tag>::value
13724                   &&  std::is_base_of<std::bidirectional_iterator_tag, typename
      std::iterator_traits<typename array_t::iterator>::iterator_category>::value,
13725                   "basic_json iterator assumes array and object type iterators satisfy the
      LegacyBidirectionalIterator named requirement.");
13726
13727   public:
13733     using iterator_category = std::bidirectional_iterator_tag;
13734
13736     using value_type = typename BasicJsonType::value_type;
13738     using difference_type = typename BasicJsonType::difference_type;
13740     using pointer = typename std::conditional<std::is_const<BasicJsonType>::value,
13741           typename BasicJsonType::const_pointer,
13742           typename BasicJsonType::pointer>::type;
13744     using reference =
13745         typename std::conditional<std::is_const<BasicJsonType>::value,
13746         typename BasicJsonType::const_reference,
13747         typename BasicJsonType::reference>::type;
13748
13749     iter_impl() = default;
13750     ~iter_impl() = default;
13751     iter_impl(iter_impl&&) noexcept = default;
13752     iter_impl& operator=(iter_impl&&) noexcept = default;
13753
13760     explicit iter_impl(pointer object) noexcept : m_object(object)
13761     {
13762         JSON_ASSERT(m_object != nullptr);
13763
13764         switch (m_object->m_data.m_type)
13765         {
13766             case value_t::object:
13767             {
13768                 m_it.object_iterator = typename object_t::iterator();
13769                 break;
13770             }
13771
13772             case value_t::array:
13773             {
13774                 m_it.array_iterator = typename array_t::iterator();
13775                 break;
13776             }
13777
13778             case value_t::null:
13779             case value_t::string:
13780             case value_t::boolean:
13781             case value_t::number_integer:
13782             case value_t::number_unsigned:
13783             case value_t::number_float:
13784             case value_t::binary:
13785             case value_t::discarded:
13786             default:
13787             {
13788                 m_it.primitive_iterator = primitive_iterator_t();
13789                 break;
13790             }
13791         }
13792     }
13793
13802
13810     iter_impl(const iter_impl<const BasicJsonType>& other) noexcept
13811         : m_object(other.m_object), m_it(other.m_it)
13812     {}
13813
13820     iter_impl& operator=(const iter_impl<const BasicJsonType>& other) noexcept
13821     {
13822         if (&other != this)
13823         {
13824             m_object = other.m_object;
13825             m_it = other.m_it;
13826         }
13827         return *this;
13828     }
13829
13835     iter_impl(const iter_impl<typename std::remove_const<BasicJsonType>::type>& other) noexcept
13836         : m_object(other.m_object), m_it(other.m_it)
13837     {}
13838
13845     iter_impl& operator=(const iter_impl<typename std::remove_const<BasicJsonType>::type>& other)
      noexcept // NOLINT(cert-oop54-cpp)
13846     {
13847         m_object = other.m_object;
13848         m_it = other.m_it;
```

```
13849         return *this;
13850     }
13851
13852   JSON_PRIVATE_UNLESS_TESTED:
13857     void set_begin() noexcept
13858     {
13859         JSON_ASSERT(m_object != nullptr);
13860
13861         switch (m_object->m_data.m_type)
13862         {
13863             case value_t::object:
13864             {
13865                 m_it.object_iterator = m_object->m_data.m_value.object->begin();
13866                 break;
13867             }
13868
13869             case value_t::array:
13870             {
13871                 m_it.array_iterator = m_object->m_data.m_value.array->begin();
13872                 break;
13873             }
13874
13875             case value_t::null:
13876             {
13877                 // set to end so begin()==end() is true: null is empty
13878                 m_it.primitive_iterator.set_end();
13879                 break;
13880             }
13881
13882             case value_t::string:
13883             case value_t::boolean:
13884             case value_t::number_integer:
13885             case value_t::number_unsigned:
13886             case value_t::number_float:
13887             case value_t::binary:
13888             case value_t::discarded:
13889             default:
13890             {
13891                 m_it.primitive_iterator.set_begin();
13892                 break;
13893             }
13894         }
13895     }
13896
13901     void set_end() noexcept
13902     {
13903         JSON_ASSERT(m_object != nullptr);
13904
13905         switch (m_object->m_data.m_type)
13906         {
13907             case value_t::object:
13908             {
13909                 m_it.object_iterator = m_object->m_data.m_value.object->end();
13910                 break;
13911             }
13912
13913             case value_t::array:
13914             {
13915                 m_it.array_iterator = m_object->m_data.m_value.array->end();
13916                 break;
13917             }
13918
13919             case value_t::null:
13920             case value_t::string:
13921             case value_t::boolean:
13922             case value_t::number_integer:
13923             case value_t::number_unsigned:
13924             case value_t::number_float:
13925             case value_t::binary:
13926             case value_t::discarded:
13927             default:
13928             {
13929                 m_it.primitive_iterator.set_end();
13930                 break;
13931             }
13932         }
13933     }
13934
13935   public:
13940     reference operator*() const
13941     {
13942         JSON_ASSERT(m_object != nullptr);
13943
13944         switch (m_object->m_data.m_type)
13945         {
13946             case value_t::object:
13947             {
```

```
13948                 JSON_ASSERT(m_it.object_iterator != m_object->m_data.m_value.object->end());
13949                 return m_it.object_iterator->second;
13950             }
13951
13952             case value_t::array:
13953             {
13954                 JSON_ASSERT(m_it.array_iterator != m_object->m_data.m_value.array->end());
13955                 return *m_it.array_iterator;
13956             }
13957
13958             case value_t::null:
13959                 JSON_THROW(invalid_iterator::create(214, "cannot get value", m_object));
13960
13961             case value_t::string:
13962             case value_t::boolean:
13963             case value_t::number_integer:
13964             case value_t::number_unsigned:
13965             case value_t::number_float:
13966             case value_t::binary:
13967             case value_t::discarded:
13968             default:
13969             {
13970                 if (JSON_HEDLEY_LIKELY(m_it.primitive_iterator.is_begin()))
13971                 {
13972                     return *m_object;
13973                 }
13974
13975                 JSON_THROW(invalid_iterator::create(214, "cannot get value", m_object));
13976             }
13977         }
13978     }
13979
13984     pointer operator->() const
13985     {
13986         JSON_ASSERT(m_object != nullptr);
13987
13988         switch (m_object->m_data.m_type)
13989         {
13990             case value_t::object:
13991             {
13992                 JSON_ASSERT(m_it.object_iterator != m_object->m_data.m_value.object->end());
13993                 return &(m_it.object_iterator->second);
13994             }
13995
13996             case value_t::array:
13997             {
13998                 JSON_ASSERT(m_it.array_iterator != m_object->m_data.m_value.array->end());
13999                 return &*m_it.array_iterator;
14000             }
14001
14002             case value_t::null:
14003             case value_t::string:
14004             case value_t::boolean:
14005             case value_t::number_integer:
14006             case value_t::number_unsigned:
14007             case value_t::number_float:
14008             case value_t::binary:
14009             case value_t::discarded:
14010             default:
14011             {
14012                 if (JSON_HEDLEY_LIKELY(m_it.primitive_iterator.is_begin()))
14013                 {
14014                     return m_object;
14015                 }
14016
14017                 JSON_THROW(invalid_iterator::create(214, "cannot get value", m_object));
14018             }
14019         }
14020     }
14021
14026     iter_impl operator++(int)& // NOLINT(cert-dcl21-cpp)
14027     {
14028         auto result = *this;
14029         ++(*this);
14030         return result;
14031     }
14032
14037     iter_impl& operator++()
14038     {
14039         JSON_ASSERT(m_object != nullptr);
14040
14041         switch (m_object->m_data.m_type)
14042         {
14043             case value_t::object:
14044             {
14045                 std::advance(m_it.object_iterator, 1);
14046                 break;
```

```
14047             }
14048
14049             case value_t::array:
14050             {
14051                 std::advance(m_it.array_iterator, 1);
14052                 break;
14053             }
14054
14055             case value_t::null:
14056             case value_t::string:
14057             case value_t::boolean:
14058             case value_t::number_integer:
14059             case value_t::number_unsigned:
14060             case value_t::number_float:
14061             case value_t::binary:
14062             case value_t::discarded:
14063             default:
14064             {
14065                 ++m_it.primitive_iterator;
14066                 break;
14067             }
14068         }
14069
14070         return *this;
14071     }
14072
14077     iter_impl operator--(int)& // NOLINT(cert-dcl21-cpp)
14078     {
14079         auto result = *this;
14080         --(*this);
14081         return result;
14082     }
14083
14088     iter_impl& operator--()
14089     {
14090         JSON_ASSERT(m_object != nullptr);
14091
14092         switch (m_object->m_data.m_type)
14093         {
14094             case value_t::object:
14095             {
14096                 std::advance(m_it.object_iterator, -1);
14097                 break;
14098             }
14099
14100             case value_t::array:
14101             {
14102                 std::advance(m_it.array_iterator, -1);
14103                 break;
14104             }
14105
14106             case value_t::null:
14107             case value_t::string:
14108             case value_t::boolean:
14109             case value_t::number_integer:
14110             case value_t::number_unsigned:
14111             case value_t::number_float:
14112             case value_t::binary:
14113             case value_t::discarded:
14114             default:
14115             {
14116                 --m_it.primitive_iterator;
14117                 break;
14118             }
14119         }
14120
14121         return *this;
14122     }
14123
14128     template < typename IterImpl, detail::enable_if_t < (std::is_same<IterImpl, iter_impl>::value ||
      std::is_same<IterImpl, other_iter_impl>::value), std::nullptr_t > = nullptr >
14129     bool operator==(const IterImpl& other) const
14130     {
14131         // if objects are not the same, the comparison is undefined
14132         if (JSON_HEDLEY_UNLIKELY(m_object != other.m_object))
14133         {
14134             JSON_THROW(invalid_iterator::create(212, "cannot compare iterators of different
      containers", m_object));
14135         }
14136
14137         // value-initialized forward iterators can be compared, and must compare equal to other
      value-initialized iterators of the same type #4493
14138         if (m_object == nullptr)
14139         {
14140             return true;
14141         }
14142
```

```
14143         switch (m_object->m_data.m_type)
14144         {
14145             case value_t::object:
14146                 return (m_it.object_iterator == other.m_it.object_iterator);
14147
14148             case value_t::array:
14149                 return (m_it.array_iterator == other.m_it.array_iterator);
14150
14151             case value_t::null:
14152             case value_t::string:
14153             case value_t::boolean:
14154             case value_t::number_integer:
14155             case value_t::number_unsigned:
14156             case value_t::number_float:
14157             case value_t::binary:
14158             case value_t::discarded:
14159             default:
14160                 return (m_it.primitive_iterator == other.m_it.primitive_iterator);
14161         }
14162     }
14163
14168     template < typename IterImpl, detail::enable_if_t < (std::is_same<IterImpl, iter_impl>::value ||
      std::is_same<IterImpl, other_iter_impl>::value), std::nullptr_t > = nullptr >
14169     bool operator!=(const IterImpl& other) const
14170     {
14171         return !operator==(other);
14172     }
14173
14178     bool operator<(const iter_impl& other) const
14179     {
14180         // if objects are not the same, the comparison is undefined
14181         if (JSON_HEDLEY_UNLIKELY(m_object != other.m_object))
14182         {
14183             JSON_THROW(invalid_iterator::create(212, "cannot compare iterators of different
      containers", m_object));
14184         }
14185
14186         // value-initialized forward iterators can be compared, and must compare equal to other
      value-initialized iterators of the same type #4493
14187         if (m_object == nullptr)
14188         {
14189             // the iterators are both value-initialized and are to be considered equal, but this
      function checks for smaller, so we return false
14190             return false;
14191         }
14192
14193         switch (m_object->m_data.m_type)
14194         {
14195             case value_t::object:
14196                 JSON_THROW(invalid_iterator::create(213, "cannot compare order of object iterators",
      m_object));
14197
14198             case value_t::array:
14199                 return (m_it.array_iterator < other.m_it.array_iterator);
14200
14201             case value_t::null:
14202             case value_t::string:
14203             case value_t::boolean:
14204             case value_t::number_integer:
14205             case value_t::number_unsigned:
14206             case value_t::number_float:
14207             case value_t::binary:
14208             case value_t::discarded:
14209             default:
14210                 return (m_it.primitive_iterator < other.m_it.primitive_iterator);
14211         }
14212     }
14213
14218     bool operator<=(const iter_impl& other) const
14219     {
14220         return !other.operator < (*this);
14221     }
14222
14227     bool operator>(const iter_impl& other) const
14228     {
14229         return !operator<=(other);
14230     }
14231
14236     bool operator>=(const iter_impl& other) const
14237     {
14238         return !operator<(other);
14239     }
14240
14245     iter_impl& operator+=(difference_type i)
14246     {
14247         JSON_ASSERT(m_object != nullptr);
14248
```

```
14249         switch (m_object->m_data.m_type)
14250         {
14251             case value_t::object:
14252                 JSON_THROW(invalid_iterator::create(209, "cannot use offsets with object iterators",
      m_object));
14253
14254             case value_t::array:
14255             {
14256                 std::advance(m_it.array_iterator, i);
14257                 break;
14258             }
14259
14260             case value_t::null:
14261             case value_t::string:
14262             case value_t::boolean:
14263             case value_t::number_integer:
14264             case value_t::number_unsigned:
14265             case value_t::number_float:
14266             case value_t::binary:
14267             case value_t::discarded:
14268             default:
14269             {
14270                 m_it.primitive_iterator += i;
14271                 break;
14272             }
14273         }
14274
14275         return *this;
14276     }
14277
14282     iter_impl& operator-=(difference_type i)
14283     {
14284         return operator+=(-i);
14285     }
14286
14291     iter_impl operator+(difference_type i) const
14292     {
14293         auto result = *this;
14294         result += i;
14295         return result;
14296     }
14297
14302     friend iter_impl operator+(difference_type i, const iter_impl& it)
14303     {
14304         auto result = it;
14305         result += i;
14306         return result;
14307     }
14308
14313     iter_impl operator-(difference_type i) const
14314     {
14315         auto result = *this;
14316         result -= i;
14317         return result;
14318     }
14319
14324     difference_type operator-(const iter_impl& other) const
14325     {
14326         JSON_ASSERT(m_object != nullptr);
14327
14328         switch (m_object->m_data.m_type)
14329         {
14330             case value_t::object:
14331                 JSON_THROW(invalid_iterator::create(209, "cannot use offsets with object iterators",
      m_object));
14332
14333             case value_t::array:
14334                 return m_it.array_iterator - other.m_it.array_iterator;
14335
14336             case value_t::null:
14337             case value_t::string:
14338             case value_t::boolean:
14339             case value_t::number_integer:
14340             case value_t::number_unsigned:
14341             case value_t::number_float:
14342             case value_t::binary:
14343             case value_t::discarded:
14344             default:
14345                 return m_it.primitive_iterator - other.m_it.primitive_iterator;
14346         }
14347     }
14348
14353     reference operator[](difference_type n) const
14354     {
14355         JSON_ASSERT(m_object != nullptr);
14356
14357         switch (m_object->m_data.m_type)
```

```
14358         {
14359             case value_t::object:
14360                 JSON_THROW(invalid_iterator::create(208, "cannot use operator[] for object iterators",
      m_object));
14361
14362             case value_t::array:
14363                 return *std::next(m_it.array_iterator, n);
14364
14365             case value_t::null:
14366                 JSON_THROW(invalid_iterator::create(214, "cannot get value", m_object));
14367
14368             case value_t::string:
14369             case value_t::boolean:
14370             case value_t::number_integer:
14371             case value_t::number_unsigned:
14372             case value_t::number_float:
14373             case value_t::binary:
14374             case value_t::discarded:
14375             default:
14376             {
14377                 if (JSON_HEDLEY_LIKELY(m_it.primitive_iterator.get_value() == -n))
14378                 {
14379                     return *m_object;
14380                 }
14381
14382                 JSON_THROW(invalid_iterator::create(214, "cannot get value", m_object));
14383             }
14384         }
14385     }
14386
14391     const typename object_t::key_type& key() const
14392     {
14393         JSON_ASSERT(m_object != nullptr);
14394
14395         if (JSON_HEDLEY_LIKELY(m_object->is_object()))
14396         {
14397             return m_it.object_iterator->first;
14398         }
14399
14400         JSON_THROW(invalid_iterator::create(207, "cannot use key() for non-object iterators",
      m_object));
14401     }
14402
14407     reference value() const
14408     {
14409         return operator*();
14410     }
14411
14412   JSON_PRIVATE_UNLESS_TESTED:
14414     pointer m_object = nullptr;
14416     internal_iterator<typename std::remove_const<BasicJsonType>::type> m_it {};
14417 };
14418
14419 }  // namespace detail
14420 NLOHMANN_JSON_NAMESPACE_END
14421
14422 // #include <nlohmann/detail/iterators/iteration_proxy.hpp>
14423
14424 // #include <nlohmann/detail/iterators/json_reverse_iterator.hpp>
14425 //      __ _____ _____ _____
14426 //   __|  |   __|     |   | |  JSON for Modern C++
14427 // |  |  |__   |  |  | | | |  version 3.12.0
14428 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
14429 //
14430 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
14431 // SPDX-License-Identifier: MIT
14432
14433
14434
14435 #include <cstddef> // ptrdiff_t
14436 #include <iterator> // reverse_iterator
14437 #include <utility> // declval
14438
14439 // #include <nlohmann/detail/abi_macros.hpp>
14440
14441
14442 NLOHMANN_JSON_NAMESPACE_BEGIN
14443 namespace detail
14444 {
14445
14447 // reverse_iterator //
14449
14468 template<typename Base>
14469 class json_reverse_iterator : public std::reverse_iterator<Base>
14470 {
14471   public:
14472     using difference_type = std::ptrdiff_t;
```

```
14474     using base_iterator = std::reverse_iterator<Base>;
14476     using reference = typename Base::reference;
14477
14479     explicit json_reverse_iterator(const typename base_iterator::iterator_type& it) noexcept
14480         : base_iterator(it) {}
14481
14483     explicit json_reverse_iterator(const base_iterator& it) noexcept : base_iterator(it) {}
14484
14486     json_reverse_iterator operator++(int)& // NOLINT(cert-dcl21-cpp)
14487     {
14488         return static_cast<json_reverse_iterator>(base_iterator::operator++(1));
14489     }
14490
14492     json_reverse_iterator& operator++()
14493     {
14494         return static_cast<json_reverse_iterator&>(base_iterator::operator++());
14495     }
14496
14498     json_reverse_iterator operator--(int)& // NOLINT(cert-dcl21-cpp)
14499     {
14500         return static_cast<json_reverse_iterator>(base_iterator::operator--(1));
14501     }
14502
14504     json_reverse_iterator& operator--()
14505     {
14506         return static_cast<json_reverse_iterator&>(base_iterator::operator--());
14507     }
14508
14510     json_reverse_iterator& operator+=(difference_type i)
14511     {
14512         return static_cast<json_reverse_iterator&>(base_iterator::operator+=(i));
14513     }
14514
14516     json_reverse_iterator operator+(difference_type i) const
14517     {
14518         return static_cast<json_reverse_iterator>(base_iterator::operator+(i));
14519     }
14520
14522     json_reverse_iterator operator-(difference_type i) const
14523     {
14524         return static_cast<json_reverse_iterator>(base_iterator::operator-(i));
14525     }
14526
14528     difference_type operator-(const json_reverse_iterator& other) const
14529     {
14530         return base_iterator(*this) - base_iterator(other);
14531     }
14532
14534     reference operator[](difference_type n) const
14535     {
14536         return *(this->operator+(n));
14537     }
14538
14540     auto key() const -> decltype(std::declval<Base>().key())
14541     {
14542         auto it = --this->base();
14543         return it.key();
14544     }
14545
14547     reference value() const
14548     {
14549         auto it = --this->base();
14550         return it.operator * ();
14551     }
14552 };
14553
14554 }  // namespace detail
14555 NLOHMANN_JSON_NAMESPACE_END
14556
14557 // #include <nlohmann/detail/iterators/primitive_iterator.hpp>
14558
14559 // #include <nlohmann/detail/json_custom_base_class.hpp>
14560 //      __ _____ _____ _____
14561 //   __|  |   __|     |   | |  JSON for Modern C++
14562 // |  |  |__   |  |  | | | |  version 3.12.0
14563 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
14564 //
14565 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
14566 // SPDX-License-Identifier: MIT
14567
14568
14569
14570 #include <type_traits> // conditional, is_same
14571
14572 // #include <nlohmann/detail/abi_macros.hpp>
14573
14574
```

```
14575 NLOHMANN_JSON_NAMESPACE_BEGIN
14576 namespace detail
14577 {
14578
14589 struct json_default_base {};
14590
14591 template<class T>
14592 using json_base_class = typename std::conditional <
14593                         std::is_same<T, void>::value,
14594                         json_default_base,
14595                         T
14596                         >::type;
14597
14598 }  // namespace detail
14599 NLOHMANN_JSON_NAMESPACE_END
14600
14601 // #include <nlohmann/detail/json_pointer.hpp>
14602 //      __ _____ _____ _____
14603 //   __|  |   __|     |   | |  JSON for Modern C++
14604 // |  |  |__   |  |  | | | |  version 3.12.0
14605 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
14606 //
14607 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
14608 // SPDX-License-Identifier: MIT
14609
14610
14611
14612 #include <algorithm> // all_of
14613 #include <cctype> // isdigit
14614 #include <cerrno> // errno, ERANGE
14615 #include <cstdlib> // strtoull
14616 #ifndef JSON_NO_IO
14617     #include <iosfwd> // ostream
14618 #endif  // JSON_NO_IO
14619 #include <limits> // max
14620 #include <numeric> // accumulate
14621 #include <string> // string
14622 #include <utility> // move
14623 #include <vector> // vector
14624
14625 // #include <nlohmann/detail/exceptions.hpp>
14626
14627 // #include <nlohmann/detail/macro_scope.hpp>
14628
14629 // #include <nlohmann/detail/string_concat.hpp>
14630
14631 // #include <nlohmann/detail/string_escape.hpp>
14632
14633 // #include <nlohmann/detail/value_t.hpp>
14634
14635
14636 NLOHMANN_JSON_NAMESPACE_BEGIN
14637
14640 template<typename RefStringType>
14641 class json_pointer
14642 {
14643     // allow basic_json to access private members
14644     NLOHMANN_BASIC_JSON_TPL_DECLARATION
14645     friend class basic_json;
14646
14647     template<typename>
14648     friend class json_pointer;
14649
14650     template<typename T>
14651     struct string_t_helper
14652     {
14653         using type = T;
14654     };
14655
14656     NLOHMANN_BASIC_JSON_TPL_DECLARATION
14657     struct string_t_helper<NLOHMANN_BASIC_JSON_TPL>
14658     {
14659         using type = StringType;
14660     };
14661
14662   public:
14663     // for backwards compatibility accept BasicJsonType
14664     using string_t = typename string_t_helper<RefStringType>::type;
14665
14668     explicit json_pointer(const string_t& s = "")
14669         : reference_tokens(split(s))
14670     {}
14671
14674     string_t to_string() const
14675     {
14676         return std::accumulate(reference_tokens.begin(), reference_tokens.end(),
14677                                string_t{},
```

```
14678                                [](const string_t& a, const string_t& b)
14679         {
14680             return detail::concat(a, '/', detail::escape(b));
14681         });
14682     }
14683
14686     JSON_HEDLEY_DEPRECATED_FOR(3.11.0, to_string())
14687     operator string_t() const
14688     {
14689         return to_string();
14690     }
14691
14692 #ifndef JSON_NO_IO
14695     friend std::ostream& operator<<(std::ostream& o, const json_pointer& ptr)
14696     {
14697         o << ptr.to_string();
14698         return o;
14699     }
14700 #endif
14701
14704     json_pointer& operator/=(const json_pointer& ptr)
14705     {
14706         reference_tokens.insert(reference_tokens.end(),
14707                                 ptr.reference_tokens.begin(),
14708                                 ptr.reference_tokens.end());
14709         return *this;
14710     }
14711
14714     json_pointer& operator/=(string_t token)
14715     {
14716         push_back(std::move(token));
14717         return *this;
14718     }
14719
14722     json_pointer& operator/=(std::size_t array_idx)
14723     {
14724         return *this /= std::to_string(array_idx);
14725     }
14726
14729     friend json_pointer operator/(const json_pointer& lhs,
14730                                   const json_pointer& rhs)
14731     {
14732         return json_pointer(lhs) /= rhs;
14733     }
14734
14737     friend json_pointer operator/(const json_pointer& lhs, string_t token) //
      NOLINT(performance-unnecessary-value-param)
14738     {
14739         return json_pointer(lhs) /= std::move(token);
14740     }
14741
14744     friend json_pointer operator/(const json_pointer& lhs, std::size_t array_idx)
14745     {
14746         return json_pointer(lhs) /= array_idx;
14747     }
14748
14751     json_pointer parent_pointer() const
14752     {
14753         if (empty())
14754         {
14755             return *this;
14756         }
14757
14758         json_pointer res = *this;
14759         res.pop_back();
14760         return res;
14761     }
14762
14765     void pop_back()
14766     {
14767         if (JSON_HEDLEY_UNLIKELY(empty()))
14768         {
14769             JSON_THROW(detail::out_of_range::create(405, "JSON pointer has no parent", nullptr));
14770         }
14771
14772         reference_tokens.pop_back();
14773     }
14774
14777     const string_t& back() const
14778     {
14779         if (JSON_HEDLEY_UNLIKELY(empty()))
14780         {
14781             JSON_THROW(detail::out_of_range::create(405, "JSON pointer has no parent", nullptr));
14782         }
14783
14784         return reference_tokens.back();
14785     }
```

```
14786
14789     void push_back(const string_t& token)
14790     {
14791         reference_tokens.push_back(token);
14792     }
14793
14796     void push_back(string_t&& token)
14797     {
14798         reference_tokens.push_back(std::move(token));
14799     }
14800
14803     bool empty() const noexcept
14804     {
14805         return reference_tokens.empty();
14806     }
14807
14808   private:
14819     template<typename BasicJsonType>
14820     static typename BasicJsonType::size_type array_index(const string_t& s)
14821     {
14822         using size_type = typename BasicJsonType::size_type;
14823
14824         // error condition (cf. RFC 6901, Sect. 4)
14825         if (JSON_HEDLEY_UNLIKELY(s.size() > 1 && s[0] == '0'))
14826         {
14827             JSON_THROW(detail::parse_error::create(106, 0, detail::concat("array index '", s, "' must
      not begin with '0'"), nullptr));
14828         }
14829
14830         // error condition (cf. RFC 6901, Sect. 4)
14831         if (JSON_HEDLEY_UNLIKELY(s.size() > 1 && !(s[0] >= '1' && s[0] <= '9')))
14832         {
14833             JSON_THROW(detail::parse_error::create(109, 0, detail::concat("array index '", s, "' is
      not a number"), nullptr));
14834         }
14835
14836         const char* p = s.c_str();
14837         char* p_end = nullptr; // NOLINT(misc-const-correctness)
14838         errno = 0; // strtoull doesn't reset errno
14839         const unsigned long long res = std::strtoull(p, &p_end, 10); // NOLINT(runtime/int)
14840         if (p == p_end // invalid input or empty string
14841                 || errno == ERANGE // out of range
14842                 || JSON_HEDLEY_UNLIKELY(static_cast<std::size_t>(p_end - p) != s.size())) //
      incomplete read
14843         {
14844             JSON_THROW(detail::out_of_range::create(404, detail::concat("unresolved reference token
      '", s, "'"), nullptr));
14845         }
14846
14847         // only triggered on special platforms (like 32bit), see also
14848         // https://github.com/nlohmann/json/pull/2203
14849         if (res >= static_cast<unsigned long long>((std::numeric_limits<size_type>::max)()))  //
      NOLINT(runtime/int)
14850         {
14851             JSON_THROW(detail::out_of_range::create(410, detail::concat("array index ", s, " exceeds
      size_type"), nullptr));   // LCOV_EXCL_LINE
14852         }
14853
14854         return static_cast<size_type>(res);
14855     }
14856
14857   JSON_PRIVATE_UNLESS_TESTED:
14858     json_pointer top() const
14859     {
14860         if (JSON_HEDLEY_UNLIKELY(empty()))
14861         {
14862             JSON_THROW(detail::out_of_range::create(405, "JSON pointer has no parent", nullptr));
14863         }
14864
14865         json_pointer result = *this;
14866         result.reference_tokens = {reference_tokens[0]};
14867         return result;
14868     }
14869
14870   private:
14879     template<typename BasicJsonType>
14880     BasicJsonType& get_and_create(BasicJsonType& j) const
14881     {
14882         auto* result = &j;
14883
14884         // in case no reference tokens exist, return a reference to the JSON value
14885         // j which will be overwritten by a primitive value
14886         for (const auto& reference_token : reference_tokens)
14887         {
14888             switch (result->type())
14889             {
14890                 case detail::value_t::null:
```

```
14891                 {
14892                     if (reference_token == "0")
14893                     {
14894                         // start a new array if the reference token is 0
14895                         result = &result->operator[](0);
14896                     }
14897                     else
14898                     {
14899                         // start a new object otherwise
14900                         result = &result->operator[](reference_token);
14901                     }
14902                     break;
14903                 }
14904 
14905                 case detail::value_t::object:
14906                 {
14907                     // create an entry in the object
14908                     result = &result->operator[](reference_token);
14909                     break;
14910                 }
14911 
14912                 case detail::value_t::array:
14913                 {
14914                     // create an entry in the array
14915                     result = &result->operator[](array_index<BasicJsonType>(reference_token));
14916                     break;
14917                 }
14918 
14919                 /*
14920                 The following code is only reached if there exists a reference
14921                 token _and_ the current value is primitive. In this case, we have
14922                 an error situation, because primitive values may only occur as
14923                 a single value; that is, with an empty list of reference tokens.
14924                 */
14925                 case detail::value_t::string:
14926                 case detail::value_t::boolean:
14927                 case detail::value_t::number_integer:
14928                 case detail::value_t::number_unsigned:
14929                 case detail::value_t::number_float:
14930                 case detail::value_t::binary:
14931                 case detail::value_t::discarded:
14932                 default:
14933                     JSON_THROW(detail::type_error::create(313, "invalid value to unflatten", &j));
14934             }
14935         }
14936 
14937         return *result;
14938     }
14939 
14959     template<typename BasicJsonType>
14960     BasicJsonType& get_unchecked(BasicJsonType* ptr) const
14961     {
14962         for (const auto& reference_token : reference_tokens)
14963         {
14964             // convert null values to arrays or objects before continuing
14965             if (ptr->is_null())
14966             {
14967                 // check if the reference token is a number
14968                 const bool nums =
14969                     std::all_of(reference_token.begin(), reference_token.end(),
14970                                 [](const unsigned char x)
14971                 {
14972                     return std::isdigit(x);
14973                 });
14974 
14975                 // change value to an array for numbers or "-" or to object otherwise
14976                 *ptr = (nums || reference_token == "-")
14977                        ? detail::value_t::array
14978                        : detail::value_t::object;
14979             }
14980 
14981             switch (ptr->type())
14982             {
14983                 case detail::value_t::object:
14984                 {
14985                     // use unchecked object access
14986                     ptr = &ptr->operator[](reference_token);
14987                     break;
14988                 }
14989 
14990                 case detail::value_t::array:
14991                 {
14992                     if (reference_token == "-")
14993                     {
14994                         // explicitly treat "-" as index beyond the end
14995                         ptr = &ptr->operator[](ptr->m_data.m_value.array->size());
14996                     }
```

```
14997                     else
14998                     {
14999                         // convert array index to number; unchecked access
15000                         ptr = &ptr->operator[](array_index<BasicJsonType>(reference_token));
15001                     }
15002                     break;
15003                 }
15004 
15005                 case detail::value_t::null:
15006                 case detail::value_t::string:
15007                 case detail::value_t::boolean:
15008                 case detail::value_t::number_integer:
15009                 case detail::value_t::number_unsigned:
15010                 case detail::value_t::number_float:
15011                 case detail::value_t::binary:
15012                 case detail::value_t::discarded:
15013                 default:
15014                     JSON_THROW(detail::out_of_range::create(404, detail::concat("unresolved reference
      token '", reference_token, "'"), ptr));
15015             }
15016         }
15017 
15018         return *ptr;
15019     }
15020 
15027     template<typename BasicJsonType>
15028     BasicJsonType& get_checked(BasicJsonType* ptr) const
15029     {
15030         for (const auto& reference_token : reference_tokens)
15031         {
15032             switch (ptr->type())
15033             {
15034                 case detail::value_t::object:
15035                 {
15036                     // note: at performs range check
15037                     ptr = &ptr->at(reference_token);
15038                     break;
15039                 }
15040 
15041                 case detail::value_t::array:
15042                 {
15043                     if (JSON_HEDLEY_UNLIKELY(reference_token == "-"))
15044                     {
15045                         // "-" always fails the range check
15046                         JSON_THROW(detail::out_of_range::create(402, detail::concat(
15047                                 "array index '-' (",
      std::to_string(ptr->m_data.m_value.array->size()),
15048                                 ") is out of range"), ptr));
15049                     }
15050 
15051                     // note: at performs range check
15052                     ptr = &ptr->at(array_index<BasicJsonType>(reference_token));
15053                     break;
15054                 }
15055 
15056                 case detail::value_t::null:
15057                 case detail::value_t::string:
15058                 case detail::value_t::boolean:
15059                 case detail::value_t::number_integer:
15060                 case detail::value_t::number_unsigned:
15061                 case detail::value_t::number_float:
15062                 case detail::value_t::binary:
15063                 case detail::value_t::discarded:
15064                 default:
15065                     JSON_THROW(detail::out_of_range::create(404, detail::concat("unresolved reference
      token '", reference_token, "'"), ptr));
15066             }
15067         }
15068 
15069         return *ptr;
15070     }
15071 
15085     template<typename BasicJsonType>
15086     const BasicJsonType& get_unchecked(const BasicJsonType* ptr) const
15087     {
15088         for (const auto& reference_token : reference_tokens)
15089         {
15090             switch (ptr->type())
15091             {
15092                 case detail::value_t::object:
15093                 {
15094                     // use unchecked object access
15095                     ptr = &ptr->operator[](reference_token);
15096                     break;
15097                 }
15098 
15099                 case detail::value_t::array:
```

```
15100                 {
15101                     if (JSON_HEDLEY_UNLIKELY(reference_token == "-"))
15102                     {
15103                         // "-" cannot be used for const access
15104                         JSON_THROW(detail::out_of_range::create(402, detail::concat("array index '-'
      (", std::to_string(ptr->m_data.m_value.array->size()), ") is out of range"), ptr));
15105                     }
15106
15107                     // use unchecked array access
15108                     ptr = &ptr->operator[](array_index<BasicJsonType>(reference_token));
15109                     break;
15110                 }
15111
15112                 case detail::value_t::null:
15113                 case detail::value_t::string:
15114                 case detail::value_t::boolean:
15115                 case detail::value_t::number_integer:
15116                 case detail::value_t::number_unsigned:
15117                 case detail::value_t::number_float:
15118                 case detail::value_t::binary:
15119                 case detail::value_t::discarded:
15120                 default:
15121                     JSON_THROW(detail::out_of_range::create(404, detail::concat("unresolved reference
      token '", reference_token, "'"), ptr));
15122             }
15123         }
15124
15125         return *ptr;
15126     }
15127
15134     template<typename BasicJsonType>
15135     const BasicJsonType& get_checked(const BasicJsonType* ptr) const
15136     {
15137         for (const auto& reference_token : reference_tokens)
15138         {
15139             switch (ptr->type())
15140             {
15141                 case detail::value_t::object:
15142                 {
15143                     // note: at performs range check
15144                     ptr = &ptr->at(reference_token);
15145                     break;
15146                 }
15147
15148                 case detail::value_t::array:
15149                 {
15150                     if (JSON_HEDLEY_UNLIKELY(reference_token == "-"))
15151                     {
15152                         // "-" always fails the range check
15153                         JSON_THROW(detail::out_of_range::create(402, detail::concat(
15154                                         "array index '-' (",
      std::to_string(ptr->m_data.m_value.array->size()),
15155                                         ") is out of range"), ptr));
15156                     }
15157
15158                     // note: at performs range check
15159                     ptr = &ptr->at(array_index<BasicJsonType>(reference_token));
15160                     break;
15161                 }
15162
15163                 case detail::value_t::null:
15164                 case detail::value_t::string:
15165                 case detail::value_t::boolean:
15166                 case detail::value_t::number_integer:
15167                 case detail::value_t::number_unsigned:
15168                 case detail::value_t::number_float:
15169                 case detail::value_t::binary:
15170                 case detail::value_t::discarded:
15171                 default:
15172                     JSON_THROW(detail::out_of_range::create(404, detail::concat("unresolved reference
      token '", reference_token, "'"), ptr));
15173             }
15174         }
15175
15176         return *ptr;
15177     }
15178
15183     template<typename BasicJsonType>
15184     bool contains(const BasicJsonType* ptr) const
15185     {
15186         for (const auto& reference_token : reference_tokens)
15187         {
15188             switch (ptr->type())
15189             {
15190                 case detail::value_t::object:
15191                 {
15192                     if (!ptr->contains(reference_token))
```

```
15193                     {
15194                         // we did not find the key in the object
15195                         return false;
15196                     }
15197 
15198                     ptr = &ptr->operator[](reference_token);
15199                     break;
15200                 }
15201 
15202                 case detail::value_t::array:
15203                 {
15204                     if (JSON_HEDLEY_UNLIKELY(reference_token == "-"))
15205                     {
15206                         // "-" always fails the range check
15207                         return false;
15208                     }
15209                     if (JSON_HEDLEY_UNLIKELY(reference_token.size() == 1 && !("0" <= reference_token
     && reference_token <= "9")))
15210                     {
15211                         // invalid char
15212                         return false;
15213                     }
15214                     if (JSON_HEDLEY_UNLIKELY(reference_token.size() > 1))
15215                     {
15216                         if (JSON_HEDLEY_UNLIKELY(!('1' <= reference_token[0] && reference_token[0] <=
     '9')))
15217                         {
15218                             // the first char should be between '1' and '9'
15219                             return false;
15220                         }
15221                         for (std::size_t i = 1; i < reference_token.size(); i++)
15222                         {
15223                             if (JSON_HEDLEY_UNLIKELY(!('0' <= reference_token[i] && reference_token[i]
     <= '9')))
15224                             {
15225                                 // other char should be between '0' and '9'
15226                                 return false;
15227                             }
15228                         }
15229                     }
15230 
15231                     const auto idx = array_index<BasicJsonType>(reference_token);
15232                     if (idx >= ptr->size())
15233                     {
15234                         // index out of range
15235                         return false;
15236                     }
15237 
15238                     ptr = &ptr->operator[](idx);
15239                     break;
15240                 }
15241 
15242                 case detail::value_t::null:
15243                 case detail::value_t::string:
15244                 case detail::value_t::boolean:
15245                 case detail::value_t::number_integer:
15246                 case detail::value_t::number_unsigned:
15247                 case detail::value_t::number_float:
15248                 case detail::value_t::binary:
15249                 case detail::value_t::discarded:
15250                 default:
15251                 {
15252                     // we do not expect primitive values if there is still a
15253                     // reference token to process
15254                     return false;
15255                 }
15256             }
15257         }
15258 
15259         // no reference token left means we found a primitive value
15260         return true;
15261     }
15262 
15272     static std::vector<string_t> split(const string_t& reference_string)
15273     {
15274         std::vector<string_t> result;
15275 
15276         // special case: empty reference string -> no reference tokens
15277         if (reference_string.empty())
15278         {
15279             return result;
15280         }
15281 
15282         // check if a nonempty reference string begins with slash
15283         if (JSON_HEDLEY_UNLIKELY(reference_string[0] != '/'))
15284         {
15285             JSON_THROW(detail::parse_error::create(107, 1, detail::concat("JSON pointer must be empty
```

```
      or begin with '/' - was: '", reference_string, "'"), nullptr));
15286         }
15287
15288         // extract the reference tokens:
15289         // - slash: position of the last read slash (or end of string)
15290         // - start: position after the previous slash
15291         for (
15292             // search for the first slash after the first character
15293             std::size_t slash = reference_string.find_first_of('/', 1),
15294             // set the beginning of the first reference token
15295             start = 1;
15296             // we can stop if start == 0 (if slash == string_t::npos)
15297             start != 0;
15298             // set the beginning of the next reference token
15299             // (will eventually be 0 if slash == string_t::npos)
15300             start = (slash == string_t::npos) ? 0 : slash + 1,
15301             // find next slash
15302             slash = reference_string.find_first_of('/', start))
15303         {
15304             // use the text between the beginning of the reference token
15305             // (start) and the last slash (slash).
15306             auto reference_token = reference_string.substr(start, slash - start);
15307
15308             // check reference tokens are properly escaped
15309             for (std::size_t pos = reference_token.find_first_of('~');
15310                     pos != string_t::npos;
15311                     pos = reference_token.find_first_of('~', pos + 1))
15312             {
15313                 JSON_ASSERT(reference_token[pos] == '~');
15314
15315                 // ~ must be followed by 0 or 1
15316                 if (JSON_HEDLEY_UNLIKELY(pos == reference_token.size() - 1 ||
15317                                          (reference_token[pos + 1] != '0' &&
15318                                           reference_token[pos + 1] != '1')))
15319                 {
15320                     JSON_THROW(detail::parse_error::create(108, 0, "escape character '~' must be
      followed with '0' or '1'", nullptr));
15321                 }
15322             }
15323
15324             // finally, store the reference token
15325             detail::unescape(reference_token);
15326             result.push_back(reference_token);
15327         }
15328
15329         return result;
15330     }
15331
15332   private:
15340     template<typename BasicJsonType>
15341     static void flatten(const string_t& reference_string,
15342                         const BasicJsonType& value,
15343                         BasicJsonType& result)
15344     {
15345         switch (value.type())
15346         {
15347             case detail::value_t::array:
15348             {
15349                 if (value.m_data.m_value.array->empty())
15350                 {
15351                     // flatten empty array as null
15352                     result[reference_string] = nullptr;
15353                 }
15354                 else
15355                 {
15356                     // iterate array and use index as a reference string
15357                     for (std::size_t i = 0; i < value.m_data.m_value.array->size(); ++i)
15358                     {
15359                         flatten(detail::concat<string_t>(reference_string, '/', std::to_string(i)),
15360                                 value.m_data.m_value.array->operator[](i), result);
15361                     }
15362                 }
15363                 break;
15364             }
15365
15366             case detail::value_t::object:
15367             {
15368                 if (value.m_data.m_value.object->empty())
15369                 {
15370                     // flatten empty object as null
15371                     result[reference_string] = nullptr;
15372                 }
15373                 else
15374                 {
15375                     // iterate object and use keys as reference string
15376                     for (const auto& element : *value.m_data.m_value.object)
15377                     {
```

```
15378                         flatten(detail::concat<string_t>(reference_string, '/',
      detail::escape(element.first)), element.second, result);
15379                     }
15380                 }
15381                 break;
15382             }
15383
15384             case detail::value_t::null:
15385             case detail::value_t::string:
15386             case detail::value_t::boolean:
15387             case detail::value_t::number_integer:
15388             case detail::value_t::number_unsigned:
15389             case detail::value_t::number_float:
15390             case detail::value_t::binary:
15391             case detail::value_t::discarded:
15392             default:
15393             {
15394                 // add a primitive value with its reference string
15395                 result[reference_string] = value;
15396                 break;
15397             }
15398         }
15399     }
15400
15411     template<typename BasicJsonType>
15412     static BasicJsonType
15413     unflatten(const BasicJsonType& value)
15414     {
15415         if (JSON_HEDLEY_UNLIKELY(!value.is_object()))
15416         {
15417             JSON_THROW(detail::type_error::create(314, "only objects can be unflattened", &value));
15418         }
15419
15420         BasicJsonType result;
15421
15422         // iterate the JSON object values
15423         for (const auto& element : *value.m_data.m_value.object)
15424         {
15425             if (JSON_HEDLEY_UNLIKELY(!element.second.is_primitive()))
15426             {
15427                 JSON_THROW(detail::type_error::create(315, "values in object must be primitive",
      &element.second));
15428             }
15429
15430             // Assign the value to the reference pointed to by JSON pointer. Note
15431             // that if the JSON pointer is "" (i.e., points to the whole value),
15432             // function get_and_create returns a reference to the result itself.
15433             // An assignment will then create a primitive value.
15434             json_pointer(element.first).get_and_create(result) = element.second;
15435         }
15436
15437         return result;
15438     }
15439
15440     // can't use the conversion operator because of ambiguity
15441     json_pointer<string_t> convert() const&
15442     {
15443         json_pointer<string_t> result;
15444         result.reference_tokens = reference_tokens;
15445         return result;
15446     }
15447
15448     json_pointer<string_t> convert()&&
15449     {
15450         json_pointer<string_t> result;
15451         result.reference_tokens = std::move(reference_tokens);
15452         return result;
15453     }
15454
15455   public:
15456 #if JSON_HAS_THREE_WAY_COMPARISON
15459     template<typename RefStringTypeRhs>
15460     bool operator==(const json_pointer<RefStringTypeRhs>& rhs) const noexcept
15461     {
15462         return reference_tokens == rhs.reference_tokens;
15463     }
15464
15467     JSON_HEDLEY_DEPRECATED_FOR(3.11.2, operator==(json_pointer))
15468     bool operator==(const string_t& rhs) const
15469     {
15470         return *this == json_pointer(rhs);
15471     }
15472
15474     template<typename RefStringTypeRhs>
15475     std::strong_ordering operator<=>(const json_pointer<RefStringTypeRhs>& rhs) const noexcept //
      *NOPAD*
15476     {
```

```
15477         return  reference_tokens <=> rhs.reference_tokens; // *NOPAD*
15478     }
15479 #else
15482     template<typename RefStringTypeLhs, typename RefStringTypeRhs>
15483     // NOLINTNEXTLINE(readability-redundant-declaration)
15484     friend bool operator==(const json_pointer<RefStringTypeLhs>& lhs,
15485                            const json_pointer<RefStringTypeRhs>& rhs) noexcept;
15486
15489     template<typename RefStringTypeLhs, typename StringType>
15490     // NOLINTNEXTLINE(readability-redundant-declaration)
15491     friend bool operator==(const json_pointer<RefStringTypeLhs>& lhs,
15492                            const StringType& rhs);
15493
15496     template<typename RefStringTypeRhs, typename StringType>
15497     // NOLINTNEXTLINE(readability-redundant-declaration)
15498     friend bool operator==(const StringType& lhs,
15499                            const json_pointer<RefStringTypeRhs>& rhs);
15500
15503     template<typename RefStringTypeLhs, typename RefStringTypeRhs>
15504     // NOLINTNEXTLINE(readability-redundant-declaration)
15505     friend bool operator!=(const json_pointer<RefStringTypeLhs>& lhs,
15506                            const json_pointer<RefStringTypeRhs>& rhs) noexcept;
15507
15510     template<typename RefStringTypeLhs, typename StringType>
15511     // NOLINTNEXTLINE(readability-redundant-declaration)
15512     friend bool operator!=(const json_pointer<RefStringTypeLhs>& lhs,
15513                            const StringType& rhs);
15514
15517     template<typename RefStringTypeRhs, typename StringType>
15518     // NOLINTNEXTLINE(readability-redundant-declaration)
15519     friend bool operator!=(const StringType& lhs,
15520                            const json_pointer<RefStringTypeRhs>& rhs);
15521
15523     template<typename RefStringTypeLhs, typename RefStringTypeRhs>
15524     // NOLINTNEXTLINE(readability-redundant-declaration)
15525     friend bool operator<(const json_pointer<RefStringTypeLhs>& lhs,
15526                           const json_pointer<RefStringTypeRhs>& rhs) noexcept;
15527 #endif
15528
15529   private:
15531     std::vector<string_t> reference_tokens;
15532 };
15533
15534 #if !JSON_HAS_THREE_WAY_COMPARISON
15535 // functions cannot be defined inside the class due to ODR violations
15536 template<typename RefStringTypeLhs, typename RefStringTypeRhs>
15537 inline bool operator==(const json_pointer<RefStringTypeLhs>& lhs,
15538                        const json_pointer<RefStringTypeRhs>& rhs) noexcept
15539 {
15540     return lhs.reference_tokens == rhs.reference_tokens;
15541 }
15542
15543 template<typename RefStringTypeLhs,
15544          typename StringType = typename json_pointer<RefStringTypeLhs>::string_t>
15545 JSON_HEDLEY_DEPRECATED_FOR(3.11.2, operator==(json_pointer, json_pointer))
15546 inline bool operator==(const json_pointer<RefStringTypeLhs>& lhs,
15547                        const StringType& rhs)
15548 {
15549     return lhs == json_pointer<RefStringTypeLhs>(rhs);
15550 }
15551
15552 template<typename RefStringTypeRhs,
15553          typename StringType = typename json_pointer<RefStringTypeRhs>::string_t>
15554 JSON_HEDLEY_DEPRECATED_FOR(3.11.2, operator==(json_pointer, json_pointer))
15555 inline bool operator==(const StringType& lhs,
15556                        const json_pointer<RefStringTypeRhs>& rhs)
15557 {
15558     return json_pointer<RefStringTypeRhs>(lhs) == rhs;
15559 }
15560
15561 template<typename RefStringTypeLhs, typename RefStringTypeRhs>
15562 inline bool operator!=(const json_pointer<RefStringTypeLhs>& lhs,
15563                        const json_pointer<RefStringTypeRhs>& rhs) noexcept
15564 {
15565     return !(lhs == rhs);
15566 }
15567
15568 template<typename RefStringTypeLhs,
15569          typename StringType = typename json_pointer<RefStringTypeLhs>::string_t>
15570 JSON_HEDLEY_DEPRECATED_FOR(3.11.2, operator!=(json_pointer, json_pointer))
15571 inline bool operator!=(const json_pointer<RefStringTypeLhs>& lhs,
15572                        const StringType& rhs)
15573 {
15574     return !(lhs == rhs);
15575 }
15576
15577 template<typename RefStringTypeRhs,
```

```
15578          typename StringType = typename json_pointer<RefStringTypeRhs>::string_t>
15579 JSON_HEDLEY_DEPRECATED_FOR(3.11.2, operator!=(json_pointer, json_pointer))
15580 inline bool operator!=(const StringType& lhs,
15581                        const json_pointer<RefStringTypeRhs>& rhs)
15582 {
15583     return !(lhs == rhs);
15584 }
15585
15586 template<typename RefStringTypeLhs, typename RefStringTypeRhs>
15587 inline bool operator<(const json_pointer<RefStringTypeLhs>& lhs,
15588                       const json_pointer<RefStringTypeRhs>& rhs) noexcept
15589 {
15590     return lhs.reference_tokens < rhs.reference_tokens;
15591 }
15592 #endif
15593
15594 NLOHMANN_JSON_NAMESPACE_END
15595
15596 // #include <nlohmann/detail/json_ref.hpp>
15597 //      __ _____ _____ _____
15598 //   __|  |   __|     |   | |  JSON for Modern C++
15599 // |  |  |__   |  |  | | | |  version 3.12.0
15600 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
15601 //
15602 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
15603 // SPDX-License-Identifier: MIT
15604
15605
15606
15607 #include <initializer_list>
15608 #include <utility>
15609
15610 // #include <nlohmann/detail/abi_macros.hpp>
15611
15612 // #include <nlohmann/detail/meta/type_traits.hpp>
15613
15614
15615 NLOHMANN_JSON_NAMESPACE_BEGIN
15616 namespace detail
15617 {
15618
15619 template<typename BasicJsonType>
15620 class json_ref
15621 {
15622   public:
15623     using value_type = BasicJsonType;
15624
15625     json_ref(value_type&& value)
15626         : owned_value(std::move(value))
15627     {}
15628
15629     json_ref(const value_type& value)
15630         : value_ref(&value)
15631     {}
15632
15633     json_ref(std::initializer_list<json_ref> init)
15634         : owned_value(init)
15635     {}
15636
15637     template <
15638         class... Args,
15639         enable_if_t<std::is_constructible<value_type, Args...>::value, int> = 0 >
15640     json_ref(Args && ... args)
15641         : owned_value(std::forward<Args>(args)...)
15642     {}
15643
15644     // class should be movable only
15645     json_ref(json_ref&&) noexcept = default;
15646     json_ref(const json_ref&) = delete;
15647     json_ref& operator=(const json_ref&) = delete;
15648     json_ref& operator=(json_ref&&) = delete;
15649     ~json_ref() = default;
15650
15651     value_type moved_or_copied() const
15652     {
15653         if (value_ref == nullptr)
15654         {
15655             return std::move(owned_value);
15656         }
15657         return *value_ref;
15658     }
15659
15660     value_type const& operator*() const
15661     {
15662         return value_ref ? *value_ref : owned_value;
15663     }
15664
```

```
15665     value_type const* operator->() const
15666     {
15667         return &** this;
15668     }
15669
15670   private:
15671     mutable value_type owned_value = nullptr;
15672     value_type const* value_ref = nullptr;
15673 };
15674
15675 }  // namespace detail
15676 NLOHMANN_JSON_NAMESPACE_END
15677
15678 // #include <nlohmann/detail/macro_scope.hpp>
15679
15680 // #include <nlohmann/detail/string_concat.hpp>
15681
15682 // #include <nlohmann/detail/string_escape.hpp>
15683
15684 // #include <nlohmann/detail/string_utils.hpp>
15685
15686 // #include <nlohmann/detail/meta/cpp_future.hpp>
15687
15688 // #include <nlohmann/detail/meta/type_traits.hpp>
15689
15690 // #include <nlohmann/detail/output/binary_writer.hpp>
15691 //     __ _____ _____ _____
15692 //  __|  |   __|     |   | |  JSON for Modern C++
15693 // |  |  |__   |  |  | | | |  version 3.12.0
15694 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
15695 //
15696 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
15697 // SPDX-License-Identifier: MIT
15698
15699
15700
15701 #include <algorithm> // reverse
15702 #include <array> // array
15703 #include <map> // map
15704 #include <cmath> // isnan, isinf
15705 #include <cstdint> // uint8_t, uint16_t, uint32_t, uint64_t
15706 #include <cstring> // memcpy
15707 #include <limits> // numeric_limits
15708 #include <string> // string
15709 #include <utility> // move
15710 #include <vector> // vector
15711
15712 // #include <nlohmann/detail/input/binary_reader.hpp>
15713
15714 // #include <nlohmann/detail/macro_scope.hpp>
15715
15716 // #include <nlohmann/detail/output/output_adapters.hpp>
15717 //     __ _____ _____ _____
15718 //  __|  |   __|     |   | |  JSON for Modern C++
15719 // |  |  |__   |  |  | | | |  version 3.12.0
15720 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
15721 //
15722 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
15723 // SPDX-License-Identifier: MIT
15724
15725
15726
15727 #include <algorithm> // copy
15728 #include <cstddef> // size_t
15729 #include <iterator> // back_inserter
15730 #include <memory> // shared_ptr, make_shared
15731 #include <string> // basic_string
15732 #include <vector> // vector
15733
15734 #ifndef JSON_NO_IO
15735     #include <ios>      // streamsize
15736     #include <ostream>  // basic_ostream
15737 #endif  // JSON_NO_IO
15738
15739 // #include <nlohmann/detail/macro_scope.hpp>
15740
15741
15742 NLOHMANN_JSON_NAMESPACE_BEGIN
15743 namespace detail
15744 {
15745
15747 template<typename CharType> struct output_adapter_protocol
15748 {
15749     virtual void write_character(CharType c) = 0;
15750     virtual void write_characters(const CharType* s, std::size_t length) = 0;
15751     virtual ~output_adapter_protocol() = default;
15752
```

```
15753     output_adapter_protocol() = default;
15754     output_adapter_protocol(const output_adapter_protocol&) = default;
15755     output_adapter_protocol(output_adapter_protocol&&) noexcept = default;
15756     output_adapter_protocol& operator=(const output_adapter_protocol&) = default;
15757     output_adapter_protocol& operator=(output_adapter_protocol&&) noexcept = default;
15758 };
15759
15761 template<typename CharType>
15762 using output_adapter_t = std::shared_ptr<output_adapter_protocol<CharType>>;
15763
15765 template<typename CharType, typename AllocatorType = std::allocator<CharType>>
15766 class output_vector_adapter : public output_adapter_protocol<CharType>
15767 {
15768   public:
15769     explicit output_vector_adapter(std::vector<CharType, AllocatorType>& vec) noexcept
15770         : v(vec)
15771     {}
15772
15773     void write_character(CharType c) override
15774     {
15775         v.push_back(c);
15776     }
15777
15778     JSON_HEDLEY_NON_NULL(2)
15779     void write_characters(const CharType* s, std::size_t length) override
15780     {
15781         v.insert(v.end(), s, s + length);
15782     }
15783
15784   private:
15785     std::vector<CharType, AllocatorType>& v;
15786 };
15787
15788 #ifndef JSON_NO_IO
15790 template<typename CharType>
15791 class output_stream_adapter : public output_adapter_protocol<CharType>
15792 {
15793   public:
15794     explicit output_stream_adapter(std::basic_ostream<CharType>& s) noexcept
15795         : stream(s)
15796     {}
15797
15798     void write_character(CharType c) override
15799     {
15800         stream.put(c);
15801     }
15802
15803     JSON_HEDLEY_NON_NULL(2)
15804     void write_characters(const CharType* s, std::size_t length) override
15805     {
15806         stream.write(s, static_cast<std::streamsize>(length));
15807     }
15808
15809   private:
15810     std::basic_ostream<CharType>& stream;
15811 };
15812 #endif  // JSON_NO_IO
15813
15815 template<typename CharType, typename StringType = std::basic_string<CharType>>
15816 class output_string_adapter : public output_adapter_protocol<CharType>
15817 {
15818   public:
15819     explicit output_string_adapter(StringType& s) noexcept
15820         : str(s)
15821     {}
15822
15823     void write_character(CharType c) override
15824     {
15825         str.push_back(c);
15826     }
15827
15828     JSON_HEDLEY_NON_NULL(2)
15829     void write_characters(const CharType* s, std::size_t length) override
15830     {
15831         str.append(s, length);
15832     }
15833
15834   private:
15835     StringType& str;
15836 };
15837
15838 template<typename CharType, typename StringType = std::basic_string<CharType>>
15839 class output_adapter
15840 {
15841   public:
15842     template<typename AllocatorType = std::allocator<CharType>>
15843     output_adapter(std::vector<CharType, AllocatorType>& vec)
```

```
15844         : oa(std::make_shared<output_vector_adapter<CharType, AllocatorType>>(vec)) {}
15845
15846 #ifndef JSON_NO_IO
15847     output_adapter(std::basic_ostream<CharType>& s)
15848         : oa(std::make_shared<output_stream_adapter<CharType>>(s)) {}
15849 #endif  // JSON_NO_IO
15850
15851     output_adapter(StringType& s)
15852         : oa(std::make_shared<output_string_adapter<CharType, StringType>>(s)) {}
15853
15854     operator output_adapter_t<CharType>()
15855     {
15856         return oa;
15857     }
15858
15859   private:
15860     output_adapter_t<CharType> oa = nullptr;
15861 };
15862
15863 }  // namespace detail
15864 NLOHMANN_JSON_NAMESPACE_END
15865
15866 // #include <nlohmann/detail/string_concat.hpp>
15867
15868
15869 NLOHMANN_JSON_NAMESPACE_BEGIN
15870 namespace detail
15871 {
15872
15874 enum class bjdata_version_t
15875 {
15876     draft2,
15877     draft3,
15878 };
15879
15881 // binary writer //
15883
15887 template<typename BasicJsonType, typename CharType>
15888 class binary_writer
15889 {
15890     using string_t = typename BasicJsonType::string_t;
15891     using binary_t = typename BasicJsonType::binary_t;
15892     using number_float_t = typename BasicJsonType::number_float_t;
15893
15894   public:
15900     explicit binary_writer(output_adapter_t<CharType> adapter) : oa(std::move(adapter))
15901     {
15902         JSON_ASSERT(oa);
15903     }
15904
15909     void write_bson(const BasicJsonType& j)
15910     {
15911         switch (j.type())
15912         {
15913             case value_t::object:
15914             {
15915                 write_bson_object(*j.m_data.m_value.object);
15916                 break;
15917             }
15918
15919             case value_t::null:
15920             case value_t::array:
15921             case value_t::string:
15922             case value_t::boolean:
15923             case value_t::number_integer:
15924             case value_t::number_unsigned:
15925             case value_t::number_float:
15926             case value_t::binary:
15927             case value_t::discarded:
15928             default:
15929             {
15930                 JSON_THROW(type_error::create(317, concat("to serialize to BSON, top-level type must
      be object, but is ", j.type_name()), &j));
15931             }
15932         }
15933     }
15934
15938     void write_cbor(const BasicJsonType& j)
15939     {
15940         switch (j.type())
15941         {
15942             case value_t::null:
15943             {
15944                 oa->write_character(to_char_type(0xF6));
15945                 break;
15946             }
15947
```

```
15948             case value_t::boolean:
15949             {
15950                 oa->write_character(j.m_data.m_value.boolean
15951                                     ? to_char_type(0xF5)
15952                                     : to_char_type(0xF4));
15953                 break;
15954             }
15955
15956             case value_t::number_integer:
15957             {
15958                 if (j.m_data.m_value.number_integer >= 0)
15959                 {
15960                     // CBOR does not differentiate between positive signed
15961                     // integers and unsigned integers. Therefore, we used the
15962                     // code from the value_t::number_unsigned case here.
15963                     if (j.m_data.m_value.number_integer <= 0x17)
15964                     {
15965                         write_number(static_cast<std::uint8_t>(j.m_data.m_value.number_integer));
15966                     }
15967                     else if (j.m_data.m_value.number_integer <=
    (std::numeric_limits<std::uint8_t>::max)())
15968                     {
15969                         oa->write_character(to_char_type(0x18));
15970                         write_number(static_cast<std::uint8_t>(j.m_data.m_value.number_integer));
15971                     }
15972                     else if (j.m_data.m_value.number_integer <=
    (std::numeric_limits<std::uint16_t>::max)())
15973                     {
15974                         oa->write_character(to_char_type(0x19));
15975                         write_number(static_cast<std::uint16_t>(j.m_data.m_value.number_integer));
15976                     }
15977                     else if (j.m_data.m_value.number_integer <=
    (std::numeric_limits<std::uint32_t>::max)())
15978                     {
15979                         oa->write_character(to_char_type(0x1A));
15980                         write_number(static_cast<std::uint32_t>(j.m_data.m_value.number_integer));
15981                     }
15982                     else
15983                     {
15984                         oa->write_character(to_char_type(0x1B));
15985                         write_number(static_cast<std::uint64_t>(j.m_data.m_value.number_integer));
15986                     }
15987                 }
15988                 else
15989                 {
15990                     // The conversions below encode the sign in the first
15991                     // byte, and the value is converted to a positive number.
15992                     const auto positive_number = -1 - j.m_data.m_value.number_integer;
15993                     if (j.m_data.m_value.number_integer >= -24)
15994                     {
15995                         write_number(static_cast<std::uint8_t>(0x20 + positive_number));
15996                     }
15997                     else if (positive_number <= (std::numeric_limits<std::uint8_t>::max)())
15998                     {
15999                         oa->write_character(to_char_type(0x38));
16000                         write_number(static_cast<std::uint8_t>(positive_number));
16001                     }
16002                     else if (positive_number <= (std::numeric_limits<std::uint16_t>::max)())
16003                     {
16004                         oa->write_character(to_char_type(0x39));
16005                         write_number(static_cast<std::uint16_t>(positive_number));
16006                     }
16007                     else if (positive_number <= (std::numeric_limits<std::uint32_t>::max)())
16008                     {
16009                         oa->write_character(to_char_type(0x3A));
16010                         write_number(static_cast<std::uint32_t>(positive_number));
16011                     }
16012                     else
16013                     {
16014                         oa->write_character(to_char_type(0x3B));
16015                         write_number(static_cast<std::uint64_t>(positive_number));
16016                     }
16017                 }
16018                 break;
16019             }
16020
16021             case value_t::number_unsigned:
16022             {
16023                 if (j.m_data.m_value.number_unsigned <= 0x17)
16024                 {
16025                     write_number(static_cast<std::uint8_t>(j.m_data.m_value.number_unsigned));
16026                 }
16027                 else if (j.m_data.m_value.number_unsigned <=
    (std::numeric_limits<std::uint8_t>::max)())
16028                 {
16029                     oa->write_character(to_char_type(0x18));
16030                     write_number(static_cast<std::uint8_t>(j.m_data.m_value.number_unsigned));
```

```
16031                 }
16032                 else if (j.m_data.m_value.number_unsigned <=
    (std::numeric_limits<std::uint16_t>::max)())
16033                 {
16034                     oa->write_character(to_char_type(0x19));
16035                     write_number(static_cast<std::uint16_t>(j.m_data.m_value.number_unsigned));
16036                 }
16037                 else if (j.m_data.m_value.number_unsigned <=
    (std::numeric_limits<std::uint32_t>::max)())
16038                 {
16039                     oa->write_character(to_char_type(0x1A));
16040                     write_number(static_cast<std::uint32_t>(j.m_data.m_value.number_unsigned));
16041                 }
16042                 else
16043                 {
16044                     oa->write_character(to_char_type(0x1B));
16045                     write_number(static_cast<std::uint64_t>(j.m_data.m_value.number_unsigned));
16046                 }
16047                 break;
16048             }
16049
16050             case value_t::number_float:
16051             {
16052                 if (std::isnan(j.m_data.m_value.number_float))
16053                 {
16054                     // NaN is 0xf97e00 in CBOR
16055                     oa->write_character(to_char_type(0xF9));
16056                     oa->write_character(to_char_type(0x7E));
16057                     oa->write_character(to_char_type(0x00));
16058                 }
16059                 else if (std::isinf(j.m_data.m_value.number_float))
16060                 {
16061                     // Infinity is 0xf97c00, -Infinity is 0xf9fc00
16062                     oa->write_character(to_char_type(0xf9));
16063                     oa->write_character(j.m_data.m_value.number_float > 0 ? to_char_type(0x7C) :
    to_char_type(0xFC));
16064                     oa->write_character(to_char_type(0x00));
16065                 }
16066                 else
16067                 {
16068                     write_compact_float(j.m_data.m_value.number_float, detail::input_format_t::cbor);
16069                 }
16070                 break;
16071             }
16072
16073             case value_t::string:
16074             {
16075                 // step 1: write control byte and the string length
16076                 const auto N = j.m_data.m_value.string->size();
16077                 if (N <= 0x17)
16078                 {
16079                     write_number(static_cast<std::uint8_t>(0x60 + N));
16080                 }
16081                 else if (N <= (std::numeric_limits<std::uint8_t>::max)())
16082                 {
16083                     oa->write_character(to_char_type(0x78));
16084                     write_number(static_cast<std::uint8_t>(N));
16085                 }
16086                 else if (N <= (std::numeric_limits<std::uint16_t>::max)())
16087                 {
16088                     oa->write_character(to_char_type(0x79));
16089                     write_number(static_cast<std::uint16_t>(N));
16090                 }
16091                 else if (N <= (std::numeric_limits<std::uint32_t>::max)())
16092                 {
16093                     oa->write_character(to_char_type(0x7A));
16094                     write_number(static_cast<std::uint32_t>(N));
16095                 }
16096                 // LCOV_EXCL_START
16097                 else if (N <= (std::numeric_limits<std::uint64_t>::max)())
16098                 {
16099                     oa->write_character(to_char_type(0x7B));
16100                     write_number(static_cast<std::uint64_t>(N));
16101                 }
16102                 // LCOV_EXCL_STOP
16103
16104                 // step 2: write the string
16105                 oa->write_characters(
16106                     reinterpret_cast<const CharType*>(j.m_data.m_value.string->c_str()),
16107                     j.m_data.m_value.string->size());
16108                 break;
16109             }
16110
16111             case value_t::array:
16112             {
16113                 // step 1: write control byte and the array size
16114                 const auto N = j.m_data.m_value.array->size();
```

```
16115                 if (N <= 0x17)
16116                 {
16117                     write_number(static_cast<std::uint8_t>(0x80 + N));
16118                 }
16119                 else if (N <= (std::numeric_limits<std::uint8_t>::max)())
16120                 {
16121                     oa->write_character(to_char_type(0x98));
16122                     write_number(static_cast<std::uint8_t>(N));
16123                 }
16124                 else if (N <= (std::numeric_limits<std::uint16_t>::max)())
16125                 {
16126                     oa->write_character(to_char_type(0x99));
16127                     write_number(static_cast<std::uint16_t>(N));
16128                 }
16129                 else if (N <= (std::numeric_limits<std::uint32_t>::max)())
16130                 {
16131                     oa->write_character(to_char_type(0x9A));
16132                     write_number(static_cast<std::uint32_t>(N));
16133                 }
16134                 // LCOV_EXCL_START
16135                 else if (N <= (std::numeric_limits<std::uint64_t>::max)())
16136                 {
16137                     oa->write_character(to_char_type(0x9B));
16138                     write_number(static_cast<std::uint64_t>(N));
16139                 }
16140                 // LCOV_EXCL_STOP
16141
16142                 // step 2: write each element
16143                 for (const auto& el : *j.m_data.m_value.array)
16144                 {
16145                     write_cbor(el);
16146                 }
16147                 break;
16148             }
16149
16150             case value_t::binary:
16151             {
16152                 if (j.m_data.m_value.binary->has_subtype())
16153                 {
16154                     if (j.m_data.m_value.binary->subtype() <=
    (std::numeric_limits<std::uint8_t>::max)())
16155                     {
16156                         write_number(static_cast<std::uint8_t>(0xd8));
16157                         write_number(static_cast<std::uint8_t>(j.m_data.m_value.binary->subtype()));
16158                     }
16159                     else if (j.m_data.m_value.binary->subtype() <=
    (std::numeric_limits<std::uint16_t>::max)())
16160                     {
16161                         write_number(static_cast<std::uint8_t>(0xd9));
16162                         write_number(static_cast<std::uint16_t>(j.m_data.m_value.binary->subtype()));
16163                     }
16164                     else if (j.m_data.m_value.binary->subtype() <=
    (std::numeric_limits<std::uint32_t>::max)())
16165                     {
16166                         write_number(static_cast<std::uint8_t>(0xda));
16167                         write_number(static_cast<std::uint32_t>(j.m_data.m_value.binary->subtype()));
16168                     }
16169                     else if (j.m_data.m_value.binary->subtype() <=
    (std::numeric_limits<std::uint64_t>::max)())
16170                     {
16171                         write_number(static_cast<std::uint8_t>(0xdb));
16172                         write_number(static_cast<std::uint64_t>(j.m_data.m_value.binary->subtype()));
16173                     }
16174                 }
16175
16176                 // step 1: write control byte and the binary array size
16177                 const auto N = j.m_data.m_value.binary->size();
16178                 if (N <= 0x17)
16179                 {
16180                     write_number(static_cast<std::uint8_t>(0x40 + N));
16181                 }
16182                 else if (N <= (std::numeric_limits<std::uint8_t>::max)())
16183                 {
16184                     oa->write_character(to_char_type(0x58));
16185                     write_number(static_cast<std::uint8_t>(N));
16186                 }
16187                 else if (N <= (std::numeric_limits<std::uint16_t>::max)())
16188                 {
16189                     oa->write_character(to_char_type(0x59));
16190                     write_number(static_cast<std::uint16_t>(N));
16191                 }
16192                 else if (N <= (std::numeric_limits<std::uint32_t>::max)())
16193                 {
16194                     oa->write_character(to_char_type(0x5A));
16195                     write_number(static_cast<std::uint32_t>(N));
16196                 }
16197                 // LCOV_EXCL_START
```

```
16198                 else if (N <= (std::numeric_limits<std::uint64_t>::max)())
16199                 {
16200                     oa->write_character(to_char_type(0x5B));
16201                     write_number(static_cast<std::uint64_t>(N));
16202                 }
16203                 // LCOV_EXCL_STOP
16204
16205                 // step 2: write each element
16206                 oa->write_characters(
16207                     reinterpret_cast<const CharType*>(j.m_data.m_value.binary->data()),
16208                     N);
16209
16210                 break;
16211             }
16212
16213             case value_t::object:
16214             {
16215                 // step 1: write control byte and the object size
16216                 const auto N = j.m_data.m_value.object->size();
16217                 if (N <= 0x17)
16218                 {
16219                     write_number(static_cast<std::uint8_t>(0xA0 + N));
16220                 }
16221                 else if (N <= (std::numeric_limits<std::uint8_t>::max)())
16222                 {
16223                     oa->write_character(to_char_type(0xB8));
16224                     write_number(static_cast<std::uint8_t>(N));
16225                 }
16226                 else if (N <= (std::numeric_limits<std::uint16_t>::max)())
16227                 {
16228                     oa->write_character(to_char_type(0xB9));
16229                     write_number(static_cast<std::uint16_t>(N));
16230                 }
16231                 else if (N <= (std::numeric_limits<std::uint32_t>::max)())
16232                 {
16233                     oa->write_character(to_char_type(0xBA));
16234                     write_number(static_cast<std::uint32_t>(N));
16235                 }
16236                 // LCOV_EXCL_START
16237                 else if (N <= (std::numeric_limits<std::uint64_t>::max)())
16238                 {
16239                     oa->write_character(to_char_type(0xBB));
16240                     write_number(static_cast<std::uint64_t>(N));
16241                 }
16242                 // LCOV_EXCL_STOP
16243
16244                 // step 2: write each element
16245                 for (const auto& el : *j.m_data.m_value.object)
16246                 {
16247                     write_cbor(el.first);
16248                     write_cbor(el.second);
16249                 }
16250                 break;
16251             }
16252
16253             case value_t::discarded:
16254             default:
16255                 break;
16256         }
16257     }
16258
16262     void write_msgpack(const BasicJsonType& j)
16263     {
16264         switch (j.type())
16265         {
16266             case value_t::null: // nil
16267             {
16268                 oa->write_character(to_char_type(0xC0));
16269                 break;
16270             }
16271
16272             case value_t::boolean: // true and false
16273             {
16274                 oa->write_character(j.m_data.m_value.boolean
16275                                     ? to_char_type(0xC3)
16276                                     : to_char_type(0xC2));
16277                 break;
16278             }
16279
16280             case value_t::number_integer:
16281             {
16282                 if (j.m_data.m_value.number_integer >= 0)
16283                 {
16284                     // MessagePack does not differentiate between positive
16285                     // signed integers and unsigned integers. Therefore, we used
16286                     // the code from the value_t::number_unsigned case here.
16287                     if (j.m_data.m_value.number_unsigned < 128)
```

```
16288                     {
16289                         // positive fixnum
16290                         write_number(static_cast<std::uint8_t>(j.m_data.m_value.number_integer));
16291                     }
16292                     else if (j.m_data.m_value.number_unsigned <=
    (std::numeric_limits<std::uint8_t>::max)())
16293                     {
16294                         // uint 8
16295                         oa->write_character(to_char_type(0xCC));
16296                         write_number(static_cast<std::uint8_t>(j.m_data.m_value.number_integer));
16297                     }
16298                     else if (j.m_data.m_value.number_unsigned <=
    (std::numeric_limits<std::uint16_t>::max)())
16299                     {
16300                         // uint 16
16301                         oa->write_character(to_char_type(0xCD));
16302                         write_number(static_cast<std::uint16_t>(j.m_data.m_value.number_integer));
16303                     }
16304                     else if (j.m_data.m_value.number_unsigned <=
    (std::numeric_limits<std::uint32_t>::max)())
16305                     {
16306                         // uint 32
16307                         oa->write_character(to_char_type(0xCE));
16308                         write_number(static_cast<std::uint32_t>(j.m_data.m_value.number_integer));
16309                     }
16310                     else if (j.m_data.m_value.number_unsigned <=
    (std::numeric_limits<std::uint64_t>::max)())
16311                     {
16312                         // uint 64
16313                         oa->write_character(to_char_type(0xCF));
16314                         write_number(static_cast<std::uint64_t>(j.m_data.m_value.number_integer));
16315                     }
16316                 }
16317                 else
16318                 {
16319                     if (j.m_data.m_value.number_integer >= -32)
16320                     {
16321                         // negative fixnum
16322                         write_number(static_cast<std::int8_t>(j.m_data.m_value.number_integer));
16323                     }
16324                     else if (j.m_data.m_value.number_integer >=
    (std::numeric_limits<std::int8_t>::min)() &&
16325                              j.m_data.m_value.number_integer <=
    (std::numeric_limits<std::int8_t>::max)())
16326                     {
16327                         // int 8
16328                         oa->write_character(to_char_type(0xD0));
16329                         write_number(static_cast<std::int8_t>(j.m_data.m_value.number_integer));
16330                     }
16331                     else if (j.m_data.m_value.number_integer >=
    (std::numeric_limits<std::int16_t>::min)() &&
16332                              j.m_data.m_value.number_integer <=
    (std::numeric_limits<std::int16_t>::max)())
16333                     {
16334                         // int 16
16335                         oa->write_character(to_char_type(0xD1));
16336                         write_number(static_cast<std::int16_t>(j.m_data.m_value.number_integer));
16337                     }
16338                     else if (j.m_data.m_value.number_integer >=
    (std::numeric_limits<std::int32_t>::min)() &&
16339                              j.m_data.m_value.number_integer <=
    (std::numeric_limits<std::int32_t>::max)())
16340                     {
16341                         // int 32
16342                         oa->write_character(to_char_type(0xD2));
16343                         write_number(static_cast<std::int32_t>(j.m_data.m_value.number_integer));
16344                     }
16345                     else if (j.m_data.m_value.number_integer >=
    (std::numeric_limits<std::int64_t>::min)() &&
16346                              j.m_data.m_value.number_integer <=
    (std::numeric_limits<std::int64_t>::max)())
16347                     {
16348                         // int 64
16349                         oa->write_character(to_char_type(0xD3));
16350                         write_number(static_cast<std::int64_t>(j.m_data.m_value.number_integer));
16351                     }
16352                 }
16353                 break;
16354             }
16355
16356             case value_t::number_unsigned:
16357             {
16358                 if (j.m_data.m_value.number_unsigned < 128)
16359                 {
16360                     // positive fixnum
16361                     write_number(static_cast<std::uint8_t>(j.m_data.m_value.number_integer));
16362                 }
```

```
16363                 else if (j.m_data.m_value.number_unsigned <=
      (std::numeric_limits<std::uint8_t>::max)())
16364                 {
16365                     // uint 8
16366                     oa->write_character(to_char_type(0xCC));
16367                     write_number(static_cast<std::uint8_t>(j.m_data.m_value.number_integer));
16368                 }
16369                 else if (j.m_data.m_value.number_unsigned <=
      (std::numeric_limits<std::uint16_t>::max)())
16370                 {
16371                     // uint 16
16372                     oa->write_character(to_char_type(0xCD));
16373                     write_number(static_cast<std::uint16_t>(j.m_data.m_value.number_integer));
16374                 }
16375                 else if (j.m_data.m_value.number_unsigned <=
      (std::numeric_limits<std::uint32_t>::max)())
16376                 {
16377                     // uint 32
16378                     oa->write_character(to_char_type(0xCE));
16379                     write_number(static_cast<std::uint32_t>(j.m_data.m_value.number_integer));
16380                 }
16381                 else if (j.m_data.m_value.number_unsigned <=
      (std::numeric_limits<std::uint64_t>::max)())
16382                 {
16383                     // uint 64
16384                     oa->write_character(to_char_type(0xCF));
16385                     write_number(static_cast<std::uint64_t>(j.m_data.m_value.number_integer));
16386                 }
16387                 break;
16388             }
16389
16390             case value_t::number_float:
16391             {
16392                 write_compact_float(j.m_data.m_value.number_float, detail::input_format_t::msgpack);
16393                 break;
16394             }
16395
16396             case value_t::string:
16397             {
16398                 // step 1: write control byte and the string length
16399                 const auto N = j.m_data.m_value.string->size();
16400                 if (N <= 31)
16401                 {
16402                     // fixstr
16403                     write_number(static_cast<std::uint8_t>(0xA0 | N));
16404                 }
16405                 else if (N <= (std::numeric_limits<std::uint8_t>::max)())
16406                 {
16407                     // str 8
16408                     oa->write_character(to_char_type(0xD9));
16409                     write_number(static_cast<std::uint8_t>(N));
16410                 }
16411                 else if (N <= (std::numeric_limits<std::uint16_t>::max)())
16412                 {
16413                     // str 16
16414                     oa->write_character(to_char_type(0xDA));
16415                     write_number(static_cast<std::uint16_t>(N));
16416                 }
16417                 else if (N <= (std::numeric_limits<std::uint32_t>::max)())
16418                 {
16419                     // str 32
16420                     oa->write_character(to_char_type(0xDB));
16421                     write_number(static_cast<std::uint32_t>(N));
16422                 }
16423
16424                 // step 2: write the string
16425                 oa->write_characters(
16426                     reinterpret_cast<const CharType*>(j.m_data.m_value.string->c_str()),
16427                     j.m_data.m_value.string->size());
16428                 break;
16429             }
16430
16431             case value_t::array:
16432             {
16433                 // step 1: write control byte and the array size
16434                 const auto N = j.m_data.m_value.array->size();
16435                 if (N <= 15)
16436                 {
16437                     // fixarray
16438                     write_number(static_cast<std::uint8_t>(0x90 | N));
16439                 }
16440                 else if (N <= (std::numeric_limits<std::uint16_t>::max)())
16441                 {
16442                     // array 16
16443                     oa->write_character(to_char_type(0xDC));
16444                     write_number(static_cast<std::uint16_t>(N));
16445                 }
```

```
16446                 else if (N <= (std::numeric_limits<std::uint32_t>::max)())
16447                 {
16448                     // array 32
16449                     oa->write_character(to_char_type(0xDD));
16450                     write_number(static_cast<std::uint32_t>(N));
16451                 }
16452 
16453                 // step 2: write each element
16454                 for (const auto& el : *j.m_data.m_value.array)
16455                 {
16456                     write_msgpack(el);
16457                 }
16458                 break;
16459             }
16460 
16461             case value_t::binary:
16462             {
16463                 // step 0: determine if the binary type has a set subtype to
16464                 // determine whether to use the ext or fixext types
16465                 const bool use_ext = j.m_data.m_value.binary->has_subtype();
16466 
16467                 // step 1: write control byte and the byte string length
16468                 const auto N = j.m_data.m_value.binary->size();
16469                 if (N <= (std::numeric_limits<std::uint8_t>::max)())
16470                 {
16471                     std::uint8_t output_type{};
16472                     bool fixed = true;
16473                     if (use_ext)
16474                     {
16475                         switch (N)
16476                         {
16477                             case 1:
16478                                 output_type = 0xD4; // fixext 1
16479                                 break;
16480                             case 2:
16481                                 output_type = 0xD5; // fixext 2
16482                                 break;
16483                             case 4:
16484                                 output_type = 0xD6; // fixext 4
16485                                 break;
16486                             case 8:
16487                                 output_type = 0xD7; // fixext 8
16488                                 break;
16489                             case 16:
16490                                 output_type = 0xD8; // fixext 16
16491                                 break;
16492                             default:
16493                                 output_type = 0xC7; // ext 8
16494                                 fixed = false;
16495                                 break;
16496                         }
16497 
16498                     }
16499                     else
16500                     {
16501                         output_type = 0xC4; // bin 8
16502                         fixed = false;
16503                     }
16504 
16505                     oa->write_character(to_char_type(output_type));
16506                     if (!fixed)
16507                     {
16508                         write_number(static_cast<std::uint8_t>(N));
16509                     }
16510                 }
16511                 else if (N <= (std::numeric_limits<std::uint16_t>::max)())
16512                 {
16513                     const std::uint8_t output_type = use_ext
16514                                                      ? 0xC8 // ext 16
16515                                                      : 0xC5; // bin 16
16516 
16517                     oa->write_character(to_char_type(output_type));
16518                     write_number(static_cast<std::uint16_t>(N));
16519                 }
16520                 else if (N <= (std::numeric_limits<std::uint32_t>::max)())
16521                 {
16522                     const std::uint8_t output_type = use_ext
16523                                                      ? 0xC9 // ext 32
16524                                                      : 0xC6; // bin 32
16525 
16526                     oa->write_character(to_char_type(output_type));
16527                     write_number(static_cast<std::uint32_t>(N));
16528                 }
16529 
16530                 // step 1.5: if this is an ext type, write the subtype
16531                 if (use_ext)
16532                 {
```

```
16533                     write_number(static_cast<std::int8_t>(j.m_data.m_value.binary->subtype()));
16534                 }
16535
16536                 // step 2: write the byte string
16537                 oa->write_characters(
16538                     reinterpret_cast<const CharType*>(j.m_data.m_value.binary->data()),
16539                     N);
16540
16541                 break;
16542             }
16543
16544             case value_t::object:
16545             {
16546                 // step 1: write control byte and the object size
16547                 const auto N = j.m_data.m_value.object->size();
16548                 if (N <= 15)
16549                 {
16550                     // fixmap
16551                     write_number(static_cast<std::uint8_t>(0x80 | (N & 0xF)));
16552                 }
16553                 else if (N <= (std::numeric_limits<std::uint16_t>::max)())
16554                 {
16555                     // map 16
16556                     oa->write_character(to_char_type(0xDE));
16557                     write_number(static_cast<std::uint16_t>(N));
16558                 }
16559                 else if (N <= (std::numeric_limits<std::uint32_t>::max)())
16560                 {
16561                     // map 32
16562                     oa->write_character(to_char_type(0xDF));
16563                     write_number(static_cast<std::uint32_t>(N));
16564                 }
16565
16566                 // step 2: write each element
16567                 for (const auto& el : *j.m_data.m_value.object)
16568                 {
16569                     write_msgpack(el.first);
16570                     write_msgpack(el.second);
16571                 }
16572                 break;
16573             }
16574
16575             case value_t::discarded:
16576             default:
16577                 break;
16578         }
16579     }
16580
16589     void write_ubjson(const BasicJsonType& j, const bool use_count,
16590                       const bool use_type, const bool add_prefix = true,
16591                       const bool use_bjdata = false, const bjdata_version_t bjdata_version =
    bjdata_version_t::draft2)
16592     {
16593         const bool bjdata_draft3 = use_bjdata && bjdata_version == bjdata_version_t::draft3;
16594
16595         switch (j.type())
16596         {
16597             case value_t::null:
16598             {
16599                 if (add_prefix)
16600                 {
16601                     oa->write_character(to_char_type('Z'));
16602                 }
16603                 break;
16604             }
16605
16606             case value_t::boolean:
16607             {
16608                 if (add_prefix)
16609                 {
16610                     oa->write_character(j.m_data.m_value.boolean
16611                                         ? to_char_type('T')
16612                                         : to_char_type('F'));
16613                 }
16614                 break;
16615             }
16616
16617             case value_t::number_integer:
16618             {
16619                 write_number_with_ubjson_prefix(j.m_data.m_value.number_integer, add_prefix,
    use_bjdata);
16620                 break;
16621             }
16622
16623             case value_t::number_unsigned:
16624             {
16625                 write_number_with_ubjson_prefix(j.m_data.m_value.number_unsigned, add_prefix,
```

```
use_bjdata);
16626                 break;
16627             }
16628
16629             case value_t::number_float:
16630             {
16631                 write_number_with_ubjson_prefix(j.m_data.m_value.number_float, add_prefix,
use_bjdata);
16632                 break;
16633             }
16634
16635             case value_t::string:
16636             {
16637                 if (add_prefix)
16638                 {
16639                     oa->write_character(to_char_type('S'));
16640                 }
16641                 write_number_with_ubjson_prefix(j.m_data.m_value.string->size(), true, use_bjdata);
16642                 oa->write_characters(
16643                     reinterpret_cast<const CharType*>(j.m_data.m_value.string->c_str()),
16644                     j.m_data.m_value.string->size());
16645                 break;
16646             }
16647
16648             case value_t::array:
16649             {
16650                 if (add_prefix)
16651                 {
16652                     oa->write_character(to_char_type('['));
16653                 }
16654
16655                 bool prefix_required = true;
16656                 if (use_type && !j.m_data.m_value.array->empty())
16657                 {
16658                     JSON_ASSERT(use_count);
16659                     const CharType first_prefix = ubjson_prefix(j.front(), use_bjdata);
16660                     const bool same_prefix = std::all_of(j.begin() + 1, j.end(),
16661                                                          [this, first_prefix, use_bjdata](const
BasicJsonType & v)
16662                     {
16663                         return ubjson_prefix(v, use_bjdata) == first_prefix;
16664                     });
16665
16666                     std::vector<CharType> bjdx = {'[', '{', 'S', 'H', 'T', 'F', 'N', 'Z'}; // excluded
markers in bjdata optimized type
16667
16668                     if (same_prefix && !(use_bjdata && std::find(bjdx.begin(), bjdx.end(),
first_prefix) != bjdx.end()))
16669                     {
16670                         prefix_required = false;
16671                         oa->write_character(to_char_type('$'));
16672                         oa->write_character(first_prefix);
16673                     }
16674                 }
16675
16676                 if (use_count)
16677                 {
16678                     oa->write_character(to_char_type('#'));
16679                     write_number_with_ubjson_prefix(j.m_data.m_value.array->size(), true, use_bjdata);
16680                 }
16681
16682                 for (const auto& el : *j.m_data.m_value.array)
16683                 {
16684                     write_ubjson(el, use_count, use_type, prefix_required, use_bjdata,
bjdata_version);
16685                 }
16686
16687                 if (!use_count)
16688                 {
16689                     oa->write_character(to_char_type(']'));
16690                 }
16691
16692                 break;
16693             }
16694
16695             case value_t::binary:
16696             {
16697                 if (add_prefix)
16698                 {
16699                     oa->write_character(to_char_type('['));
16700                 }
16701
16702                 if (use_type && (bjdata_draft3 || !j.m_data.m_value.binary->empty()))
16703                 {
16704                     JSON_ASSERT(use_count);
16705                     oa->write_character(to_char_type('$'));
16706                     oa->write_character(bjdata_draft3 ? 'B' : 'U');
```

```
16707                 }
16708
16709                 if (use_count)
16710                 {
16711                     oa->write_character(to_char_type('#'));
16712                     write_number_with_ubjson_prefix(j.m_data.m_value.binary->size(), true,
      use_bjdata);
16713                 }
16714
16715                 if (use_type)
16716                 {
16717                     oa->write_characters(
16718                         reinterpret_cast<const CharType*>(j.m_data.m_value.binary->data()),
16719                         j.m_data.m_value.binary->size());
16720                 }
16721                 else
16722                 {
16723                     for (size_t i = 0; i < j.m_data.m_value.binary->size(); ++i)
16724                     {
16725                         oa->write_character(to_char_type(bjdata_draft3 ? 'B' : 'U'));
16726                         oa->write_character(j.m_data.m_value.binary->data()[i]);
16727                     }
16728                 }
16729
16730                 if (!use_count)
16731                 {
16732                     oa->write_character(to_char_type(']'));
16733                 }
16734
16735                 break;
16736             }
16737
16738             case value_t::object:
16739             {
16740                 if (use_bjdata && j.m_data.m_value.object->size() == 3 &&
      j.m_data.m_value.object->find("_ArrayType_") != j.m_data.m_value.object->end() &&
      j.m_data.m_value.object->find("_ArraySize_") != j.m_data.m_value.object->end() &&
      j.m_data.m_value.object->find("_ArrayData_") != j.m_data.m_value.object->end())
16741                 {
16742                     if (!write_bjdata_ndarray(*j.m_data.m_value.object, use_count, use_type,
      bjdata_version))  // decode bjdata ndarray in the JData format (https://github.com/NeuroJSON/jdata)
16743                     {
16744                         break;
16745                     }
16746                 }
16747
16748                 if (add_prefix)
16749                 {
16750                     oa->write_character(to_char_type('{'));
16751                 }
16752
16753                 bool prefix_required = true;
16754                 if (use_type && !j.m_data.m_value.object->empty())
16755                 {
16756                     JSON_ASSERT(use_count);
16757                     const CharType first_prefix = ubjson_prefix(j.front(), use_bjdata);
16758                     const bool same_prefix = std::all_of(j.begin(), j.end(),
16759                                                          [this, first_prefix, use_bjdata](const
      BasicJsonType & v)
16760                     {
16761                         return ubjson_prefix(v, use_bjdata) == first_prefix;
16762                     });
16763
16764                     std::vector<CharType> bjdx = {'[', '{', 'S', 'H', 'T', 'F', 'N', 'Z'}; // excluded
      markers in bjdata optimized type
16765
16766                     if (same_prefix && !(use_bjdata && std::find(bjdx.begin(), bjdx.end(),
      first_prefix) != bjdx.end()))
16767                     {
16768                         prefix_required = false;
16769                         oa->write_character(to_char_type('$'));
16770                         oa->write_character(first_prefix);
16771                     }
16772                 }
16773
16774                 if (use_count)
16775                 {
16776                     oa->write_character(to_char_type('#'));
16777                     write_number_with_ubjson_prefix(j.m_data.m_value.object->size(), true,
      use_bjdata);
16778                 }
16779
16780                 for (const auto& el : *j.m_data.m_value.object)
16781                 {
16782                     write_number_with_ubjson_prefix(el.first.size(), true, use_bjdata);
16783                     oa->write_characters(
16784                         reinterpret_cast<const CharType*>(el.first.c_str()),
```

```
16785                     el.first.size());
16786                 write_ubjson(el.second, use_count, use_type, prefix_required, use_bjdata,
    bjdata_version);
16787             }
16788
16789             if (!use_count)
16790             {
16791                 oa->write_character(to_char_type('}'));
16792             }
16793
16794             break;
16795         }
16796
16797         case value_t::discarded:
16798         default:
16799             break;
16800     }
16801 }
16802
16803  private:
16805   // BSON //
16807
16812   static std::size_t calc_bson_entry_header_size(const string_t& name, const BasicJsonType& j)
16813   {
16814       const auto it = name.find(static_cast<typename string_t::value_type>(0));
16815       if (JSON_HEDLEY_UNLIKELY(it != BasicJsonType::string_t::npos))
16816       {
16817           JSON_THROW(out_of_range::create(409, concat("BSON key cannot contain code point U+0000 (at
    byte ", std::to_string(it), ")"), &j));
16818       }
16819
16820       static_cast<void>(j);
16821       return /*id*/ 1ul + name.size() + /*zero-terminator*/1u;
16822   }
16823
16827   void write_bson_entry_header(const string_t& name,
16828                                const std::uint8_t element_type)
16829   {
16830       oa->write_character(to_char_type(element_type)); // boolean
16831       oa->write_characters(
16832           reinterpret_cast<const CharType*>(name.c_str()),
16833           name.size() + 1u);
16834   }
16835
16839   void write_bson_boolean(const string_t& name,
16840                           const bool value)
16841   {
16842       write_bson_entry_header(name, 0x08);
16843       oa->write_character(value ? to_char_type(0x01) : to_char_type(0x00));
16844   }
16845
16849   void write_bson_double(const string_t& name,
16850                          const double value)
16851   {
16852       write_bson_entry_header(name, 0x01);
16853       write_number<double>(value, true);
16854   }
16855
16859   static std::size_t calc_bson_string_size(const string_t& value)
16860   {
16861       return sizeof(std::int32_t) + value.size() + 1ul;
16862   }
16863
16867   void write_bson_string(const string_t& name,
16868                          const string_t& value)
16869   {
16870       write_bson_entry_header(name, 0x02);
16871
16872       write_number<std::int32_t>(static_cast<std::int32_t>(value.size() + 1ul), true);
16873       oa->write_characters(
16874           reinterpret_cast<const CharType*>(value.c_str()),
16875           value.size() + 1);
16876   }
16877
16881   void write_bson_null(const string_t& name)
16882   {
16883       write_bson_entry_header(name, 0x0A);
16884   }
16885
16889   static std::size_t calc_bson_integer_size(const std::int64_t value)
16890   {
16891       return (std::numeric_limits<std::int32_t>::min)() <= value && value <=
    (std::numeric_limits<std::int32_t>::max)()
16892               ? sizeof(std::int32_t)
16893               : sizeof(std::int64_t);
16894   }
16895
```

```
16899     void write_bson_integer(const string_t& name,
16900                             const std::int64_t value)
16901     {
16902         if ((std::numeric_limits<std::int32_t>::min)() <= value && value <=
      (std::numeric_limits<std::int32_t>::max)())
16903         {
16904             write_bson_entry_header(name, 0x10); // int32
16905             write_number<std::int32_t>(static_cast<std::int32_t>(value), true);
16906         }
16907         else
16908         {
16909             write_bson_entry_header(name, 0x12); // int64
16910             write_number<std::int64_t>(static_cast<std::int64_t>(value), true);
16911         }
16912     }
16913
16917     static constexpr std::size_t calc_bson_unsigned_size(const std::uint64_t value) noexcept
16918     {
16919         return (value <= static_cast<std::uint64_t>((std::numeric_limits<std::int32_t>::max)()))
16920                ? sizeof(std::int32_t)
16921                : sizeof(std::int64_t);
16922     }
16923
16927     void write_bson_unsigned(const string_t& name,
16928                              const BasicJsonType& j)
16929     {
16930         if (j.m_data.m_value.number_unsigned <=
      static_cast<std::uint64_t>((std::numeric_limits<std::int32_t>::max)()))
16931         {
16932             write_bson_entry_header(name, 0x10 /* int32 */);
16933             write_number<std::int32_t>(static_cast<std::int32_t>(j.m_data.m_value.number_unsigned),
      true);
16934         }
16935         else if (j.m_data.m_value.number_unsigned <=
      static_cast<std::uint64_t>((std::numeric_limits<std::int64_t>::max)()))
16936         {
16937             write_bson_entry_header(name, 0x12 /* int64 */);
16938             write_number<std::int64_t>(static_cast<std::int64_t>(j.m_data.m_value.number_unsigned),
      true);
16939         }
16940         else
16941         {
16942             write_bson_entry_header(name, 0x11 /* uint64 */);
16943             write_number<std::uint64_t>(static_cast<std::uint64_t>(j.m_data.m_value.number_unsigned),
      true);
16944         }
16945     }
16946
16950     void write_bson_object_entry(const string_t& name,
16951                                  const typename BasicJsonType::object_t& value)
16952     {
16953         write_bson_entry_header(name, 0x03); // object
16954         write_bson_object(value);
16955     }
16956
16960     static std::size_t calc_bson_array_size(const typename BasicJsonType::array_t& value)
16961     {
16962         std::size_t array_index = 0ul;
16963
16964         const std::size_t embedded_document_size = std::accumulate(std::begin(value), std::end(value),
      static_cast<std::size_t>(0), [&array_index](std::size_t result, const typename
      BasicJsonType::array_t::value_type & el)
16965         {
16966             return result + calc_bson_element_size(std::to_string(array_index++), el);
16967         });
16968
16969         return sizeof(std::int32_t) + embedded_document_size + 1ul;
16970     }
16971
16975     static std::size_t calc_bson_binary_size(const typename BasicJsonType::binary_t& value)
16976     {
16977         return sizeof(std::int32_t) + value.size() + 1ul;
16978     }
16979
16983     void write_bson_array(const string_t& name,
16984                           const typename BasicJsonType::array_t& value)
16985     {
16986         write_bson_entry_header(name, 0x04); // array
16987         write_number<std::int32_t>(static_cast<std::int32_t>(calc_bson_array_size(value)), true);
16988
16989         std::size_t array_index = 0ul;
16990
16991         for (const auto& el : value)
16992         {
16993             write_bson_element(std::to_string(array_index++), el);
16994         }
16995
```

```
16996         oa->write_character(to_char_type(0x00));
16997     }
16998
17002     void write_bson_binary(const string_t& name,
17003                            const binary_t& value)
17004     {
17005         write_bson_entry_header(name, 0x05);
17006
17007         write_number<std::int32_t>(static_cast<std::int32_t>(value.size()), true);
17008         write_number(value.has_subtype() ? static_cast<std::uint8_t>(value.subtype()) :
      static_cast<std::uint8_t>(0x00));
17009
17010         oa->write_characters(reinterpret_cast<const CharType*>(value.data()), value.size());
17011     }
17012
17017     static std::size_t calc_bson_element_size(const string_t& name,
17018             const BasicJsonType& j)
17019     {
17020         const auto header_size = calc_bson_entry_header_size(name, j);
17021         switch (j.type())
17022         {
17023             case value_t::object:
17024                 return header_size + calc_bson_object_size(*j.m_data.m_value.object);
17025
17026             case value_t::array:
17027                 return header_size + calc_bson_array_size(*j.m_data.m_value.array);
17028
17029             case value_t::binary:
17030                 return header_size + calc_bson_binary_size(*j.m_data.m_value.binary);
17031
17032             case value_t::boolean:
17033                 return header_size + 1ul;
17034
17035             case value_t::number_float:
17036                 return header_size + 8ul;
17037
17038             case value_t::number_integer:
17039                 return header_size + calc_bson_integer_size(j.m_data.m_value.number_integer);
17040
17041             case value_t::number_unsigned:
17042                 return header_size + calc_bson_unsigned_size(j.m_data.m_value.number_unsigned);
17043
17044             case value_t::string:
17045                 return header_size + calc_bson_string_size(*j.m_data.m_value.string);
17046
17047             case value_t::null:
17048                 return header_size + 0ul;
17049
17050             // LCOV_EXCL_START
17051             case value_t::discarded:
17052             default:
17053                 JSON_ASSERT(false); // NOLINT(cert-dcl03-c,hicpp-static-assert,misc-static-assert)
17054                 return 0ul;
17055                 // LCOV_EXCL_STOP
17056         }
17057     }
17058
17065     void write_bson_element(const string_t& name,
17066                             const BasicJsonType& j)
17067     {
17068         switch (j.type())
17069         {
17070             case value_t::object:
17071                 return write_bson_object_entry(name, *j.m_data.m_value.object);
17072
17073             case value_t::array:
17074                 return write_bson_array(name, *j.m_data.m_value.array);
17075
17076             case value_t::binary:
17077                 return write_bson_binary(name, *j.m_data.m_value.binary);
17078
17079             case value_t::boolean:
17080                 return write_bson_boolean(name, j.m_data.m_value.boolean);
17081
17082             case value_t::number_float:
17083                 return write_bson_double(name, j.m_data.m_value.number_float);
17084
17085             case value_t::number_integer:
17086                 return write_bson_integer(name, j.m_data.m_value.number_integer);
17087
17088             case value_t::number_unsigned:
17089                 return write_bson_unsigned(name, j);
17090
17091             case value_t::string:
17092                 return write_bson_string(name, *j.m_data.m_value.string);
17093
17094             case value_t::null:
```

```
17095                 return write_bson_null(name);
17096
17097             // LCOV_EXCL_START
17098             case value_t::discarded:
17099             default:
17100                 JSON_ASSERT(false); // NOLINT(cert-dcl03-c,hicpp-static-assert,misc-static-assert)
17101                 return;
17102                 // LCOV_EXCL_STOP
17103         }
17104     }
17105
17112     static std::size_t calc_bson_object_size(const typename BasicJsonType::object_t& value)
17113     {
17114         const std::size_t document_size = std::accumulate(value.begin(), value.end(),
      static_cast<std::size_t>(0),
17115                                           [](size_t result, const typename
      BasicJsonType::object_t::value_type & el)
17116         {
17117             return result += calc_bson_element_size(el.first, el.second);
17118         });
17119
17120         return sizeof(std::int32_t) + document_size + 1ul;
17121     }
17122
17127     void write_bson_object(const typename BasicJsonType::object_t& value)
17128     {
17129         write_number<std::int32_t>(static_cast<std::int32_t>(calc_bson_object_size(value)), true);
17130
17131         for (const auto& el : value)
17132         {
17133             write_bson_element(el.first, el.second);
17134         }
17135
17136         oa->write_character(to_char_type(0x00));
17137     }
17138
17140     // CBOR //
17142
17143     static constexpr CharType get_cbor_float_prefix(float /*unused*/)
17144     {
17145         return to_char_type(0xFA);  // Single-Precision Float
17146     }
17147
17148     static constexpr CharType get_cbor_float_prefix(double /*unused*/)
17149     {
17150         return to_char_type(0xFB);  // Double-Precision Float
17151     }
17152
17154     // MsgPack //
17156
17157     static constexpr CharType get_msgpack_float_prefix(float /*unused*/)
17158     {
17159         return to_char_type(0xCA);  // float 32
17160     }
17161
17162     static constexpr CharType get_msgpack_float_prefix(double /*unused*/)
17163     {
17164         return to_char_type(0xCB);  // float 64
17165     }
17166
17168     // UBJSON //
17170
17171     // UBJSON: write number (floating point)
17172     template<typename NumberType, typename std::enable_if<
17173                  std::is_floating_point<NumberType>::value, int>::type = 0>
17174     void write_number_with_ubjson_prefix(const NumberType n,
17175                                          const bool add_prefix,
17176                                          const bool use_bjdata)
17177     {
17178         if (add_prefix)
17179         {
17180             oa->write_character(get_ubjson_float_prefix(n));
17181         }
17182         write_number(n, use_bjdata);
17183     }
17184
17185     // UBJSON: write number (unsigned integer)
17186     template<typename NumberType, typename std::enable_if<
17187                  std::is_unsigned<NumberType>::value, int>::type = 0>
17188     void write_number_with_ubjson_prefix(const NumberType n,
17189                                          const bool add_prefix,
17190                                          const bool use_bjdata)
17191     {
17192         if (n <= static_cast<std::uint64_t>((std::numeric_limits<std::int8_t>::max)()))
17193         {
17194             if (add_prefix)
17195             {
```

```
17196                 oa->write_character(to_char_type('i'));  // int8
17197             }
17198             write_number(static_cast<std::uint8_t>(n), use_bjdata);
17199         }
17200         else if (n <= (std::numeric_limits<std::uint8_t>::max)())
17201         {
17202             if (add_prefix)
17203             {
17204                 oa->write_character(to_char_type('U'));  // uint8
17205             }
17206             write_number(static_cast<std::uint8_t>(n), use_bjdata);
17207         }
17208         else if (n <= static_cast<std::uint64_t>((std::numeric_limits<std::int16_t>::max)()))
17209         {
17210             if (add_prefix)
17211             {
17212                 oa->write_character(to_char_type('I'));  // int16
17213             }
17214             write_number(static_cast<std::int16_t>(n), use_bjdata);
17215         }
17216         else if (use_bjdata && n <= static_cast<uint64_t>((std::numeric_limits<uint16_t>::max)()))
17217         {
17218             if (add_prefix)
17219             {
17220                 oa->write_character(to_char_type('u'));  // uint16 - bjdata only
17221             }
17222             write_number(static_cast<std::uint16_t>(n), use_bjdata);
17223         }
17224         else if (n <= static_cast<std::uint64_t>((std::numeric_limits<std::int32_t>::max)()))
17225         {
17226             if (add_prefix)
17227             {
17228                 oa->write_character(to_char_type('l'));  // int32
17229             }
17230             write_number(static_cast<std::int32_t>(n), use_bjdata);
17231         }
17232         else if (use_bjdata && n <= static_cast<uint64_t>((std::numeric_limits<uint32_t>::max)()))
17233         {
17234             if (add_prefix)
17235             {
17236                 oa->write_character(to_char_type('m'));  // uint32 - bjdata only
17237             }
17238             write_number(static_cast<std::uint32_t>(n), use_bjdata);
17239         }
17240         else if (n <= static_cast<std::uint64_t>((std::numeric_limits<std::int64_t>::max)()))
17241         {
17242             if (add_prefix)
17243             {
17244                 oa->write_character(to_char_type('L'));  // int64
17245             }
17246             write_number(static_cast<std::int64_t>(n), use_bjdata);
17247         }
17248         else if (use_bjdata && n <= (std::numeric_limits<uint64_t>::max)())
17249         {
17250             if (add_prefix)
17251             {
17252                 oa->write_character(to_char_type('M'));  // uint64 - bjdata only
17253             }
17254             write_number(static_cast<std::uint64_t>(n), use_bjdata);
17255         }
17256         else
17257         {
17258             if (add_prefix)
17259             {
17260                 oa->write_character(to_char_type('H'));  // high-precision number
17261             }
17262
17263             const auto number = BasicJsonType(n).dump();
17264             write_number_with_ubjson_prefix(number.size(), true, use_bjdata);
17265             for (std::size_t i = 0; i < number.size(); ++i)
17266             {
17267                 oa->write_character(to_char_type(static_cast<std::uint8_t>(number[i])));
17268             }
17269         }
17270     }
17271
17272     // UBJSON: write number (signed integer)
17273     template < typename NumberType, typename std::enable_if <
17274                    std::is_signed<NumberType>::value&&
17275                    !std::is_floating_point<NumberType>::value, int >::type = 0 >
17276     void write_number_with_ubjson_prefix(const NumberType n,
17277                                          const bool add_prefix,
17278                                          const bool use_bjdata)
17279     {
17280         if ((std::numeric_limits<std::int8_t>::min)() <= n && n <=
    (std::numeric_limits<std::int8_t>::max)())
17281         {
```

```
17282             if (add_prefix)
17283             {
17284                 oa->write_character(to_char_type('i'));  // int8
17285             }
17286             write_number(static_cast<std::int8_t>(n), use_bjdata);
17287         }
17288         else if (static_cast<std::int64_t>((std::numeric_limits<std::uint8_t>::min)()) <= n && n <=
      static_cast<std::int64_t>((std::numeric_limits<std::uint8_t>::max)()))
17289         {
17290             if (add_prefix)
17291             {
17292                 oa->write_character(to_char_type('U'));  // uint8
17293             }
17294             write_number(static_cast<std::uint8_t>(n), use_bjdata);
17295         }
17296         else if ((std::numeric_limits<std::int16_t>::min)() <= n && n <=
      (std::numeric_limits<std::int16_t>::max)())
17297         {
17298             if (add_prefix)
17299             {
17300                 oa->write_character(to_char_type('I'));  // int16
17301             }
17302             write_number(static_cast<std::int16_t>(n), use_bjdata);
17303         }
17304         else if (use_bjdata && (static_cast<std::int64_t>((std::numeric_limits<std::uint16_t>::min)())
      <= n && n <= static_cast<std::int64_t>((std::numeric_limits<std::uint16_t>::max)())))
17305         {
17306             if (add_prefix)
17307             {
17308                 oa->write_character(to_char_type('u'));  // uint16 - bjdata only
17309             }
17310             write_number(static_cast<uint16_t>(n), use_bjdata);
17311         }
17312         else if ((std::numeric_limits<std::int32_t>::min)() <= n && n <=
      (std::numeric_limits<std::int32_t>::max)())
17313         {
17314             if (add_prefix)
17315             {
17316                 oa->write_character(to_char_type('l'));  // int32
17317             }
17318             write_number(static_cast<std::int32_t>(n), use_bjdata);
17319         }
17320         else if (use_bjdata && (static_cast<std::int64_t>((std::numeric_limits<std::uint32_t>::min)())
      <= n && n <= static_cast<std::int64_t>((std::numeric_limits<std::uint32_t>::max)())))
17321         {
17322             if (add_prefix)
17323             {
17324                 oa->write_character(to_char_type('m'));  // uint32 - bjdata only
17325             }
17326             write_number(static_cast<uint32_t>(n), use_bjdata);
17327         }
17328         else if ((std::numeric_limits<std::int64_t>::min)() <= n && n <=
      (std::numeric_limits<std::int64_t>::max)())
17329         {
17330             if (add_prefix)
17331             {
17332                 oa->write_character(to_char_type('L'));  // int64
17333             }
17334             write_number(static_cast<std::int64_t>(n), use_bjdata);
17335         }
17336         // LCOV_EXCL_START
17337         else
17338         {
17339             if (add_prefix)
17340             {
17341                 oa->write_character(to_char_type('H'));  // high-precision number
17342             }
17343
17344             const auto number = BasicJsonType(n).dump();
17345             write_number_with_ubjson_prefix(number.size(), true, use_bjdata);
17346             for (std::size_t i = 0; i < number.size(); ++i)
17347             {
17348                 oa->write_character(to_char_type(static_cast<std::uint8_t>(number[i])));
17349             }
17350         }
17351         // LCOV_EXCL_STOP
17352     }
17353
17357     CharType ubjson_prefix(const BasicJsonType& j, const bool use_bjdata) const noexcept
17358     {
17359         switch (j.type())
17360         {
17361             case value_t::null:
17362                 return 'Z';
17363
17364             case value_t::boolean:
17365                 return j.m_data.m_value.boolean ? 'T' : 'F';
```

```
17366
17367             case value_t::number_integer:
17368             {
17369                 if ((std::numeric_limits<std::int8_t>::min)() <= j.m_data.m_value.number_integer &&
      j.m_data.m_value.number_integer <= (std::numeric_limits<std::int8_t>::max)())
17370                 {
17371                     return 'i';
17372                 }
17373                 if ((std::numeric_limits<std::uint8_t>::min)() <= j.m_data.m_value.number_integer &&
      j.m_data.m_value.number_integer <= (std::numeric_limits<std::uint8_t>::max)())
17374                 {
17375                     return 'U';
17376                 }
17377                 if ((std::numeric_limits<std::int16_t>::min)() <= j.m_data.m_value.number_integer &&
      j.m_data.m_value.number_integer <= (std::numeric_limits<std::int16_t>::max)())
17378                 {
17379                     return 'I';
17380                 }
17381                 if (use_bjdata && ((std::numeric_limits<std::uint16_t>::min)() <=
      j.m_data.m_value.number_integer && j.m_data.m_value.number_integer <=
      (std::numeric_limits<std::uint16_t>::max)()))
17382                 {
17383                     return 'u';
17384                 }
17385                 if ((std::numeric_limits<std::int32_t>::min)() <= j.m_data.m_value.number_integer &&
      j.m_data.m_value.number_integer <= (std::numeric_limits<std::int32_t>::max)())
17386                 {
17387                     return 'l';
17388                 }
17389                 if (use_bjdata && ((std::numeric_limits<std::uint32_t>::min)() <=
      j.m_data.m_value.number_integer && j.m_data.m_value.number_integer <=
      (std::numeric_limits<std::uint32_t>::max)()))
17390                 {
17391                     return 'm';
17392                 }
17393                 if ((std::numeric_limits<std::int64_t>::min)() <= j.m_data.m_value.number_integer &&
      j.m_data.m_value.number_integer <= (std::numeric_limits<std::int64_t>::max)())
17394                 {
17395                     return 'L';
17396                 }
17397                 // anything else is treated as a high-precision number
17398                 return 'H'; // LCOV_EXCL_LINE
17399             }
17400
17401             case value_t::number_unsigned:
17402             {
17403                 if (j.m_data.m_value.number_unsigned <=
      static_cast<std::uint64_t>((std::numeric_limits<std::int8_t>::max)()))
17404                 {
17405                     return 'i';
17406                 }
17407                 if (j.m_data.m_value.number_unsigned <=
      static_cast<std::uint64_t>((std::numeric_limits<std::uint8_t>::max)()))
17408                 {
17409                     return 'U';
17410                 }
17411                 if (j.m_data.m_value.number_unsigned <=
      static_cast<std::uint64_t>((std::numeric_limits<std::int16_t>::max)()))
17412                 {
17413                     return 'I';
17414                 }
17415                 if (use_bjdata && j.m_data.m_value.number_unsigned <=
      static_cast<std::uint64_t>((std::numeric_limits<std::uint16_t>::max)()))
17416                 {
17417                     return 'u';
17418                 }
17419                 if (j.m_data.m_value.number_unsigned <=
      static_cast<std::uint64_t>((std::numeric_limits<std::int32_t>::max)()))
17420                 {
17421                     return 'l';
17422                 }
17423                 if (use_bjdata && j.m_data.m_value.number_unsigned <=
      static_cast<std::uint64_t>((std::numeric_limits<std::uint32_t>::max)()))
17424                 {
17425                     return 'm';
17426                 }
17427                 if (j.m_data.m_value.number_unsigned <=
      static_cast<std::uint64_t>((std::numeric_limits<std::int64_t>::max)()))
17428                 {
17429                     return 'L';
17430                 }
17431                 if (use_bjdata && j.m_data.m_value.number_unsigned <=
      (std::numeric_limits<std::uint64_t>::max)())
17432                 {
17433                     return 'M';
17434                 }
17435                 // anything else is treated as a high-precision number
```

```
17436                 return 'H'; // LCOV_EXCL_LINE
17437             }
17438
17439             case value_t::number_float:
17440                 return get_ubjson_float_prefix(j.m_data.m_value.number_float);
17441
17442             case value_t::string:
17443                 return 'S';
17444
17445             case value_t::array: // fallthrough
17446             case value_t::binary:
17447                 return '[';
17448
17449             case value_t::object:
17450                 return '{';
17451
17452             case value_t::discarded:
17453             default:  // discarded values
17454                 return 'N';
17455         }
17456     }
17457
17458     static constexpr CharType get_ubjson_float_prefix(float /*unused*/)
17459     {
17460         return 'd';  // float 32
17461     }
17462
17463     static constexpr CharType get_ubjson_float_prefix(double /*unused*/)
17464     {
17465         return 'D';  // float 64
17466     }
17467
17471     bool write_bjdata_ndarray(const typename BasicJsonType::object_t& value, const bool use_count,
      const bool use_type, const bjdata_version_t bjdata_version)
17472     {
17473         std::map<string_t, CharType> bjdtype = {{"uint8", 'U'},  {"int8", 'i'},  {"uint16", 'u'},
      {"int16", 'I'},
17474             {"uint32", 'm'}, {"int32", 'l'}, {"uint64", 'M'}, {"int64", 'L'}, {"single", 'd'},
      {"double", 'D'},
17475             {"char", 'C'}, {"byte", 'B'}
17476         };
17477
17478         string_t key = "_ArrayType_";
17479         auto it = bjdtype.find(static_cast<string_t>(value.at(key)));
17480         if (it == bjdtype.end())
17481         {
17482             return true;
17483         }
17484         CharType dtype = it->second;
17485
17486         key = "_ArraySize_";
17487         std::size_t len = (value.at(key).empty() ? 0 : 1);
17488         for (const auto& el : value.at(key))
17489         {
17490             len *= static_cast<std::size_t>(el.m_data.m_value.number_unsigned);
17491         }
17492
17493         key = "_ArrayData_";
17494         if (value.at(key).size() != len)
17495         {
17496             return true;
17497         }
17498
17499         oa->write_character('[');
17500         oa->write_character('$');
17501         oa->write_character(dtype);
17502         oa->write_character('#');
17503
17504         key = "_ArraySize_";
17505         write_ubjson(value.at(key), use_count, use_type, true,  true, bjdata_version);
17506
17507         key = "_ArrayData_";
17508         if (dtype == 'U' || dtype == 'C' || dtype == 'B')
17509         {
17510             for (const auto& el : value.at(key))
17511             {
17512                 write_number(static_cast<std::uint8_t>(el.m_data.m_value.number_unsigned), true);
17513             }
17514         }
17515         else if (dtype == 'i')
17516         {
17517             for (const auto& el : value.at(key))
17518             {
17519                 write_number(static_cast<std::int8_t>(el.m_data.m_value.number_integer), true);
17520             }
17521         }
17522         else if (dtype == 'u')
```

```
17523         {
17524             for (const auto& el : value.at(key))
17525             {
17526                 write_number(static_cast<std::uint16_t>(el.m_data.m_value.number_unsigned), true);
17527             }
17528         }
17529         else if (dtype == 'I')
17530         {
17531             for (const auto& el : value.at(key))
17532             {
17533                 write_number(static_cast<std::int16_t>(el.m_data.m_value.number_integer), true);
17534             }
17535         }
17536         else if (dtype == 'm')
17537         {
17538             for (const auto& el : value.at(key))
17539             {
17540                 write_number(static_cast<std::uint32_t>(el.m_data.m_value.number_unsigned), true);
17541             }
17542         }
17543         else if (dtype == 'l')
17544         {
17545             for (const auto& el : value.at(key))
17546             {
17547                 write_number(static_cast<std::int32_t>(el.m_data.m_value.number_integer), true);
17548             }
17549         }
17550         else if (dtype == 'M')
17551         {
17552             for (const auto& el : value.at(key))
17553             {
17554                 write_number(static_cast<std::uint64_t>(el.m_data.m_value.number_unsigned), true);
17555             }
17556         }
17557         else if (dtype == 'L')
17558         {
17559             for (const auto& el : value.at(key))
17560             {
17561                 write_number(static_cast<std::int64_t>(el.m_data.m_value.number_integer), true);
17562             }
17563         }
17564         else if (dtype == 'd')
17565         {
17566             for (const auto& el : value.at(key))
17567             {
17568                 write_number(static_cast<float>(el.m_data.m_value.number_float), true);
17569             }
17570         }
17571         else if (dtype == 'D')
17572         {
17573             for (const auto& el : value.at(key))
17574             {
17575                 write_number(static_cast<double>(el.m_data.m_value.number_float), true);
17576             }
17577         }
17578         return false;
17579     }
17580
17582     // Utility functions //
17584
17585     /*
17586     @brief write a number to output input
17587     @param[in] n number of type @a NumberType
17588     @param[in] OutputIsLittleEndian Set to true if output data is
17589                                     required to be little endian
17590     @tparam NumberType the type of the number
17591
17592     @note This function needs to respect the system's endianness, because bytes
17593           in CBOR, MessagePack, and UBJSON are stored in network order (big
17594           endian) and therefore need reordering on little endian systems.
17595           On the other hand, BSON and BJData use little endian and should reorder
17596           on big endian systems.
17597     */
17598     template<typename NumberType>
17599     void write_number(const NumberType n, const bool OutputIsLittleEndian = false)
17600     {
17601         // step 1: write the number to an array of length NumberType
17602         std::array<CharType, sizeof(NumberType)> vec{};
17603         std::memcpy(vec.data(), &n, sizeof(NumberType));
17604
17605         // step 2: write the array to output (with possible reordering)
17606         if (is_little_endian != OutputIsLittleEndian)
17607         {
17608             // reverse byte order prior to conversion if necessary
17609             std::reverse(vec.begin(), vec.end());
17610         }
17611
```

```
17612         oa->write_characters(vec.data(), sizeof(NumberType));
17613     }
17614
17615     void write_compact_float(const number_float_t n, detail::input_format_t format)
17616     {
17617 #ifdef __GNUC__
17618 #pragma GCC diagnostic push
17619 #pragma GCC diagnostic ignored "-Wfloat-equal"
17620 #endif
17621         if (!std::isfinite(n) || ((static_cast<double>(n) >=
      static_cast<double>(std::numeric_limits<float>::lowest()) &&
17622                                    static_cast<double>(n) <=
      static_cast<double>((std::numeric_limits<float>::max)()) &&
17623                                    static_cast<double>(static_cast<float>(n)) ==
      static_cast<double>(n))))
17624         {
17625             oa->write_character(format == detail::input_format_t::cbor
17626                                 ? get_cbor_float_prefix(static_cast<float>(n))
17627                                 : get_msgpack_float_prefix(static_cast<float>(n)));
17628             write_number(static_cast<float>(n));
17629         }
17630         else
17631         {
17632             oa->write_character(format == detail::input_format_t::cbor
17633                                 ? get_cbor_float_prefix(n)
17634                                 : get_msgpack_float_prefix(n));
17635             write_number(n);
17636         }
17637 #ifdef __GNUC__
17638 #pragma GCC diagnostic pop
17639 #endif
17640     }
17641
17642   public:
17643     // The following to_char_type functions are implement the conversion
17644     // between uint8_t and CharType. In case CharType is not unsigned,
17645     // such a conversion is required to allow values greater than 128.
17646     // See <https://github.com/nlohmann/json/issues/1286> for a discussion.
17647     template < typename C = CharType,
17648                enable_if_t < std::is_signed<C>::value && std::is_signed<char>::value > * = nullptr >
17649     static constexpr CharType to_char_type(std::uint8_t x) noexcept
17650     {
17651         return *reinterpret_cast<char*>(&x);
17652     }
17653
17654     template < typename C = CharType,
17655                enable_if_t < std::is_signed<C>::value && std::is_unsigned<char>::value > * = nullptr >
17656     static CharType to_char_type(std::uint8_t x) noexcept
17657     {
17658         // The std::is_trivial trait is deprecated in C++26. The replacement is to use
17659         // std::is_trivially_copyable and std::is_trivially_default_constructible.
17660         // However, some older library implementations support std::is_trivial
17661         // but not all the std::is_trivially_* traits.
17662         // Since detecting full support across all libraries is difficult,
17663         // we use std::is_trivial unless we are using a standard where it has been deprecated.
17664         // For more details, see: https://github.com/nlohmann/json/pull/4775#issuecomment-2884361627
17665 #ifdef JSON_HAS_CPP_26
17666         static_assert(std::is_trivially_copyable<CharType>::value, "CharType must be trivially
      copyable");
17667         static_assert(std::is_trivially_default_constructible<CharType>::value, "CharType must be
      trivially default constructible");
17668 #else
17669         static_assert(std::is_trivial<CharType>::value, "CharType must be trivial");
17670 #endif
17671
17672         static_assert(sizeof(std::uint8_t) == sizeof(CharType), "size of CharType must be equal to
      std::uint8_t");
17673         CharType result;
17674         std::memcpy(&result, &x, sizeof(x));
17675         return result;
17676     }
17677
17678     template<typename C = CharType,
17679              enable_if_t<std::is_unsigned<C>::value>* = nullptr>
17680     static constexpr CharType to_char_type(std::uint8_t x) noexcept
17681     {
17682         return x;
17683     }
17684
17685     template < typename InputCharType, typename C = CharType,
17686                enable_if_t <
17687                    std::is_signed<C>::value &&
17688                    std::is_signed<char>::value &&
17689                    std::is_same<char, typename std::remove_cv<InputCharType>::type>::value
17690                    > * = nullptr >
17691     static constexpr CharType to_char_type(InputCharType x) noexcept
17692     {
```

```
17693         return x;
17694     }
17695
17696   private:
17698     const bool is_little_endian = little_endianness();
17699
17701     output_adapter_t<CharType> oa = nullptr;
17702 };
17703
17704 }  // namespace detail
17705 NLOHMANN_JSON_NAMESPACE_END
17706
17707 // #include <nlohmann/detail/output/output_adapters.hpp>
17708
17709 // #include <nlohmann/detail/output/serializer.hpp>
17710 //     __ _____ _____ _____
17711 //  __|  |   __|     |   | |  JSON for Modern C++
17712 // |  |  |__   |  |  | | | |  version 3.12.0
17713 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
17714 //
17715 // SPDX-FileCopyrightText: 2008, 2009 Björn Hoehrmann <bjoern@hoehrmann.de>
17716 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
17717 // SPDX-License-Identifier: MIT
17718
17719
17720
17721 #include <algorithm> // reverse, remove, fill, find, none_of
17722 #include <array> // array
17723 #include <clocale> // localeconv, lconv
17724 #include <cmath> // labs, isfinite, isnan, signbit
17725 #include <cstddef> // size_t, ptrdiff_t
17726 #include <cstdint> // uint8_t
17727 #include <cstdio> // snprintf
17728 #include <limits> // numeric_limits
17729 #include <string> // string, char_traits
17730 #include <iomanip> // setfill, setw
17731 #include <type_traits> // is_same
17732 #include <utility> // move
17733
17734 // #include <nlohmann/detail/conversions/to_chars.hpp>
17735 //     __ _____ _____ _____
17736 //  __|  |   __|     |   | |  JSON for Modern C++
17737 // |  |  |__   |  |  | | | |  version 3.12.0
17738 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
17739 //
17740 // SPDX-FileCopyrightText: 2009 Florian Loitsch <https://florian.loitsch.com/>
17741 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
17742 // SPDX-License-Identifier: MIT
17743
17744
17745
17746 #include <array> // array
17747 #include <cmath>   // signbit, isfinite
17748 #include <cstdint> // intN_t, uintN_t
17749 #include <cstring> // memcpy, memmove
17750 #include <limits> // numeric_limits
17751 #include <type_traits> // conditional
17752
17753 // #include <nlohmann/detail/macro_scope.hpp>
17754
17755
17756 NLOHMANN_JSON_NAMESPACE_BEGIN
17757 namespace detail
17758 {
17759
17779 namespace dtoa_impl
17780 {
17781
17782 template<typename Target, typename Source>
17783 Target reinterpret_bits(const Source source)
17784 {
17785     static_assert(sizeof(Target) == sizeof(Source), "size mismatch");
17786
17787     Target target;
17788     std::memcpy(&target, &source, sizeof(Source));
17789     return target;
17790 }
17791
17792 struct diyfp // f * 2^e
17793 {
17794     static constexpr int kPrecision = 64; // = q
17795
17796     std::uint64_t f = 0;
17797     int e = 0;
17798
17799     constexpr diyfp(std::uint64_t f_, int e_) noexcept : f(f_), e(e_) {}
17800
```

```
17805     static diyfp sub(const diyfp& x, const diyfp& y) noexcept
17806     {
17807         JSON_ASSERT(x.e == y.e);
17808         JSON_ASSERT(x.f >= y.f);
17809
17810         return {x.f - y.f, x.e};
17811     }
17812
17817     static diyfp mul(const diyfp& x, const diyfp& y) noexcept
17818     {
17819         static_assert(kPrecision == 64, "internal error");
17820
17821         // Computes:
17822         //  f = round((x.f * y.f) / 2^q)
17823         //  e = x.e + y.e + q
17824
17825         // Emulate the 64-bit * 64-bit multiplication:
17826         //
17827         // p = u * v
17828         //   = (u_lo + 2^32 u_hi) (v_lo + 2^32 v_hi)
17829         //   = (u_lo v_lo         ) + 2^32 ((u_lo v_hi         ) + (u_hi v_lo         )) + 2^64 (u_hi
    v_hi         )
17830         //   = (p0                ) + 2^32 ((p1                ) + (p2                )) + 2^64 (p3
    )
17831         //   = (p0_lo + 2^32 p0_hi) + 2^32 ((p1_lo + 2^32 p1_hi) + (p2_lo + 2^32 p2_hi)) + 2^64 (p3
    )
17832         //   = (p0_lo             ) + 2^32 (p0_hi + p1_lo + p2_lo                      ) + 2^64 (p1_hi
    + p2_hi + p3)
17833         //   = (p0_lo             ) + 2^32 (Q                                          ) + 2^64 (H
    )
17834         //   = (p0_lo             ) + 2^32 (Q_lo + 2^32 Q_hi                           ) + 2^64 (H
    )
17835         //
17836         // (Since Q might be larger than 2^32 - 1)
17837         //
17838         //   = (p0_lo + 2^32 Q_lo) + 2^64 (Q_hi + H)
17839         //
17840         // (Q_hi + H does not overflow a 64-bit int)
17841         //
17842         //   = p_lo + 2^64 p_hi
17843
17844         const std::uint64_t u_lo = x.f & 0xFFFFFFFFu;
17845         const std::uint64_t u_hi = x.f >> 32u;
17846         const std::uint64_t v_lo = y.f & 0xFFFFFFFFu;
17847         const std::uint64_t v_hi = y.f >> 32u;
17848
17849         const std::uint64_t p0 = u_lo * v_lo;
17850         const std::uint64_t p1 = u_lo * v_hi;
17851         const std::uint64_t p2 = u_hi * v_lo;
17852         const std::uint64_t p3 = u_hi * v_hi;
17853
17854         const std::uint64_t p0_hi = p0 >> 32u;
17855         const std::uint64_t p1_lo = p1 & 0xFFFFFFFFu;
17856         const std::uint64_t p1_hi = p1 >> 32u;
17857         const std::uint64_t p2_lo = p2 & 0xFFFFFFFFu;
17858         const std::uint64_t p2_hi = p2 >> 32u;
17859
17860         std::uint64_t Q = p0_hi + p1_lo + p2_lo;
17861
17862         // The full product might now be computed as
17863         //
17864         // p_hi = p3 + p2_hi + p1_hi + (Q >> 32)
17865         // p_lo = p0_lo + (Q << 32)
17866         //
17867         // But in this particular case here, the full p_lo is not required.
17868         // Effectively, we only need to add the highest bit in p_lo to p_hi (and
17869         // Q_hi + 1 does not overflow).
17870
17871         Q += std::uint64_t{1} << (64u - 32u - 1u); // round, ties up
17872
17873         const std::uint64_t h = p3 + p2_hi + p1_hi + (Q >> 32u);
17874
17875         return {h, x.e + y.e + 64};
17876     }
17877
17882     static diyfp normalize(diyfp x) noexcept
17883     {
17884         JSON_ASSERT(x.f != 0);
17885
17886         while ((x.f >> 63u) == 0)
17887         {
17888             x.f <<= 1u;
17889             x.e--;
17890         }
17891
17892         return x;
17893     }
```

```
17894
17899     static diyfp normalize_to(const diyfp& x, const int target_exponent) noexcept
17900     {
17901         const int delta = x.e - target_exponent;
17902
17903         JSON_ASSERT(delta >= 0);
17904         JSON_ASSERT(((x.f << delta) >> delta) == x.f);
17905
17906         return {x.f << delta, target_exponent};
17907     }
17908 };
17909
17910 struct boundaries
17911 {
17912     diyfp w;
17913     diyfp minus;
17914     diyfp plus;
17915 };
17916
17923 template<typename FloatType>
17924 boundaries compute_boundaries(FloatType value)
17925 {
17926     JSON_ASSERT(std::isfinite(value));
17927     JSON_ASSERT(value > 0);
17928
17929     // Convert the IEEE representation into a diyfp.
17930     //
17931     // If v is denormal:
17932     //      value = 0.F * 2^(1 - bias) = (          F) * 2^(1 - bias - (p-1))
17933     // If v is normalized:
17934     //      value = 1.F * 2^(E - bias) = (2^(p-1) + F) * 2^(E - bias - (p-1))
17935
17936     static_assert(std::numeric_limits<FloatType>::is_iec559,
17937                   "internal error: dtoa_short requires an IEEE-754 floating-point implementation");
17938
17939     constexpr int      kPrecision = std::numeric_limits<FloatType>::digits; // = p (includes the
      hidden bit)
17940     constexpr int      kBias      = std::numeric_limits<FloatType>::max_exponent - 1 + (kPrecision -
      1);
17941     constexpr int      kMinExp    = 1 - kBias;
17942     constexpr std::uint64_t kHiddenBit = std::uint64_t{1} << (kPrecision - 1); // = 2^(p-1)
17943
17944     using bits_type = typename std::conditional<kPrecision == 24, std::uint32_t, std::uint64_t
      >::type;
17945
17946     const auto bits = static_cast<std::uint64_t>(reinterpret_bits<bits_type>(value));
17947     const std::uint64_t E = bits >> (kPrecision - 1);
17948     const std::uint64_t F = bits & (kHiddenBit - 1);
17949
17950     const bool is_denormal = E == 0;
17951     const diyfp v = is_denormal
17952                     ? diyfp(F, kMinExp)
17953                     : diyfp(F + kHiddenBit, static_cast<int>(E) - kBias);
17954
17955     // Compute the boundaries m- and m+ of the floating-point value
17956     // v = f * 2^e.
17957     //
17958     // Determine v- and v+, the floating-point predecessor and successor of v,
17959     // respectively.
17960     //
17961     //      v- = v - 2^e        if f != 2^(p-1) or e == e_min                (A)
17962     //         = v - 2^(e-1)    if f == 2^(p-1) and e > e_min                (B)
17963     //
17964     //      v+ = v + 2^e
17965     //
17966     // Let m- = (v- + v) / 2 and m+ = (v + v+) / 2. All real numbers _strictly_
17967     // between m- and m+ round to v, regardless of how the input rounding
17968     // algorithm breaks ties.
17969     //
17970     //      ---+-------------+-------------+-------------+-------------+---  (A)
17971     //         v-            m-            v             m+            v+
17972     //
17973     //      -----------------+------+------+-------------+-------------+---  (B)
17974     //                       v-     m-     v             m+            v+
17975
17976     const bool lower_boundary_is_closer = F == 0 && E > 1;
17977     const diyfp m_plus = diyfp((2 * v.f) + 1, v.e - 1);
17978     const diyfp m_minus = lower_boundary_is_closer
17979                           ? diyfp((4 * v.f) - 1, v.e - 2)  // (B)
17980                           : diyfp((2 * v.f) - 1, v.e - 1); // (A)
17981
17982     // Determine the normalized w+ = m+.
17983     const diyfp w_plus = diyfp::normalize(m_plus);
17984
17985     // Determine w- = m- such that e_(w-) = e_(w+).
17986     const diyfp w_minus = diyfp::normalize_to(m_minus, w_plus.e);
17987
```

```
17988     return {diyfp::normalize(v), w_minus, w_plus};
17989 }
17990
17991 // Given normalized diyfp w, Grisu needs to find a (normalized) cached
17992 // power-of-ten c, such that the exponent of the product c * w = f * 2^e lies
17993 // within a certain range [alpha, gamma] (Definition 3.2 from [1])
17994 //
17995 //      alpha <= e = e_c + e_w + q <= gamma
17996 //
17997 // or
17998 //
17999 //      f_c * f_w * 2^alpha <= f_c 2^(e_c) * f_w 2^(e_w) * 2^q
18000 //                          <= f_c * f_w * 2^gamma
18001 //
18002 // Since c and w are normalized, i.e. 2^(q-1) <= f < 2^q, this implies
18003 //
18004 //      2^(q-1) * 2^(q-1) * 2^alpha <= c * w * 2^q < 2^q * 2^q * 2^gamma
18005 //
18006 // or
18007 //
18008 //      2^(q - 2 + alpha) <= c * w < 2^(q + gamma)
18009 //
18010 // The choice of (alpha,gamma) determines the size of the table and the form of
18011 // the digit generation procedure. Using (alpha,gamma)=(-60,-32) works out well
18012 // in practice:
18013 //
18014 // The idea is to cut the number c * w = f * 2^e into two parts, which can be
18015 // processed independently: An integral part p1, and a fractional part p2:
18016 //
18017 //      f * 2^e = ( (f div 2^-e) * 2^-e + (f mod 2^-e) ) * 2^e
18018 //              = (f div 2^-e) + (f mod 2^-e) * 2^e
18019 //              = p1 + p2 * 2^e
18020 //
18021 // The conversion of p1 into decimal form requires a series of divisions and
18022 // modulos by (a power of) 10. These operations are faster for 32-bit than for
18023 // 64-bit integers, so p1 should ideally fit into a 32-bit integer. This can be
18024 // achieved by choosing
18025 //
18026 //      -e >= 32   or   e <= -32 := gamma
18027 //
18028 // In order to convert the fractional part
18029 //
18030 //      p2 * 2^e = p2 / 2^-e = d[-1] / 10^1 + d[-2] / 10^2 + ...
18031 //
18032 // into decimal form, the fraction is repeatedly multiplied by 10 and the digits
18033 // d[-i] are extracted in order:
18034 //
18035 //      (10 * p2) div 2^-e = d[-1]
18036 //      (10 * p2) mod 2^-e = d[-2] / 10^1 + ...
18037 //
18038 // The multiplication by 10 must not overflow. It is sufficient to choose
18039 //
18040 //      10 * p2 < 16 * p2 = 2^4 * p2 <= 2^64.
18041 //
18042 // Since p2 = f mod 2^-e < 2^-e,
18043 //
18044 //      -e <= 60   or   e >= -60 := alpha
18045
18046 constexpr int kAlpha = -60;
18047 constexpr int kGamma = -32;
18048
18049 struct cached_power // c = f * 2^e ~= 10^k
18050 {
18051     std::uint64_t f;
18052     int e;
18053     int k;
18054 };
18055
18063 inline cached_power get_cached_power_for_binary_exponent(int e)
18064 {
18065     // Now
18066     //
18067     //      alpha <= e_c + e + q <= gamma                                    (1)
18068     //      ==> f_c * 2^alpha <= c * 2^e * 2^q
18069     //
18070     // and since the c's are normalized, 2^(q-1) <= f_c,
18071     //
18072     //      ==> 2^(q - 1 + alpha) <= c * 2^(e + q)
18073     //      ==> 2^(alpha - e - 1) <= c
18074     //
18075     // If c were an exact power of ten, i.e. c = 10^k, one may determine k as
18076     //
18077     //      k = ceil( log_10( 2^(alpha - e - 1) ) )
18078     //        = ceil( (alpha - e - 1) * log_10(2) )
18079     //
18080     // From the paper:
18081     // "In theory the result of the procedure could be wrong since c is rounded,
```

```
18082     //  and the computation itself is approximated [...]. In practice, however,
18083     //  this simple function is sufficient."
18084     //
18085     // For IEEE double precision floating-point numbers converted into
18086     // normalized diyfp's w = f * 2^e, with q = 64,
18087     //
18088     //      e >= -1022      (min IEEE exponent)
18089     //           -52        (p - 1)
18090     //           -52        (p - 1, possibly normalize denormal IEEE numbers)
18091     //           -11        (normalize the diyfp)
18092     //         = -1137
18093     //
18094     // and
18095     //
18096     //      e <= +1023      (max IEEE exponent)
18097     //           -52        (p - 1)
18098     //           -11        (normalize the diyfp)
18099     //         = 960
18100     //
18101     // This binary exponent range [-1137,960] results in a decimal exponent
18102     // range [-307,324]. One does not need to store a cached power for each
18103     // k in this range. For each such k it suffices to find a cached power
18104     // such that the exponent of the product lies in [alpha,gamma].
18105     // This implies that the difference of the decimal exponents of adjacent
18106     // table entries must be less than or equal to
18107     //
18108     //      floor( (gamma - alpha) * log_10(2) ) = 8.
18109     //
18110     // (A smaller distance gamma-alpha would require a larger table.)
18111
18112     // NB:
18113     // Actually, this function returns c, such that -60 <= e_c + e + 64 <= -34.
18114
18115     constexpr int kCachedPowersMinDecExp = -300;
18116     constexpr int kCachedPowersDecStep = 8;
18117
18118     static constexpr std::array<cached_power, 79> kCachedPowers =
18119     {
18120         {
18121             { 0xAB70FE17C79AC6CA, -1060, -300 },
18122             { 0xFF77B1FCBEBCDC4F, -1034, -292 },
18123             { 0xBE5691EF416BD60C, -1007, -284 },
18124             { 0x8DD01FAD907FFC3C,  -980, -276 },
18125             { 0xD3515C2831559A83,  -954, -268 },
18126             { 0x9D71AC8FADA6C9B5,  -927, -260 },
18127             { 0xEA9C227723EE8BCB,  -901, -252 },
18128             { 0xAECC49914078536D,  -874, -244 },
18129             { 0x823C12795DB6CE57,  -847, -236 },
18130             { 0xC21094364DFB5637,  -821, -228 },
18131             { 0x9096EA6F3848984F,  -794, -220 },
18132             { 0xD77485CB25823AC7,  -768, -212 },
18133             { 0xA086CFCD97BF97F4,  -741, -204 },
18134             { 0xEF340A98172AACE5,  -715, -196 },
18135             { 0xB23867FB2A35B28E,  -688, -188 },
18136             { 0x84C8D4DFD2C63F3B,  -661, -180 },
18137             { 0xC5DD44271AD3CDBA,  -635, -172 },
18138             { 0x936B9FCEBB25C996,  -608, -164 },
18139             { 0xDBAC6C247D62A584,  -582, -156 },
18140             { 0xA3AB66580D5FDAF6,  -555, -148 },
18141             { 0xF3E2F893DEC3F126,  -529, -140 },
18142             { 0xB5B5ADA8AAFF80B8,  -502, -132 },
18143             { 0x87625F056C7C4A8B,  -475, -124 },
18144             { 0xC9BCFF6034C13053,  -449, -116 },
18145             { 0x964E858C91BA2655,  -422, -108 },
18146             { 0xDFF9772470297EBD,  -396, -100 },
18147             { 0xA6DFBD9FB8E5B88F,  -369,  -92 },
18148             { 0xF8A95FCF88747D94,  -343,  -84 },
18149             { 0xB94470938FA89BCF,  -316,  -76 },
18150             { 0x8A08F0F8BF0F156B,  -289,  -68 },
18151             { 0xCDB02555653131B6,  -263,  -60 },
18152             { 0x993FE2C6D07B7FAC,  -236,  -52 },
18153             { 0xE45C10C42A2B3B06,  -210,  -44 },
18154             { 0xAA242499697392D3,  -183,  -36 },
18155             { 0xFD87B5F28300CA0E,  -157,  -28 },
18156             { 0xBCE5086492111AEB,  -130,  -20 },
18157             { 0x8CBCCC096F5088CC,  -103,  -12 },
18158             { 0xD1B71758E219652C,   -77,   -4 },
18159             { 0x9C40000000000000,   -50,    4 },
18160             { 0xE8D4A51000000000,   -24,   12 },
18161             { 0xAD78EBC5AC620000,     3,   20 },
18162             { 0x813F3978F8940984,    30,   28 },
18163             { 0xC097CE7BC90715B3,    56,   36 },
18164             { 0x8F7E32CE7BEA5C70,    83,   44 },
18165             { 0xD5D238A4ABE98068,   109,   52 },
18166             { 0x9F4F2726179A2245,   136,   60 },
18167             { 0xED63A231D4C4FB27,   162,   68 },
18168             { 0xB0DE65388CC8ADA8,   189,   76 },
```

```
18169             { 0x83C7088E1AAB65DB,   216,   84 },
18170             { 0xC45D1DF942711D9A,   242,   92 },
18171             { 0x924D692CA61BE758,   269,  100 },
18172             { 0xDA01EE641A708DEA,   295,  108 },
18173             { 0xA26DA3999AEF774A,   322,  116 },
18174             { 0xF209787BB47D6B85,   348,  124 },
18175             { 0xB454E4A179DD1877,   375,  132 },
18176             { 0x865B86925B9BC5C2,   402,  140 },
18177             { 0xC83553C5C8965D3D,   428,  148 },
18178             { 0x952AB45CFA97A0B3,   455,  156 },
18179             { 0xDE469FBD99A05FE3,   481,  164 },
18180             { 0xA59BC234DB398C25,   508,  172 },
18181             { 0xF6C69A72A3989F5C,   534,  180 },
18182             { 0xB7DCBF5354E9BECE,   561,  188 },
18183             { 0x88FCF317F22241E2,   588,  196 },
18184             { 0xCC20CE9BD35C78A5,   614,  204 },
18185             { 0x98165AF37B2153DF,   641,  212 },
18186             { 0xE2A0B5DC971F303A,   667,  220 },
18187             { 0xA8D9D1535CE3B396,   694,  228 },
18188             { 0xFB9B7CD9A4A7443C,   720,  236 },
18189             { 0xBB764C4CA7A44410,   747,  244 },
18190             { 0x8BAB8EEFB6409C1A,   774,  252 },
18191             { 0xD01FEF10A657842C,   800,  260 },
18192             { 0x9B10A4E5E9913129,   827,  268 },
18193             { 0xE7109BFBA19C0C9D,   853,  276 },
18194             { 0xAC2820D9623BF429,   880,  284 },
18195             { 0x80444B5E7AA7CF85,   907,  292 },
18196             { 0xBF21E44003ACDD2D,   933,  300 },
18197             { 0x8E679C2F5E44FF8F,   960,  308 },
18198             { 0xD433179D9C8CB841,   986,  316 },
18199             { 0x9E19DB92B4E31BA9,  1013,  324 },
18200         }
18201     };
18202
18203     // This computation gives exactly the same results for k as
18204     //      k = ceil((kAlpha - e - 1) * 0.30102999566398114)
18205     // for |e| <= 1500, but doesn't require floating-point operations.
18206     // NB: log_10(2) ~= 78913 / 2^18
18207     JSON_ASSERT(e >= -1500);
18208     JSON_ASSERT(e <=  1500);
18209     const int f = kAlpha - e - 1;
18210     const int k = ((f * 78913) / (1 << 18)) + static_cast<int>(f > 0);
18211
18212     const int index = (-kCachedPowersMinDecExp + k + (kCachedPowersDecStep - 1)) /
      kCachedPowersDecStep;
18213     JSON_ASSERT(index >= 0);
18214     JSON_ASSERT(static_cast<std::size_t>(index) < kCachedPowers.size());
18215
18216     const cached_power cached = kCachedPowers[static_cast<std::size_t>(index)];
18217     JSON_ASSERT(kAlpha <= cached.e + e + 64);
18218     JSON_ASSERT(kGamma >= cached.e + e + 64);
18219
18220     return cached;
18221 }
18222
18227 inline int find_largest_pow10(const std::uint32_t n, std::uint32_t& pow10)
18228 {
18229     // LCOV_EXCL_START
18230     if (n >= 1000000000)
18231     {
18232         pow10 = 1000000000;
18233         return 10;
18234     }
18235     // LCOV_EXCL_STOP
18236     if (n >= 100000000)
18237     {
18238         pow10 = 100000000;
18239         return  9;
18240     }
18241     if (n >= 10000000)
18242     {
18243         pow10 = 10000000;
18244         return  8;
18245     }
18246     if (n >= 1000000)
18247     {
18248         pow10 = 1000000;
18249         return  7;
18250     }
18251     if (n >= 100000)
18252     {
18253         pow10 = 100000;
18254         return  6;
18255     }
18256     if (n >= 10000)
18257     {
18258         pow10 = 10000;
```

```
18259         return  5;
18260     }
18261     if (n >= 1000)
18262     {
18263         pow10 = 1000;
18264         return  4;
18265     }
18266     if (n >= 100)
18267     {
18268         pow10 = 100;
18269         return  3;
18270     }
18271     if (n >= 10)
18272     {
18273         pow10 = 10;
18274         return  2;
18275     }
18276
18277     pow10 = 1;
18278     return 1;
18279 }
18280
18281 inline void grisu2_round(char* buf, int len, std::uint64_t dist, std::uint64_t delta,
18282                          std::uint64_t rest, std::uint64_t ten_k)
18283 {
18284     JSON_ASSERT(len >= 1);
18285     JSON_ASSERT(dist <= delta);
18286     JSON_ASSERT(rest <= delta);
18287     JSON_ASSERT(ten_k > 0);
18288
18289     //               <---------------------------- delta ---->
18290     //                                   <---- dist --------->
18291     // --------------[------------------+-------------------]--------------
18292     //               M-                  w                   M+
18293     //
18294     //                                   ten_k
18295     //                                 <------>
18296     //                                        <---- rest ---->
18297     // --------------[------------------+----+--------------]--------------
18298     //                                   w    V
18299     //                                        = buf * 10^k
18300     //
18301     // ten_k represents a unit-in-the-last-place in the decimal representation
18302     // stored in buf.
18303     // Decrement buf by ten_k while this takes buf closer to w.
18304
18305     // The tests are written in this order to avoid overflow in unsigned
18306     // integer arithmetic.
18307
18308     while (rest < dist
18309             && delta - rest >= ten_k
18310             && (rest + ten_k < dist || dist - rest > rest + ten_k - dist))
18311     {
18312         JSON_ASSERT(buf[len - 1] != '0');
18313         buf[len - 1]--;
18314         rest += ten_k;
18315     }
18316 }
18317
18322 inline void grisu2_digit_gen(char* buffer, int& length, int& decimal_exponent,
18323                              diyfp M_minus, diyfp w, diyfp M_plus)
18324 {
18325     static_assert(kAlpha >= -60, "internal error");
18326     static_assert(kGamma <= -32, "internal error");
18327
18328     // Generates the digits (and the exponent) of a decimal floating-point
18329     // number V = buffer * 10^decimal_exponent in the range [M-, M+]. The diyfp's
18330     // w, M- and M+ share the same exponent e, which satisfies alpha <= e <= gamma.
18331     //
18332     //               <---------------------------- delta ---->
18333     //                                   <---- dist --------->
18334     // --------------[------------------+-------------------]--------------
18335     //               M-                  w                   M+
18336     //
18337     // Grisu2 generates the digits of M+ from left to right and stops as soon as
18338     // V is in [M-,M+].
18339
18340     JSON_ASSERT(M_plus.e >= kAlpha);
18341     JSON_ASSERT(M_plus.e <= kGamma);
18342
18343     std::uint64_t delta = diyfp::sub(M_plus, M_minus).f; // (significand of (M+ - M-), implicit
    exponent is e)
18344     std::uint64_t dist  = diyfp::sub(M_plus, w      ).f; // (significand of (M+ - w ), implicit
    exponent is e)
18345
18346     // Split M+ = f * 2^e into two parts p1 and p2 (note: e < 0):
18347     //
```

```
18348     //      M+ = f * 2^e
18349     //         = ((f div 2^-e) * 2^-e + (f mod 2^-e)) * 2^e
18350     //         = ((p1        ) * 2^-e + (p2        )) * 2^e
18351     //         = p1 + p2 * 2^e
18352
18353     const diyfp one(std::uint64_t{1} << -M_plus.e, M_plus.e);
18354
18355     auto p1 = static_cast<std::uint32_t>(M_plus.f >> -one.e); // p1 = f div 2^-e (Since -e >= 32, p1
      fits into a 32-bit int.)
18356     std::uint64_t p2 = M_plus.f & (one.f - 1);                    // p2 = f mod 2^-e
18357
18358     // 1)
18359     //
18360     // Generate the digits of the integral part p1 = d[n-1]...d[1]d[0]
18361
18362     JSON_ASSERT(p1 > 0);
18363
18364     std::uint32_t pow10{};
18365     const int k = find_largest_pow10(p1, pow10);
18366
18367     //      10^(k-1) <= p1 < 10^k, pow10 = 10^(k-1)
18368     //
18369     //      p1 = (p1 div 10^(k-1)) * 10^(k-1) + (p1 mod 10^(k-1))
18370     //         = (d[k-1]         ) * 10^(k-1) + (p1 mod 10^(k-1))
18371     //
18372     //      M+ = p1                                             + p2 * 2^e
18373     //         = d[k-1] * 10^(k-1) + (p1 mod 10^(k-1))          + p2 * 2^e
18374     //         = d[k-1] * 10^(k-1) + ((p1 mod 10^(k-1)) * 2^-e + p2) * 2^e
18375     //         = d[k-1] * 10^(k-1) + (                         rest) * 2^e
18376     //
18377     // Now generate the digits d[n] of p1 from left to right (n = k-1,...,0)
18378     //
18379     //      p1 = d[k-1]...d[n] * 10^n + d[n-1]...d[0]
18380     //
18381     // but stop as soon as
18382     //
18383     //      rest * 2^e = (d[n-1]...d[0] * 2^-e + p2) * 2^e <= delta * 2^e
18384
18385     int n = k;
18386     while (n > 0)
18387     {
18388         // Invariants:
18389         //      M+ = buffer * 10^n + (p1 + p2 * 2^e)    (buffer = 0 for n = k)
18390         //      pow10 = 10^(n-1) <= p1 < 10^n
18391         //
18392         const std::uint32_t d = p1 / pow10;  // d = p1 div 10^(n-1)
18393         const std::uint32_t r = p1 % pow10;  // r = p1 mod 10^(n-1)
18394         //
18395         //      M+ = buffer * 10^n + (d * 10^(n-1) + r) + p2 * 2^e
18396         //         = (buffer * 10 + d) * 10^(n-1) + (r + p2 * 2^e)
18397         //
18398         JSON_ASSERT(d <= 9);
18399         buffer[length++] = static_cast<char>('0' + d); // buffer := buffer * 10 + d
18400         //
18401         //      M+ = buffer * 10^(n-1) + (r + p2 * 2^e)
18402         //
18403         p1 = r;
18404         n--;
18405         //
18406         //      M+ = buffer * 10^n + (p1 + p2 * 2^e)
18407         //      pow10 = 10^n
18408         //
18409
18410         // Now check if enough digits have been generated.
18411         // Compute
18412         //
18413         //      p1 + p2 * 2^e = (p1 * 2^-e + p2) * 2^e = rest * 2^e
18414         //
18415         // Note:
18416         // Since rest and delta share the same exponent e, it suffices to
18417         // compare the significands.
18418         const std::uint64_t rest = (std::uint64_t{p1} << -one.e) + p2;
18419         if (rest <= delta)
18420         {
18421             // V = buffer * 10^n, with M- <= V <= M+.
18422
18423             decimal_exponent += n;
18424
18425             // We may now just stop. But instead, it looks as if the buffer
18426             // could be decremented to bring V closer to w.
18427             //
18428             // pow10 = 10^n is now 1 ulp in the decimal representation V.
18429             // The rounding procedure works with diyfp's with an implicit
18430             // exponent of e.
18431             //
18432             //      10^n = (10^n * 2^-e) * 2^e = ulp * 2^e
18433             //
```

```
18434             const std::uint64_t ten_n = std::uint64_t{pow10} « -one.e;
18435             grisu2_round(buffer, length, dist, delta, rest, ten_n);
18436
18437             return;
18438         }
18439
18440         pow10 /= 10;
18441         //
18442         //      pow10 = 10^(n-1) <= p1 < 10^n
18443         // Invariants restored.
18444     }
18445
18446     // 2)
18447     //
18448     // The digits of the integral part have been generated:
18449     //
18450     //      M+ = d[k-1]...d[1]d[0] + p2 * 2^e
18451     //         = buffer            + p2 * 2^e
18452     //
18453     // Now generate the digits of the fractional part p2 * 2^e.
18454     //
18455     // Note:
18456     // No decimal point is generated: the exponent is adjusted instead.
18457     //
18458     // p2 actually represents the fraction
18459     //
18460     //      p2 * 2^e
18461     //          = p2 / 2^-e
18462     //          = d[-1] / 10^1 + d[-2] / 10^2 + ...
18463     //
18464     // Now generate the digits d[-m] of p1 from left to right (m = 1,2,...)
18465     //
18466     //      p2 * 2^e = d[-1]d[-2]...d[-m] * 10^-m
18467     //                      + 10^-m * (d[-m-1] / 10^1 + d[-m-2] / 10^2 + ...)
18468     //
18469     // using
18470     //
18471     //      10^m * p2 = ((10^m * p2) div 2^-e) * 2^-e + ((10^m * p2) mod 2^-e)
18472     //                = (                   d) * 2^-e + (                   r)
18473     //
18474     // or
18475     //      10^m * p2 * 2^e = d + r * 2^e
18476     //
18477     // i.e.
18478     //
18479     //      M+ = buffer + p2 * 2^e
18480     //         = buffer + 10^-m * (d + r * 2^e)
18481     //         = (buffer * 10^m + d) * 10^-m + 10^-m * r * 2^e
18482     //
18483     // and stop as soon as 10^-m * r * 2^e <= delta * 2^e
18484
18485     JSON_ASSERT(p2 > delta);
18486
18487     int m = 0;
18488     for (;;)
18489     {
18490         // Invariant:
18491         //      M+ = buffer * 10^-m + 10^-m * (d[-m-1] / 10 + d[-m-2] / 10^2 + ...) * 2^e
18492         //         = buffer * 10^-m + 10^-m * (p2                                 ) * 2^e
18493         //         = buffer * 10^-m + 10^-m * (1/10 * (10 * p2)                   ) * 2^e
18494         //         = buffer * 10^-m + 10^-m * (1/10 * ((10*p2 div 2^-e) * 2^-e + (10*p2 mod 2^-e)) *
      2^e
18495         //
18496         JSON_ASSERT(p2 <= (std::numeric_limits<std::uint64_t>::max)() / 10);
18497         p2 *= 10;
18498         const std::uint64_t d = p2 » -one.e;     // d = (10 * p2) div 2^-e
18499         const std::uint64_t r = p2 & (one.f - 1); // r = (10 * p2) mod 2^-e
18500         //
18501         //      M+ = buffer * 10^-m + 10^-m * (1/10 * (d * 2^-e + r) * 2^e
18502         //         = buffer * 10^-m + 10^-m * (1/10 * (d + r * 2^e))
18503         //         = (buffer * 10 + d) * 10^(-m-1) + 10^(-m-1) * r * 2^e
18504         //
18505         JSON_ASSERT(d <= 9);
18506         buffer[length++] = static_cast<char>('0' + d); // buffer := buffer * 10 + d
18507         //
18508         //      M+ = buffer * 10^(-m-1) + 10^(-m-1) * r * 2^e
18509         //
18510         p2 = r;
18511         m++;
18512         //
18513         //      M+ = buffer * 10^-m + 10^-m * p2 * 2^e
18514         // Invariant restored.
18515
18516         // Check if enough digits have been generated.
18517         //
18518         //      10^-m * p2 * 2^e <= delta * 2^e
18519         //              p2 * 2^e <= 10^m * delta * 2^e
```

```
18520         //                    p2 <= 10^m * delta
18521         delta *= 10;
18522         dist  *= 10;
18523         if (p2 <= delta)
18524         {
18525             break;
18526         }
18527     }
18528
18529     // V = buffer * 10^-m, with M- <= V <= M+.
18530
18531     decimal_exponent -= m;
18532
18533     // 1 ulp in the decimal representation is now 10^-m.
18534     // Since delta and dist are now scaled by 10^m, we need to do the
18535     // same with ulp in order to keep the units in sync.
18536     //
18537     //      10^m * 10^-m = 1 = 2^-e * 2^e = ten_m * 2^e
18538     //
18539     const std::uint64_t ten_m = one.f;
18540     grisu2_round(buffer, length, dist, delta, p2, ten_m);
18541
18542     // By construction this algorithm generates the shortest possible decimal
18543     // number (Loitsch, Theorem 6.2) which rounds back to w.
18544     // For an input number of precision p, at least
18545     //
18546     //      N = 1 + ceil(p * log_10(2))
18547     //
18548     // decimal digits are sufficient to identify all binary floating-point
18549     // numbers (Matula, "In-and-Out conversions").
18550     // This implies that the algorithm does not produce more than N decimal
18551     // digits.
18552     //
18553     //      N = 17 for p = 53 (IEEE double precision)
18554     //      N = 9  for p = 24 (IEEE single precision)
18555 }
18556
18562 JSON_HEDLEY_NON_NULL(1)
18563 inline void grisu2(char* buf, int& len, int& decimal_exponent,
18564                    diyfp m_minus, diyfp v, diyfp m_plus)
18565 {
18566     JSON_ASSERT(m_plus.e == m_minus.e);
18567     JSON_ASSERT(m_plus.e == v.e);
18568
18569     //  --------(-----------------------+-----------------------)--------    (A)
18570     //          m-                      v                       m+
18571     //
18572     //  --------------------(-----------+-----------------------)--------    (B)
18573     //                      m-          v                       m+
18574     //
18575     // First scale v (and m- and m+) such that the exponent is in the range
18576     // [alpha, gamma].
18577
18578     const cached_power cached = get_cached_power_for_binary_exponent(m_plus.e);
18579
18580     const diyfp c_minus_k(cached.f, cached.e); // = c ~= 10^-k
18581
18582     // The exponent of the products is = v.e + c_minus_k.e + q and is in the range [alpha,gamma]
18583     const diyfp w       = diyfp::mul(v,       c_minus_k);
18584     const diyfp w_minus = diyfp::mul(m_minus, c_minus_k);
18585     const diyfp w_plus  = diyfp::mul(m_plus,  c_minus_k);
18586
18587     //  ----(---+---)---------------(---+---)---------------(---+---)----
18588     //          w-                      w                       w+
18589     //          = c*m-                  = c*v                   = c*m+
18590     //
18591     // diyfp::mul rounds its result and c_minus_k is approximated too. w, w- and
18592     // w+ are now off by a small amount.
18593     // In fact:
18594     //
18595     //      w - v * 10^k < 1 ulp
18596     //
18597     // To account for this inaccuracy, add resp. subtract 1 ulp.
18598     //
18599     //  --------+---[---------------(---+---)---------------]---+--------
18600     //          w-  M-                  w                   M+  w+
18601     //
18602     // Now any number in [M-, M+] (bounds included) will round to w when input,
18603     // regardless of how the input rounding algorithm breaks ties.
18604     //
18605     // And digit_gen generates the shortest possible such number in [M-, M+].
18606     // Note that this does not mean that Grisu2 always generates the shortest
18607     // possible number in the interval (m-, m+).
18608     const diyfp M_minus(w_minus.f + 1, w_minus.e);
18609     const diyfp M_plus (w_plus.f  - 1, w_plus.e );
18610
18611     decimal_exponent = -cached.k; // = -(-k) = k
```

```
18612
18613     grisu2_digit_gen(buf, len, decimal_exponent, M_minus, w, M_plus);
18614 }
18615
18621 template<typename FloatType>
18622 JSON_HEDLEY_NON_NULL(1)
18623 void grisu2(char* buf, int& len, int& decimal_exponent, FloatType value)
18624 {
18625     static_assert(diyfp::kPrecision >= std::numeric_limits<FloatType>::digits + 3,
18626                   "internal error: not enough precision");
18627
18628     JSON_ASSERT(std::isfinite(value));
18629     JSON_ASSERT(value > 0);
18630
18631     // If the neighbors (and boundaries) of 'value' are always computed for double-precision
18632     // numbers, all float's can be recovered using strtod (and strtof). However, the resulting
18633     // decimal representations are not exactly "short".
18634     //
18635     // The documentation for 'std::to_chars' (https://en.cppreference.com/w/cpp/utility/to_chars)
18636     // says "value is converted to a string as if by std::sprintf in the default ("C") locale"
18637     // and since sprintf promotes floats to doubles, I think this is exactly what 'std::to_chars'
18638     // does.
18639     // On the other hand, the documentation for 'std::to_chars' requires that "parsing the
18640     // representation using the corresponding std::from_chars function recovers value exactly". That
18641     // indicates that single precision floating-point numbers should be recovered using
18642     // 'std::strtof'.
18643     //
18644     // NB: If the neighbors are computed for single-precision numbers, there is a single float
18645     //     (7.0385307e-26f) which can't be recovered using strtod. The resulting double precision
18646     //     value is off by 1 ulp.
18647 #if 0 // NOLINT(readability-avoid-unconditional-preprocessor-if)
18648     const boundaries w = compute_boundaries(static_cast<double>(value));
18649 #else
18650     const boundaries w = compute_boundaries(value);
18651 #endif
18652
18653     grisu2(buf, len, decimal_exponent, w.minus, w.w, w.plus);
18654 }
18655
18661 JSON_HEDLEY_NON_NULL(1)
18662 JSON_HEDLEY_RETURNS_NON_NULL
18663 inline char* append_exponent(char* buf, int e)
18664 {
18665     JSON_ASSERT(e > -1000);
18666     JSON_ASSERT(e <  1000);
18667
18668     if (e < 0)
18669     {
18670         e = -e;
18671         *buf++ = '-';
18672     }
18673     else
18674     {
18675         *buf++ = '+';
18676     }
18677
18678     auto k = static_cast<std::uint32_t>(e);
18679     if (k < 10)
18680     {
18681         // Always print at least two digits in the exponent.
18682         // This is for compatibility with printf("%g").
18683         *buf++ = '0';
18684         *buf++ = static_cast<char>('0' + k);
18685     }
18686     else if (k < 100)
18687     {
18688         *buf++ = static_cast<char>('0' + (k / 10));
18689         k %= 10;
18690         *buf++ = static_cast<char>('0' + k);
18691     }
18692     else
18693     {
18694         *buf++ = static_cast<char>('0' + (k / 100));
18695         k %= 100;
18696         *buf++ = static_cast<char>('0' + (k / 10));
18697         k %= 10;
18698         *buf++ = static_cast<char>('0' + k);
18699     }
18700
18701     return buf;
18702 }
18703
18713 JSON_HEDLEY_NON_NULL(1)
18714 JSON_HEDLEY_RETURNS_NON_NULL
18715 inline char* format_buffer(char* buf, int len, int decimal_exponent,
18716                            int min_exp, int max_exp)
18717 {
```

```
18718     JSON_ASSERT(min_exp < 0);
18719     JSON_ASSERT(max_exp > 0);
18720
18721     const int k = len;
18722     const int n = len + decimal_exponent;
18723
18724     // v = buf * 10^(n-k)
18725     // k is the length of the buffer (number of decimal digits)
18726     // n is the position of the decimal point relative to the start of the buffer.
18727
18728     if (k <= n && n <= max_exp)
18729     {
18730         // digits[000]
18731         // len <= max_exp + 2
18732
18733         std::memset(buf + k, '0', static_cast<size_t>(n) - static_cast<size_t>(k));
18734         // Make it look like a floating-point number (#362, #378)
18735         buf[n + 0] = '.';
18736         buf[n + 1] = '0';
18737         return buf + (static_cast<size_t>(n) + 2);
18738     }
18739
18740     if (0 < n && n <= max_exp)
18741     {
18742         // dig.its
18743         // len <= max_digits10 + 1
18744
18745         JSON_ASSERT(k > n);
18746
18747         std::memmove(buf + (static_cast<size_t>(n) + 1), buf + n, static_cast<size_t>(k) -
      static_cast<size_t>(n));
18748         buf[n] = '.';
18749         return buf + (static_cast<size_t>(k) + 1U);
18750     }
18751
18752     if (min_exp < n && n <= 0)
18753     {
18754         // 0.[000]digits
18755         // len <= 2 + (-min_exp - 1) + max_digits10
18756
18757         std::memmove(buf + (2 + static_cast<size_t>(-n)), buf, static_cast<size_t>(k));
18758         buf[0] = '0';
18759         buf[1] = '.';
18760         std::memset(buf + 2, '0', static_cast<size_t>(-n));
18761         return buf + (2U + static_cast<size_t>(-n) + static_cast<size_t>(k));
18762     }
18763
18764     if (k == 1)
18765     {
18766         // dE+123
18767         // len <= 1 + 5
18768
18769         buf += 1;
18770     }
18771     else
18772     {
18773         // d.igitsE+123
18774         // len <= max_digits10 + 1 + 5
18775
18776         std::memmove(buf + 2, buf + 1, static_cast<size_t>(k) - 1);
18777         buf[1] = '.';
18778         buf += 1 + static_cast<size_t>(k);
18779     }
18780
18781     *buf++ = 'e';
18782     return append_exponent(buf, n - 1);
18783 }
18784
18785 }  // namespace dtoa_impl
18786
18797 template<typename FloatType>
18798 JSON_HEDLEY_NON_NULL(1, 2)
18799 JSON_HEDLEY_RETURNS_NON_NULL
18800 char* to_chars(char* first, const char* last, FloatType value)
18801 {
18802     static_cast<void>(last); // maybe unused - fix warning
18803     JSON_ASSERT(std::isfinite(value));
18804
18805     // Use signbit(value) instead of (value < 0) since signbit works for -0.
18806     if (std::signbit(value))
18807     {
18808         value = -value;
18809         *first++ = '-';
18810     }
18811
18812 #ifdef __GNUC__
18813 #pragma GCC diagnostic push
```

```
18814 #pragma GCC diagnostic ignored "-Wfloat-equal"
18815 #endif
18816     if (value == 0) // +-0
18817     {
18818         *first++ = '0';
18819         // Make it look like a floating-point number (#362, #378)
18820         *first++ = '.';
18821         *first++ = '0';
18822         return first;
18823     }
18824 #ifdef __GNUC__
18825 #pragma GCC diagnostic pop
18826 #endif
18827
18828     JSON_ASSERT(last - first >= std::numeric_limits<FloatType>::max_digits10);
18829
18830     // Compute v = buffer * 10^decimal_exponent.
18831     // The decimal digits are stored in the buffer, which needs to be interpreted
18832     // as an unsigned decimal integer.
18833     // len is the length of the buffer, i.e., the number of decimal digits.
18834     int len = 0;
18835     int decimal_exponent = 0;
18836     dtoa_impl::grisu2(first, len, decimal_exponent, value);
18837
18838     JSON_ASSERT(len <= std::numeric_limits<FloatType>::max_digits10);
18839
18840     // Format the buffer like printf("%.*g", prec, value)
18841     constexpr int kMinExp = -4;
18842     // Use digits10 here to increase compatibility with version 2.
18843     constexpr int kMaxExp = std::numeric_limits<FloatType>::digits10;
18844
18845     JSON_ASSERT(last - first >= kMaxExp + 2);
18846     JSON_ASSERT(last - first >= 2 + (-kMinExp - 1) + std::numeric_limits<FloatType>::max_digits10);
18847     JSON_ASSERT(last - first >= std::numeric_limits<FloatType>::max_digits10 + 6);
18848
18849     return dtoa_impl::format_buffer(first, len, decimal_exponent, kMinExp, kMaxExp);
18850 }
18851
18852 }  // namespace detail
18853 NLOHMANN_JSON_NAMESPACE_END
18854
18855 // #include <nlohmann/detail/exceptions.hpp>
18856
18857 // #include <nlohmann/detail/macro_scope.hpp>
18858
18859 // #include <nlohmann/detail/meta/cpp_future.hpp>
18860
18861 // #include <nlohmann/detail/output/binary_writer.hpp>
18862
18863 // #include <nlohmann/detail/output/output_adapters.hpp>
18864
18865 // #include <nlohmann/detail/string_concat.hpp>
18866
18867 // #include <nlohmann/detail/value_t.hpp>
18868
18869
18870 NLOHMANN_JSON_NAMESPACE_BEGIN
18871 namespace detail
18872 {
18873
18875 // serialization //
18877
18879 enum class error_handler_t
18880 {
18881     strict,
18882     replace,
18883     ignore
18884 };
18885
18886 template<typename BasicJsonType>
18887 class serializer
18888 {
18889     using string_t = typename BasicJsonType::string_t;
18890     using number_float_t = typename BasicJsonType::number_float_t;
18891     using number_integer_t = typename BasicJsonType::number_integer_t;
18892     using number_unsigned_t = typename BasicJsonType::number_unsigned_t;
18893     using binary_char_t = typename BasicJsonType::binary_t::value_type;
18894     static constexpr std::uint8_t UTF8_ACCEPT = 0;
18895     static constexpr std::uint8_t UTF8_REJECT = 1;
18896
18897   public:
18903     serializer(output_adapter_t<char> s, const char ichar,
18904                error_handler_t error_handler_ = error_handler_t::strict)
18905         : o(std::move(s))
18906         , loc(std::localeconv())
18907         , thousands_sep(loc->thousands_sep == nullptr ? '\0' : std::char_traits<char>::to_char_type(*
      (loc->thousands_sep)))
```

```
18908         , decimal_point(loc->decimal_point == nullptr ? '\0' : std::char_traits<char>::to_char_type(*
      (loc->decimal_point)))
18909         , indent_char(ichar)
18910         , indent_string(512, indent_char)
18911         , error_handler(error_handler_)
18912     {}
18913
18914     // deleted because of pointer members
18915     serializer(const serializer&) = delete;
18916     serializer& operator=(const serializer&) = delete;
18917     serializer(serializer&&) = delete;
18918     serializer& operator=(serializer&&) = delete;
18919     ~serializer() = default;
18920
18943     void dump(const BasicJsonType& val,
18944               const bool pretty_print,
18945               const bool ensure_ascii,
18946               const unsigned int indent_step,
18947               const unsigned int current_indent = 0)
18948     {
18949         switch (val.m_data.m_type)
18950         {
18951             case value_t::object:
18952             {
18953                 if (val.m_data.m_value.object->empty())
18954                 {
18955                     o->write_characters("{}", 2);
18956                     return;
18957                 }
18958
18959                 if (pretty_print)
18960                 {
18961                     o->write_characters("{\n", 2);
18962
18963                     // variable to hold indentation for recursive calls
18964                     const auto new_indent = current_indent + indent_step;
18965                     if (JSON_HEDLEY_UNLIKELY(indent_string.size() < new_indent))
18966                     {
18967                         indent_string.resize(indent_string.size() * 2, ' ');
18968                     }
18969
18970                     // first n-1 elements
18971                     auto i = val.m_data.m_value.object->cbegin();
18972                     for (std::size_t cnt = 0; cnt < val.m_data.m_value.object->size() - 1; ++cnt, ++i)
18973                     {
18974                         o->write_characters(indent_string.c_str(), new_indent);
18975                         o->write_character('\"');
18976                         dump_escaped(i->first, ensure_ascii);
18977                         o->write_characters("\": ", 3);
18978                         dump(i->second, true, ensure_ascii, indent_step, new_indent);
18979                         o->write_characters(",\n", 2);
18980                     }
18981
18982                     // last element
18983                     JSON_ASSERT(i != val.m_data.m_value.object->cend());
18984                     JSON_ASSERT(std::next(i) == val.m_data.m_value.object->cend());
18985                     o->write_characters(indent_string.c_str(), new_indent);
18986                     o->write_character('\"');
18987                     dump_escaped(i->first, ensure_ascii);
18988                     o->write_characters("\": ", 3);
18989                     dump(i->second, true, ensure_ascii, indent_step, new_indent);
18990
18991                     o->write_character('\n');
18992                     o->write_characters(indent_string.c_str(), current_indent);
18993                     o->write_character('}');
18994                 }
18995                 else
18996                 {
18997                     o->write_character('{');
18998
18999                     // first n-1 elements
19000                     auto i = val.m_data.m_value.object->cbegin();
19001                     for (std::size_t cnt = 0; cnt < val.m_data.m_value.object->size() - 1; ++cnt, ++i)
19002                     {
19003                         o->write_character('\"');
19004                         dump_escaped(i->first, ensure_ascii);
19005                         o->write_characters("\":", 2);
19006                         dump(i->second, false, ensure_ascii, indent_step, current_indent);
19007                         o->write_character(',');
19008                     }
19009
19010                     // last element
19011                     JSON_ASSERT(i != val.m_data.m_value.object->cend());
19012                     JSON_ASSERT(std::next(i) == val.m_data.m_value.object->cend());
19013                     o->write_character('\"');
19014                     dump_escaped(i->first, ensure_ascii);
19015                     o->write_characters("\":", 2);
```

```
19016                     dump(i->second, false, ensure_ascii, indent_step, current_indent);
19017
19018                     o->write_character('}');
19019                 }
19020
19021                 return;
19022             }
19023
19024             case value_t::array:
19025             {
19026                 if (val.m_data.m_value.array->empty())
19027                 {
19028                     o->write_characters("[]", 2);
19029                     return;
19030                 }
19031
19032                 if (pretty_print)
19033                 {
19034                     o->write_characters("[\n", 2);
19035
19036                     // variable to hold indentation for recursive calls
19037                     const auto new_indent = current_indent + indent_step;
19038                     if (JSON_HEDLEY_UNLIKELY(indent_string.size() < new_indent))
19039                     {
19040                         indent_string.resize(indent_string.size() * 2, ' ');
19041                     }
19042
19043                     // first n-1 elements
19044                     for (auto i = val.m_data.m_value.array->cbegin();
19045                             i != val.m_data.m_value.array->cend() - 1; ++i)
19046                     {
19047                         o->write_characters(indent_string.c_str(), new_indent);
19048                         dump(*i, true, ensure_ascii, indent_step, new_indent);
19049                         o->write_characters(",\n", 2);
19050                     }
19051
19052                     // last element
19053                     JSON_ASSERT(!val.m_data.m_value.array->empty());
19054                     o->write_characters(indent_string.c_str(), new_indent);
19055                     dump(val.m_data.m_value.array->back(), true, ensure_ascii, indent_step,
      new_indent);
19056
19057                     o->write_character('\n');
19058                     o->write_characters(indent_string.c_str(), current_indent);
19059                     o->write_character(']');
19060                 }
19061                 else
19062                 {
19063                     o->write_character('[');
19064
19065                     // first n-1 elements
19066                     for (auto i = val.m_data.m_value.array->cbegin();
19067                             i != val.m_data.m_value.array->cend() - 1; ++i)
19068                     {
19069                         dump(*i, false, ensure_ascii, indent_step, current_indent);
19070                         o->write_character(',');
19071                     }
19072
19073                     // last element
19074                     JSON_ASSERT(!val.m_data.m_value.array->empty());
19075                     dump(val.m_data.m_value.array->back(), false, ensure_ascii, indent_step,
      current_indent);
19076
19077                     o->write_character(']');
19078                 }
19079
19080                 return;
19081             }
19082
19083             case value_t::string:
19084             {
19085                 o->write_character('\"');
19086                 dump_escaped(*val.m_data.m_value.string, ensure_ascii);
19087                 o->write_character('\"');
19088                 return;
19089             }
19090
19091             case value_t::binary:
19092             {
19093                 if (pretty_print)
19094                 {
19095                     o->write_characters("{\n", 2);
19096
19097                     // variable to hold indentation for recursive calls
19098                     const auto new_indent = current_indent + indent_step;
19099                     if (JSON_HEDLEY_UNLIKELY(indent_string.size() < new_indent))
19100                     {
```

```
19101                         indent_string.resize(indent_string.size() * 2, ' ');
19102                     }
19103 
19104                     o->write_characters(indent_string.c_str(), new_indent);
19105 
19106                     o->write_characters("\"bytes\": [", 10);
19107 
19108                     if (!val.m_data.m_value.binary->empty())
19109                     {
19110                         for (auto i = val.m_data.m_value.binary->cbegin();
19111                                 i != val.m_data.m_value.binary->cend() - 1; ++i)
19112                         {
19113                             dump_integer(*i);
19114                             o->write_characters(", ", 2);
19115                         }
19116                         dump_integer(val.m_data.m_value.binary->back());
19117                     }
19118 
19119                     o->write_characters("],\n", 3);
19120                     o->write_characters(indent_string.c_str(), new_indent);
19121 
19122                     o->write_characters("\"subtype\": ", 11);
19123                     if (val.m_data.m_value.binary->has_subtype())
19124                     {
19125                         dump_integer(val.m_data.m_value.binary->subtype());
19126                     }
19127                     else
19128                     {
19129                         o->write_characters("null", 4);
19130                     }
19131                     o->write_character('\n');
19132                     o->write_characters(indent_string.c_str(), current_indent);
19133                     o->write_character('}');
19134                 }
19135                 else
19136                 {
19137                     o->write_characters("{\"bytes\":[", 10);
19138 
19139                     if (!val.m_data.m_value.binary->empty())
19140                     {
19141                         for (auto i = val.m_data.m_value.binary->cbegin();
19142                                 i != val.m_data.m_value.binary->cend() - 1; ++i)
19143                         {
19144                             dump_integer(*i);
19145                             o->write_character(',');
19146                         }
19147                         dump_integer(val.m_data.m_value.binary->back());
19148                     }
19149 
19150                     o->write_characters("],\"subtype\":", 12);
19151                     if (val.m_data.m_value.binary->has_subtype())
19152                     {
19153                         dump_integer(val.m_data.m_value.binary->subtype());
19154                         o->write_character('}');
19155                     }
19156                     else
19157                     {
19158                         o->write_characters("null}", 5);
19159                     }
19160                 }
19161                 return;
19162             }
19163 
19164             case value_t::boolean:
19165             {
19166                 if (val.m_data.m_value.boolean)
19167                 {
19168                     o->write_characters("true", 4);
19169                 }
19170                 else
19171                 {
19172                     o->write_characters("false", 5);
19173                 }
19174                 return;
19175             }
19176 
19177             case value_t::number_integer:
19178             {
19179                 dump_integer(val.m_data.m_value.number_integer);
19180                 return;
19181             }
19182 
19183             case value_t::number_unsigned:
19184             {
19185                 dump_integer(val.m_data.m_value.number_unsigned);
19186                 return;
19187             }
```

```
19188
19189             case value_t::number_float:
19190             {
19191                 dump_float(val.m_data.m_value.number_float);
19192                 return;
19193             }
19194
19195             case value_t::discarded:
19196             {
19197                 o->write_characters("<discarded>", 11);
19198                 return;
19199             }
19200
19201             case value_t::null:
19202             {
19203                 o->write_characters("null", 4);
19204                 return;
19205             }
19206
19207             default:            // LCOV_EXCL_LINE
19208                 JSON_ASSERT(false); // NOLINT(cert-dcl03-c,hicpp-static-assert,misc-static-assert)
    LCOV_EXCL_LINE
19209         }
19210     }
19211
19212   JSON_PRIVATE_UNLESS_TESTED:
19227     void dump_escaped(const string_t& s, const bool ensure_ascii)
19228     {
19229         std::uint32_t codepoint{};
19230         std::uint8_t state = UTF8_ACCEPT;
19231         std::size_t bytes = 0;  // number of bytes written to string_buffer
19232
19233         // number of bytes written at the point of the last valid byte
19234         std::size_t bytes_after_last_accept = 0;
19235         std::size_t undumped_chars = 0;
19236
19237         for (std::size_t i = 0; i < s.size(); ++i)
19238         {
19239             const auto byte = static_cast<std::uint8_t>(s[i]);
19240
19241             switch (decode(state, codepoint, byte))
19242             {
19243                 case UTF8_ACCEPT:  // decode found a new code point
19244                 {
19245                     switch (codepoint)
19246                     {
19247                         case 0x08: // backspace
19248                         {
19249                             string_buffer[bytes++] = '\\';
19250                             string_buffer[bytes++] = 'b';
19251                             break;
19252                         }
19253
19254                         case 0x09: // horizontal tab
19255                         {
19256                             string_buffer[bytes++] = '\\';
19257                             string_buffer[bytes++] = 't';
19258                             break;
19259                         }
19260
19261                         case 0x0A: // newline
19262                         {
19263                             string_buffer[bytes++] = '\\';
19264                             string_buffer[bytes++] = 'n';
19265                             break;
19266                         }
19267
19268                         case 0x0C: // formfeed
19269                         {
19270                             string_buffer[bytes++] = '\\';
19271                             string_buffer[bytes++] = 'f';
19272                             break;
19273                         }
19274
19275                         case 0x0D: // carriage return
19276                         {
19277                             string_buffer[bytes++] = '\\';
19278                             string_buffer[bytes++] = 'r';
19279                             break;
19280                         }
19281
19282                         case 0x22: // quotation mark
19283                         {
19284                             string_buffer[bytes++] = '\\';
19285                             string_buffer[bytes++] = '\"';
19286                             break;
19287                         }
```

```
19288
19289                         case 0x5C: // reverse solidus
19290                         {
19291                             string_buffer[bytes++] = '\\';
19292                             string_buffer[bytes++] = '\\';
19293                             break;
19294                         }
19295
19296                         default:
19297                         {
19298                             // escape control characters (0x00..0x1F) or, if
19299                             // ensure_ascii parameter is used, non-ASCII characters
19300                             if ((codepoint <= 0x1F) || (ensure_ascii && (codepoint >= 0x7F)))
19301                             {
19302                                 if (codepoint <= 0xFFFF)
19303                                 {
19304                                     // NOLINTNEXTLINE(cppcoreguidelines-pro-type-vararg,hicpp-vararg)
19305                                     static_cast<void>((std::snprintf)(string_buffer.data() + bytes, 7,
      "\\u%04x",
19306
      static_cast<std::uint16_t>(codepoint)));
19307                                     bytes += 6;
19308                                 }
19309                                 else
19310                                 {
19311                                     // NOLINTNEXTLINE(cppcoreguidelines-pro-type-vararg,hicpp-vararg)
19312                                     static_cast<void>((std::snprintf)(string_buffer.data() + bytes,
      13, "\\u%04x\\u%04x",
19313
      static_cast<std::uint16_t>(0xD7C0u + (codepoint » 10u)),
19314
      static_cast<std::uint16_t>(0xDC00u + (codepoint & 0x3FFu))));
19315                                     bytes += 12;
19316                                 }
19317                             }
19318                             else
19319                             {
19320                                 // copy byte to buffer (all previous bytes
19321                                 // been copied have in default case above)
19322                                 string_buffer[bytes++] = s[i];
19323                             }
19324                             break;
19325                         }
19326                     }
19327
19328                     // write buffer and reset index; there must be 13 bytes
19329                     // left, as this is the maximal number of bytes to be
19330                     // written ("\uxxxx\uxxxx\0") for one code point
19331                     if (string_buffer.size() - bytes < 13)
19332                     {
19333                         o->write_characters(string_buffer.data(), bytes);
19334                         bytes = 0;
19335                     }
19336
19337                     // remember the byte position of this accept
19338                     bytes_after_last_accept = bytes;
19339                     undumped_chars = 0;
19340                     break;
19341                 }
19342
19343                 case UTF8_REJECT:  // decode found invalid UTF-8 byte
19344                 {
19345                     switch (error_handler)
19346                     {
19347                         case error_handler_t::strict:
19348                         {
19349                             JSON_THROW(type_error::create(316, concat("invalid UTF-8 byte at index ",
      std::to_string(i), ": 0x", hex_bytes(byte | 0)), nullptr));
19350                         }
19351
19352                         case error_handler_t::ignore:
19353                         case error_handler_t::replace:
19354                         {
19355                             // in case we saw this character the first time, we
19356                             // would like to read it again, because the byte
19357                             // may be OK for itself, but just not OK for the
19358                             // previous sequence
19359                             if (undumped_chars > 0)
19360                             {
19361                                 --i;
19362                             }
19363
19364                             // reset length buffer to the last accepted index;
19365                             // thus removing/ignoring the invalid characters
19366                             bytes = bytes_after_last_accept;
19367
19368                             if (error_handler == error_handler_t::replace)
```

```
19369                             {
19370                                 // add a replacement character
19371                                 if (ensure_ascii)
19372                                 {
19373                                     string_buffer[bytes++] = '\\';
19374                                     string_buffer[bytes++] = 'u';
19375                                     string_buffer[bytes++] = 'f';
19376                                     string_buffer[bytes++] = 'f';
19377                                     string_buffer[bytes++] = 'f';
19378                                     string_buffer[bytes++] = 'd';
19379                                 }
19380                                 else
19381                                 {
19382                                     string_buffer[bytes++] = detail::binary_writer<BasicJsonType,
    char>::to_char_type('\xEF');
19383                                     string_buffer[bytes++] = detail::binary_writer<BasicJsonType,
    char>::to_char_type('\xBF');
19384                                     string_buffer[bytes++] = detail::binary_writer<BasicJsonType,
    char>::to_char_type('\xBD');
19385                                 }
19386
19387                                 // write buffer and reset index; there must be 13 bytes
19388                                 // left, as this is the maximal number of bytes to be
19389                                 // written ("\uxxxx\uxxxx\0") for one code point
19390                                 if (string_buffer.size() - bytes < 13)
19391                                 {
19392                                     o->write_characters(string_buffer.data(), bytes);
19393                                     bytes = 0;
19394                                 }
19395
19396                                 bytes_after_last_accept = bytes;
19397                             }
19398
19399                             undumped_chars = 0;
19400
19401                             // continue processing the string
19402                             state = UTF8_ACCEPT;
19403                             break;
19404                         }
19405
19406                         default:            // LCOV_EXCL_LINE
19407                             JSON_ASSERT(false); //
    NOLINT(cert-dcl03-c,hicpp-static-assert,misc-static-assert) LCOV_EXCL_LINE
19408                     }
19409                     break;
19410                 }
19411
19412                 default:  // decode found yet incomplete multibyte code point
19413                 {
19414                     if (!ensure_ascii)
19415                     {
19416                         // code point will not be escaped - copy byte to buffer
19417                         string_buffer[bytes++] = s[i];
19418                     }
19419                     ++undumped_chars;
19420                     break;
19421                 }
19422             }
19423         }
19424
19425         // we finished processing the string
19426         if (JSON_HEDLEY_LIKELY(state == UTF8_ACCEPT))
19427         {
19428             // write buffer
19429             if (bytes > 0)
19430             {
19431                 o->write_characters(string_buffer.data(), bytes);
19432             }
19433         }
19434         else
19435         {
19436             // we finish reading, but do not accept: string was incomplete
19437             switch (error_handler)
19438             {
19439                 case error_handler_t::strict:
19440                 {
19441                     JSON_THROW(type_error::create(316, concat("incomplete UTF-8 string; last byte:
    0x", hex_bytes(static_cast<std::uint8_t>(s.back() | 0))), nullptr));
19442                 }
19443
19444                 case error_handler_t::ignore:
19445                 {
19446                     // write all accepted bytes
19447                     o->write_characters(string_buffer.data(), bytes_after_last_accept);
19448                     break;
19449                 }
19450
```

```
19451                 case error_handler_t::replace:
19452                 {
19453                     // write all accepted bytes
19454                     o->write_characters(string_buffer.data(), bytes_after_last_accept);
19455                     // add a replacement character
19456                     if (ensure_ascii)
19457                     {
19458                         o->write_characters("\\ufffd", 6);
19459                     }
19460                     else
19461                     {
19462                         o->write_characters("\xEF\xBF\xBD", 3);
19463                     }
19464                     break;
19465                 }
19466 
19467                 default:            // LCOV_EXCL_LINE
19468                     JSON_ASSERT(false); // NOLINT(cert-dcl03-c,hicpp-static-assert,misc-static-assert)
      LCOV_EXCL_LINE
19469             }
19470         }
19471     }
19472 
19473   private:
19482     unsigned int count_digits(number_unsigned_t x) noexcept
19483     {
19484         unsigned int n_digits = 1;
19485         for (;;)
19486         {
19487             if (x < 10)
19488             {
19489                 return n_digits;
19490             }
19491             if (x < 100)
19492             {
19493                 return n_digits + 1;
19494             }
19495             if (x < 1000)
19496             {
19497                 return n_digits + 2;
19498             }
19499             if (x < 10000)
19500             {
19501                 return n_digits + 3;
19502             }
19503             x = x / 10000u;
19504             n_digits += 4;
19505         }
19506     }
19507 
19513     static std::string hex_bytes(std::uint8_t byte)
19514     {
19515         std::string result = "FF";
19516         constexpr const char* nibble_to_hex = "0123456789ABCDEF";
19517         result[0] = nibble_to_hex[byte / 16];
19518         result[1] = nibble_to_hex[byte % 16];
19519         return result;
19520     }
19521 
19522     // templates to avoid warnings about useless casts
19523     template <typename NumberType, enable_if_t<std::is_signed<NumberType>::value, int> = 0>
19524     bool is_negative_number(NumberType x)
19525     {
19526         return x < 0;
19527     }
19528 
19529     template < typename NumberType, enable_if_t <std::is_unsigned<NumberType>::value, int > = 0 >
19530     bool is_negative_number(NumberType /*unused*/)
19531     {
19532         return false;
19533     }
19534 
19544     template < typename NumberType, detail::enable_if_t <
19545                     std::is_integral<NumberType>::value ||
19546                     std::is_same<NumberType, number_unsigned_t>::value ||
19547                     std::is_same<NumberType, number_integer_t>::value ||
19548                     std::is_same<NumberType, binary_char_t>::value,
19549                     int > = 0 >
19550     void dump_integer(NumberType x)
19551     {
19552         static constexpr std::array<std::array<char, 2>, 100> digits_to_99
19553         {
19554             {
19555                 {{'0', '0'}}, {{'0', '1'}}, {{'0', '2'}}, {{'0', '3'}}, {{'0', '4'}}, {{'0', '5'}},
      {{'0', '6'}}, {{'0', '7'}}, {{'0', '8'}}, {{'0', '9'}},
19556                 {{'1', '0'}}, {{'1', '1'}}, {{'1', '2'}}, {{'1', '3'}}, {{'1', '4'}}, {{'1', '5'}},
      {{'1', '6'}}, {{'1', '7'}}, {{'1', '8'}}, {{'1', '9'}},
```

```
19557                 {{'2', '0'}}, {{'2', '1'}}, {{'2', '2'}}, {{'2', '3'}}, {{'2', '4'}}, {{'2', '5'}},
      {{'2', '6'}}, {{'2', '7'}}, {{'2', '8'}}, {{'2', '9'}},
19558                 {{'3', '0'}}, {{'3', '1'}}, {{'3', '2'}}, {{'3', '3'}}, {{'3', '4'}}, {{'3', '5'}},
      {{'3', '6'}}, {{'3', '7'}}, {{'3', '8'}}, {{'3', '9'}},
19559                 {{'4', '0'}}, {{'4', '1'}}, {{'4', '2'}}, {{'4', '3'}}, {{'4', '4'}}, {{'4', '5'}},
      {{'4', '6'}}, {{'4', '7'}}, {{'4', '8'}}, {{'4', '9'}},
19560                 {{'5', '0'}}, {{'5', '1'}}, {{'5', '2'}}, {{'5', '3'}}, {{'5', '4'}}, {{'5', '5'}},
      {{'5', '6'}}, {{'5', '7'}}, {{'5', '8'}}, {{'5', '9'}},
19561                 {{'6', '0'}}, {{'6', '1'}}, {{'6', '2'}}, {{'6', '3'}}, {{'6', '4'}}, {{'6', '5'}},
      {{'6', '6'}}, {{'6', '7'}}, {{'6', '8'}}, {{'6', '9'}},
19562                 {{'7', '0'}}, {{'7', '1'}}, {{'7', '2'}}, {{'7', '3'}}, {{'7', '4'}}, {{'7', '5'}},
      {{'7', '6'}}, {{'7', '7'}}, {{'7', '8'}}, {{'7', '9'}},
19563                 {{'8', '0'}}, {{'8', '1'}}, {{'8', '2'}}, {{'8', '3'}}, {{'8', '4'}}, {{'8', '5'}},
      {{'8', '6'}}, {{'8', '7'}}, {{'8', '8'}}, {{'8', '9'}},
19564                 {{'9', '0'}}, {{'9', '1'}}, {{'9', '2'}}, {{'9', '3'}}, {{'9', '4'}}, {{'9', '5'}},
      {{'9', '6'}}, {{'9', '7'}}, {{'9', '8'}}, {{'9', '9'}},
19565             }
19566         };
19567
19568         // special case for "0"
19569         if (x == 0)
19570         {
19571             o->write_character('0');
19572             return;
19573         }
19574
19575         // use a pointer to fill the buffer
19576         auto buffer_ptr = number_buffer.begin(); //
      NOLINT(llvm-qualified-auto,readability-qualified-auto,cppcoreguidelines-pro-type-vararg,hicpp-vararg)
19577
19578         number_unsigned_t abs_value;
19579
19580         unsigned int n_chars{};
19581
19582         if (is_negative_number(x))
19583         {
19584             *buffer_ptr = '-';
19585             abs_value = remove_sign(static_cast<number_integer_t>(x));
19586
19587             // account one more byte for the minus sign
19588             n_chars = 1 + count_digits(abs_value);
19589         }
19590         else
19591         {
19592             abs_value = static_cast<number_unsigned_t>(x);
19593             n_chars = count_digits(abs_value);
19594         }
19595
19596         // spare 1 byte for '\0'
19597         JSON_ASSERT(n_chars < number_buffer.size() - 1);
19598
19599         // jump to the end to generate the string from backward,
19600         // so we later avoid reversing the result
19601         buffer_ptr += static_cast<typename decltype(number_buffer)::difference_type>(n_chars);
19602
19603         // Fast int2ascii implementation inspired by "Fastware" talk by Andrei Alexandrescu
19604         // See: https://www.youtube.com/watch?v=o4-CwDo2zpg
19605         while (abs_value >= 100)
19606         {
19607             const auto digits_index = static_cast<unsigned>((abs_value % 100));
19608             abs_value /= 100;
19609             *(--buffer_ptr) = digits_to_99[digits_index][1];
19610             *(--buffer_ptr) = digits_to_99[digits_index][0];
19611         }
19612
19613         if (abs_value >= 10)
19614         {
19615             const auto digits_index = static_cast<unsigned>(abs_value);
19616             *(--buffer_ptr) = digits_to_99[digits_index][1];
19617             *(--buffer_ptr) = digits_to_99[digits_index][0];
19618         }
19619         else
19620         {
19621             *(--buffer_ptr) = static_cast<char>('0' + abs_value);
19622         }
19623
19624         o->write_characters(number_buffer.data(), n_chars);
19625     }
19626
19635     void dump_float(number_float_t x)
19636     {
19637         // NaN / inf
19638         if (!std::isfinite(x))
19639         {
19640             o->write_characters("null", 4);
19641             return;
19642         }
```

```
19643
19644         // If number_float_t is an IEEE-754 single or double precision number,
19645         // use the Grisu2 algorithm to produce short numbers which are
19646         // guaranteed to round-trip, using strtof and strtod, resp.
19647         //
19648         // NB: The test below works if <long double> == <double>.
19649         static constexpr bool is_ieee_single_or_double
19650             = (std::numeric_limits<number_float_t>::is_iec559 &&
      std::numeric_limits<number_float_t>::digits == 24 && std::numeric_limits<number_float_t>::max_exponent
      == 128) ||
19651                 (std::numeric_limits<number_float_t>::is_iec559 &&
      std::numeric_limits<number_float_t>::digits == 53 && std::numeric_limits<number_float_t>::max_exponent
      == 1024);
19652
19653         dump_float(x, std::integral_constant<bool, is_ieee_single_or_double>());
19654     }
19655
19656     void dump_float(number_float_t x, std::true_type /*is_ieee_single_or_double*/)
19657     {
19658         auto* begin = number_buffer.data();
19659         auto* end = ::nlohmann::detail::to_chars(begin, begin + number_buffer.size(), x);
19660
19661         o->write_characters(begin, static_cast<size_t>(end - begin));
19662     }
19663
19664     void dump_float(number_float_t x, std::false_type /*is_ieee_single_or_double*/)
19665     {
19666         // get the number of digits for a float -> text -> float round-trip
19667         static constexpr auto d = std::numeric_limits<number_float_t>::max_digits10;
19668
19669         // the actual conversion
19670         // NOLINTNEXTLINE(cppcoreguidelines-pro-type-vararg,hicpp-vararg)
19671         std::ptrdiff_t len = (std::snprintf)(number_buffer.data(), number_buffer.size(), "%.*g", d,
      x);
19672
19673         // negative value indicates an error
19674         JSON_ASSERT(len > 0);
19675         // check if the buffer was large enough
19676         JSON_ASSERT(static_cast<std::size_t>(len) < number_buffer.size());
19677
19678         // erase thousands separators
19679         if (thousands_sep != '\0')
19680         {
19681             // NOLINTNEXTLINE(readability-qualified-auto,llvm-qualified-auto): std::remove returns an
      iterator, see https://github.com/nlohmann/json/issues/3081
19682             const auto end = std::remove(number_buffer.begin(), number_buffer.begin() + len,
      thousands_sep);
19683             std::fill(end, number_buffer.end(), '\0');
19684             JSON_ASSERT((end - number_buffer.begin()) <= len);
19685             len = (end - number_buffer.begin());
19686         }
19687
19688         // convert decimal point to '.'
19689         if (decimal_point != '\0' && decimal_point != '.')
19690         {
19691             // NOLINTNEXTLINE(readability-qualified-auto,llvm-qualified-auto): std::find returns an
      iterator, see https://github.com/nlohmann/json/issues/3081
19692             const auto dec_pos = std::find(number_buffer.begin(), number_buffer.end(), decimal_point);
19693             if (dec_pos != number_buffer.end())
19694             {
19695                 *dec_pos = '.';
19696             }
19697         }
19698
19699         o->write_characters(number_buffer.data(), static_cast<std::size_t>(len));
19700
19701         // determine if we need to append ".0"
19702         const bool value_is_int_like =
19703             std::none_of(number_buffer.begin(), number_buffer.begin() + len + 1,
19704                          [](char c)
19705         {
19706             return c == '.' || c == 'e';
19707         });
19708
19709         if (value_is_int_like)
19710         {
19711             o->write_characters(".0", 2);
19712         }
19713     }
19714
19736     static std::uint8_t decode(std::uint8_t& state, std::uint32_t& codep, const std::uint8_t byte)
      noexcept
19737     {
19738         static const std::array<std::uint8_t, 400> utf8d =
19739         {
19740             {
19741                 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
    0, 0, 0, // 00..1F
19742                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, // 20..3F
19743                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, // 40..5F
19744                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, // 60..7F
19745                1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
    9, 9, 9, // 80..9F
19746                7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
    7, 7, 7, // A0..BF
19747                8, 8, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
    2, 2, 2, // C0..DF
19748                0xA, 0x3, 0x3, 0x3, 0x3, 0x3, 0x3, 0x3, 0x3, 0x3, 0x3, 0x3, 0x3, 0x4, 0x3, 0x3, //
    E0..EF
19749                0xB, 0x6, 0x6, 0x6, 0x5, 0x8, 0x8, 0x8, 0x8, 0x8, 0x8, 0x8, 0x8, 0x8, 0x8, 0x8, //
    F0..FF
19750                0x0, 0x1, 0x2, 0x3, 0x5, 0x8, 0x7, 0x1, 0x1, 0x1, 0x4, 0x6, 0x1, 0x1, 0x1, 0x1, //
    s0..s0
19751                1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1,
    1, 1, 1, // s1..s2
19752                1, 2, 1, 1, 1, 1, 1, 2, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1,
    1, 1, 1, // s3..s4
19753                1, 2, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 1, 3, 1, 1, 1,
    1, 1, 1, // s5..s6
19754                1, 3, 1, 1, 1, 1, 1, 3, 1, 3, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1 // s7..s8
19755            }
19756        };
19757
19758        JSON_ASSERT(static_cast<std::size_t>(byte) < utf8d.size());
19759        const std::uint8_t type = utf8d[byte];
19760
19761        codep = (state != UTF8_ACCEPT)
19762                ? (byte & 0x3fu) | (codep << 6u)
19763                : (0xFFu >> type) & (byte);
19764
19765        const std::size_t index = 256u + (static_cast<size_t>(state) * 16u) +
    static_cast<size_t>(type);
19766        JSON_ASSERT(index < utf8d.size());
19767        state = utf8d[index];
19768        return state;
19769    }
19770
19771    /*
19772     * Overload to make the compiler happy while it is instantiating
19773     * dump_integer for number_unsigned_t.
19774     * Must never be called.
19775     */
19776    number_unsigned_t remove_sign(number_unsigned_t x)
19777    {
19778        JSON_ASSERT(false); // NOLINT(cert-dcl03-c,hicpp-static-assert,misc-static-assert)
    LCOV_EXCL_LINE
19779        return x; // LCOV_EXCL_LINE
19780    }
19781
19782    /*
19783     * Helper function for dump_integer
19784     *
19785     * This function takes a negative signed integer and returns its absolute
19786     * value as an unsigned integer. The plus/minus shuffling is necessary as we
19787     * cannot directly remove the sign of an arbitrary signed integer as the
19788     * absolute values of INT_MIN and INT_MAX are usually not the same. See
19789     * #1708 for details.
19790     */
19791    number_unsigned_t remove_sign(number_integer_t x) noexcept
19792    {
19793        JSON_ASSERT(x < 0 && x < (std::numeric_limits<number_integer_t>::max())); //
    NOLINT(misc-redundant-expression)
19794        return static_cast<number_unsigned_t>(-(x + 1)) + 1;
19795    }
19796
19797  private:
19799    output_adapter_t<char> o = nullptr;
19800
19802    std::array<char, 64> number_buffer{{}};
19803
19805    const std::lconv* loc = nullptr;
19807    const char thousands_sep = '\0';
19809    const char decimal_point = '\0';
19810
19812    std::array<char, 512> string_buffer{{}};
19813
19815    const char indent_char;
19817    string_t indent_string;
19818
19820    const error_handler_t error_handler;
```

```
19821 };
19822
19823 }  // namespace detail
19824 NLOHMANN_JSON_NAMESPACE_END
19825
19826 // #include <nlohmann/detail/value_t.hpp>
19827
19828 // #include <nlohmann/json_fwd.hpp>
19829
19830 // #include <nlohmann/ordered_map.hpp>
19831 //     __ _____ _____ _____
19832 //  __|  |   __|     |   | |  JSON for Modern C++
19833 // |  |  |__   |  |  | | | |  version 3.12.0
19834 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
19835 //
19836 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
19837 // SPDX-License-Identifier: MIT
19838
19839
19840
19841 #include <functional> // equal_to, less
19842 #include <initializer_list> // initializer_list
19843 #include <iterator> // input_iterator_tag, iterator_traits
19844 #include <memory> // allocator
19845 #include <stdexcept> // for out_of_range
19846 #include <type_traits> // enable_if, is_convertible
19847 #include <utility> // pair
19848 #include <vector> // vector
19849
19850 // #include <nlohmann/detail/macro_scope.hpp>
19851
19852 // #include <nlohmann/detail/meta/type_traits.hpp>
19853
19854
19855 NLOHMANN_JSON_NAMESPACE_BEGIN
19856
19859 template <class Key, class T, class IgnoredLess = std::less<Key>,
19860           class Allocator = std::allocator<std::pair<const Key, T>>
19861               struct ordered_map : std::vector<std::pair<const Key, T>, Allocator>
19862 {
19863     using key_type = Key;
19864     using mapped_type = T;
19865     using Container = std::vector<std::pair<const Key, T>, Allocator>;
19866     using iterator = typename Container::iterator;
19867     using const_iterator = typename Container::const_iterator;
19868     using size_type = typename Container::size_type;
19869     using value_type = typename Container::value_type;
19870 #ifdef JSON_HAS_CPP_14
19871     using key_compare = std::equal_to<>;
19872 #else
19873     using key_compare = std::equal_to<Key>;
19874 #endif
19875
19876     // Explicit constructors instead of `using Container::Container`
19877     // otherwise older compilers choke on it (GCC <= 5.5, xcode <= 9.4)
19878     ordered_map() noexcept(noexcept(Container())) : Container{} {}
19879     explicit ordered_map(const Allocator& alloc) noexcept(noexcept(Container(alloc))) :
      Container{alloc} {}
19880     template <class It>
19881     ordered_map(It first, It last, const Allocator& alloc = Allocator())
19882         : Container{first, last, alloc} {}
19883     ordered_map(std::initializer_list<value_type> init, const Allocator& alloc = Allocator() )
19884         : Container{init, alloc} {}
19885
19886     std::pair<iterator, bool> emplace(const key_type& key, T&& t)
19887     {
19888         for (auto it = this->begin(); it != this->end(); ++it)
19889         {
19890             if (m_compare(it->first, key))
19891             {
19892                 return {it, false};
19893             }
19894         }
19895         Container::emplace_back(key, std::forward<T>(t));
19896         return {std::prev(this->end()), true};
19897     }
19898
19899     template<class KeyType, detail::enable_if_t<
19900                  detail::is_usable_as_key_type<key_compare, key_type, KeyType>::value, int> = 0>
19901     std::pair<iterator, bool> emplace(KeyType && key, T && t)
19902     {
19903         for (auto it = this->begin(); it != this->end(); ++it)
19904         {
19905             if (m_compare(it->first, key))
19906             {
19907                 return {it, false};
19908             }
```

```
19909         }
19910         Container::emplace_back(std::forward<KeyType>(key), std::forward<T>(t));
19911         return {std::prev(this->end()), true};
19912     }
19913
19914     T& operator[](const key_type& key)
19915     {
19916         return emplace(key, T{}).first->second;
19917     }
19918
19919     template<class KeyType, detail::enable_if_t<
19920                 detail::is_usable_as_key_type<key_compare, key_type, KeyType>::value, int> = 0>
19921     T & operator[](KeyType && key)
19922     {
19923         return emplace(std::forward<KeyType>(key), T{}).first->second;
19924     }
19925
19926     const T& operator[](const key_type& key) const
19927     {
19928         return at(key);
19929     }
19930
19931     template<class KeyType, detail::enable_if_t<
19932                 detail::is_usable_as_key_type<key_compare, key_type, KeyType>::value, int> = 0>
19933     const T & operator[](KeyType && key) const
19934     {
19935         return at(std::forward<KeyType>(key));
19936     }
19937
19938     T& at(const key_type& key)
19939     {
19940         for (auto it = this->begin(); it != this->end(); ++it)
19941         {
19942             if (m_compare(it->first, key))
19943             {
19944                 return it->second;
19945             }
19946         }
19947
19948         JSON_THROW(std::out_of_range("key not found"));
19949     }
19950
19951     template<class KeyType, detail::enable_if_t<
19952                 detail::is_usable_as_key_type<key_compare, key_type, KeyType>::value, int> = 0>
19953     T & at(KeyType && key) // NOLINT(cppcoreguidelines-missing-std-forward)
19954     {
19955         for (auto it = this->begin(); it != this->end(); ++it)
19956         {
19957             if (m_compare(it->first, key))
19958             {
19959                 return it->second;
19960             }
19961         }
19962
19963         JSON_THROW(std::out_of_range("key not found"));
19964     }
19965
19966     const T& at(const key_type& key) const
19967     {
19968         for (auto it = this->begin(); it != this->end(); ++it)
19969         {
19970             if (m_compare(it->first, key))
19971             {
19972                 return it->second;
19973             }
19974         }
19975
19976         JSON_THROW(std::out_of_range("key not found"));
19977     }
19978
19979     template<class KeyType, detail::enable_if_t<
19980                 detail::is_usable_as_key_type<key_compare, key_type, KeyType>::value, int> = 0>
19981     const T & at(KeyType && key) const // NOLINT(cppcoreguidelines-missing-std-forward)
19982     {
19983         for (auto it = this->begin(); it != this->end(); ++it)
19984         {
19985             if (m_compare(it->first, key))
19986             {
19987                 return it->second;
19988             }
19989         }
19990
19991         JSON_THROW(std::out_of_range("key not found"));
19992     }
19993
19994     size_type erase(const key_type& key)
19995     {
```

```
19996         for (auto it = this->begin(); it != this->end(); ++it)
19997         {
19998             if (m_compare(it->first, key))
19999             {
20000                 // Since we cannot move const Keys, re-construct them in place
20001                 for (auto next = it; ++next != this->end(); ++it)
20002                 {
20003                     it->~value_type(); // Destroy but keep allocation
20004                     new (&*it) value_type{std::move(*next)};
20005                 }
20006                 Container::pop_back();
20007                 return 1;
20008             }
20009         }
20010         return 0;
20011     }
20012
20013     template<class KeyType, detail::enable_if_t<
20014                  detail::is_usable_as_key_type<key_compare, key_type, KeyType>::value, int> = 0>
20015     size_type erase(KeyType && key) // NOLINT(cppcoreguidelines-missing-std-forward)
20016     {
20017         for (auto it = this->begin(); it != this->end(); ++it)
20018         {
20019             if (m_compare(it->first, key))
20020             {
20021                 // Since we cannot move const Keys, re-construct them in place
20022                 for (auto next = it; ++next != this->end(); ++it)
20023                 {
20024                     it->~value_type(); // Destroy but keep allocation
20025                     new (&*it) value_type{std::move(*next)};
20026                 }
20027                 Container::pop_back();
20028                 return 1;
20029             }
20030         }
20031         return 0;
20032     }
20033
20034     iterator erase(iterator pos)
20035     {
20036         return erase(pos, std::next(pos));
20037     }
20038
20039     iterator erase(iterator first, iterator last)
20040     {
20041         if (first == last)
20042         {
20043             return first;
20044         }
20045
20046         const auto elements_affected = std::distance(first, last);
20047         const auto offset = std::distance(Container::begin(), first);
20048
20049         // This is the start situation. We need to delete elements_affected
20050         // elements (3 in this example: e, f, g), and need to return an
20051         // iterator past the last deleted element (h in this example).
20052         // Note that offset is the distance from the start of the vector
20053         // to first. We will need this later.
20054
20055         // [ a, b, c, d, e, f, g, h, i, j ]
20056         //               ^        ^
20057         //             first    last
20058
20059         // Since we cannot move const Keys, we re-construct them in place.
20060         // We start at first and re-construct (viz. copy) the elements from
20061         // the back of the vector. Example for the first iteration:
20062
20063         //               ,--------.
20064         //               v        |   destroy e and re-construct with h
20065         // [ a, b, c, d, e, f, g, h, i, j ]
20066         //               ^        ^
20067         //               it       it + elements_affected
20068
20069         for (auto it = first; std::next(it, elements_affected) != Container::end(); ++it)
20070         {
20071             it->~value_type(); // destroy but keep allocation
20072             new (&*it) value_type{std::move(*std::next(it, elements_affected))}; // "move" next
     element to it
20073         }
20074
20075         // [ a, b, c, d, h, i, j, h, i, j ]
20076         //               ^        ^
20077         //             first    last
20078
20079         // remove the unneeded elements at the end of the vector
20080         Container::resize(this->size() - static_cast<size_type>(elements_affected));
20081
```

```
20082         // [ a, b, c, d, h, i, j ]
20083         //              ^        ^
20084         //            first    last
20085
20086         // first is now pointing past the last deleted element, but we cannot
20087         // use this iterator, because it may have been invalidated by the
20088         // resize call. Instead, we can return begin() + offset.
20089         return Container::begin() + offset;
20090     }
20091
20092     size_type count(const key_type& key) const
20093     {
20094         for (auto it = this->begin(); it != this->end(); ++it)
20095         {
20096             if (m_compare(it->first, key))
20097             {
20098                 return 1;
20099             }
20100         }
20101         return 0;
20102     }
20103
20104     template<class KeyType, detail::enable_if_t<
20105                  detail::is_usable_as_key_type<key_compare, key_type, KeyType>::value, int> = 0>
20106     size_type count(KeyType && key) const // NOLINT(cppcoreguidelines-missing-std-forward)
20107     {
20108         for (auto it = this->begin(); it != this->end(); ++it)
20109         {
20110             if (m_compare(it->first, key))
20111             {
20112                 return 1;
20113             }
20114         }
20115         return 0;
20116     }
20117
20118     iterator find(const key_type& key)
20119     {
20120         for (auto it = this->begin(); it != this->end(); ++it)
20121         {
20122             if (m_compare(it->first, key))
20123             {
20124                 return it;
20125             }
20126         }
20127         return Container::end();
20128     }
20129
20130     template<class KeyType, detail::enable_if_t<
20131                  detail::is_usable_as_key_type<key_compare, key_type, KeyType>::value, int> = 0>
20132     iterator find(KeyType && key) // NOLINT(cppcoreguidelines-missing-std-forward)
20133     {
20134         for (auto it = this->begin(); it != this->end(); ++it)
20135         {
20136             if (m_compare(it->first, key))
20137             {
20138                 return it;
20139             }
20140         }
20141         return Container::end();
20142     }
20143
20144     const_iterator find(const key_type& key) const
20145     {
20146         for (auto it = this->begin(); it != this->end(); ++it)
20147         {
20148             if (m_compare(it->first, key))
20149             {
20150                 return it;
20151             }
20152         }
20153         return Container::end();
20154     }
20155
20156     std::pair<iterator, bool> insert( value_type&& value )
20157     {
20158         return emplace(value.first, std::move(value.second));
20159     }
20160
20161     std::pair<iterator, bool> insert( const value_type& value )
20162     {
20163         for (auto it = this->begin(); it != this->end(); ++it)
20164         {
20165             if (m_compare(it->first, value.first))
20166             {
20167                 return {it, false};
20168             }
```

```
20169         }
20170         Container::push_back(value);
20171         return {--this->end(), true};
20172     }
20173 
20174     template<typename InputIt>
20175     using require_input_iter = typename std::enable_if<std::is_convertible<typename
      std::iterator_traits<InputIt>::iterator_category,
20176         std::input_iterator_tag>::value>::type;
20177 
20178     template<typename InputIt, typename = require_input_iter<InputIt»
20179     void insert(InputIt first, InputIt last)
20180     {
20181         for (auto it = first; it != last; ++it)
20182         {
20183             insert(*it);
20184         }
20185     }
20186 
20187 private:
20188     JSON_NO_UNIQUE_ADDRESS key_compare m_compare = key_compare();
20189 };
20190 
20191 NLOHMANN_JSON_NAMESPACE_END
20192 
20193 
20194 #if defined(JSON_HAS_CPP_17)
20195     #if JSON_HAS_STATIC_RTTI
20196         #include <any>
20197     #endif
20198     #include <string_view>
20199 #endif
20200 
20206 NLOHMANN_JSON_NAMESPACE_BEGIN
20207 
20226 NLOHMANN_BASIC_JSON_TPL_DECLARATION
20227 class basic_json // NOLINT(cppcoreguidelines-special-member-functions,hicpp-special-member-functions)
20228     : public ::nlohmann::detail::json_base_class<CustomBaseClass>
20229 {
20230   private:
20231     template<detail::value_t> friend struct detail::external_constructor;
20232 
20233     template<typename>
20234     friend class ::nlohmann::json_pointer;
20235     // can be restored when json_pointer backwards compatibility is removed
20236     // friend ::nlohmann::json_pointer<StringType>;
20237 
20238     template<typename BasicJsonType, typename InputType>
20239     friend class ::nlohmann::detail::parser;
20240     friend ::nlohmann::detail::serializer<basic_json>;
20241     template<typename BasicJsonType>
20242     friend class ::nlohmann::detail::iter_impl;
20243     template<typename BasicJsonType, typename CharType>
20244     friend class ::nlohmann::detail::binary_writer;
20245     template<typename BasicJsonType, typename InputType, typename SAX>
20246     friend class ::nlohmann::detail::binary_reader;
20247     template<typename BasicJsonType, typename InputAdapterType>
20248     friend class ::nlohmann::detail::json_sax_dom_parser;
20249     template<typename BasicJsonType, typename InputAdapterType>
20250     friend class ::nlohmann::detail::json_sax_dom_callback_parser;
20251     friend class ::nlohmann::detail::exception;
20252 
20254     using basic_json_t = NLOHMANN_BASIC_JSON_TPL;
20255     using json_base_class_t = ::nlohmann::detail::json_base_class<CustomBaseClass>;
20256 
20257   JSON_PRIVATE_UNLESS_TESTED:
20258     // convenience aliases for types residing in namespace detail;
20259     using lexer = ::nlohmann::detail::lexer_base<basic_json>;
20260 
20261     template<typename InputAdapterType>
20262     static ::nlohmann::detail::parser<basic_json, InputAdapterType> parser(
20263         InputAdapterType adapter,
20264         detail::parser_callback_t<basic_json>cb = nullptr,
20265         const bool allow_exceptions = true,
20266         const bool ignore_comments = false,
20267         const bool ignore_trailing_commas = false
20268                                     )
20269     {
20270         return ::nlohmann::detail::parser<basic_json, InputAdapterType>(std::move(adapter),
20271             std::move(cb), allow_exceptions, ignore_comments, ignore_trailing_commas);
20272     }
20273 
20274   private:
20275     using primitive_iterator_t = ::nlohmann::detail::primitive_iterator_t;
20276     template<typename BasicJsonType>
20277     using internal_iterator = ::nlohmann::detail::internal_iterator<BasicJsonType>;
20278     template<typename BasicJsonType>
```

```
20279     using iter_impl = ::nlohmann::detail::iter_impl<BasicJsonType>;
20280     template<typename Iterator>
20281     using iteration_proxy = ::nlohmann::detail::iteration_proxy<Iterator>;
20282     template<typename Base> using json_reverse_iterator =
    ::nlohmann::detail::json_reverse_iterator<Base>;
20283
20284     template<typename CharType>
20285     using output_adapter_t = ::nlohmann::detail::output_adapter_t<CharType>;
20286
20287     template<typename InputType>
20288     using binary_reader = ::nlohmann::detail::binary_reader<basic_json, InputType>;
20289     template<typename CharType> using binary_writer = ::nlohmann::detail::binary_writer<basic_json,
    CharType>;
20290
20291   JSON_PRIVATE_UNLESS_TESTED:
20292     using serializer = ::nlohmann::detail::serializer<basic_json>;
20293
20294   public:
20295     using value_t = detail::value_t;
20297     using json_pointer = ::nlohmann::json_pointer<StringType>;
20298     template<typename T, typename SFINAE>
20299     using json_serializer = JSONSerializer<T, SFINAE>;
20301     using error_handler_t = detail::error_handler_t;
20303     using cbor_tag_handler_t = detail::cbor_tag_handler_t;
20305     using bjdata_version_t = detail::bjdata_version_t;
20307     using initializer_list_t = std::initializer_list<detail::json_ref<basic_json>>;
20308
20309     using input_format_t = detail::input_format_t;
20311     using json_sax_t = json_sax<basic_json>;
20312
20314     // exceptions //
20316
20320
20321     using exception = detail::exception;
20322     using parse_error = detail::parse_error;
20323     using invalid_iterator = detail::invalid_iterator;
20324     using type_error = detail::type_error;
20325     using out_of_range = detail::out_of_range;
20326     using other_error = detail::other_error;
20327
20329
20331     // container types //
20333
20338
20340     using value_type = basic_json;
20341
20343     using reference = value_type&;
20345     using const_reference = const value_type&;
20346
20348     using difference_type = std::ptrdiff_t;
20350     using size_type = std::size_t;
20351
20353     using allocator_type = AllocatorType<basic_json>;
20354
20356     using pointer = typename std::allocator_traits<allocator_type>::pointer;
20358     using const_pointer = typename std::allocator_traits<allocator_type>::const_pointer;
20359
20361     using iterator = iter_impl<basic_json>;
20363     using const_iterator = iter_impl<const basic_json>;
20365     using reverse_iterator = json_reverse_iterator<typename basic_json::iterator>;
20367     using const_reverse_iterator = json_reverse_iterator<typename basic_json::const_iterator>;
20368
20370
20373     static allocator_type get_allocator()
20374     {
20375         return allocator_type();
20376     }
20377
20380     JSON_HEDLEY_WARN_UNUSED_RESULT
20381     static basic_json meta()
20382     {
20383         basic_json result;
20384
20385         result["copyright"] = "(C) 2013-2026 Niels Lohmann";
20386         result["name"] = "JSON for Modern C++";
20387         result["url"] = "https://github.com/nlohmann/json";
20388         result["version"]["string"] =
20389             detail::concat(std::to_string(NLOHMANN_JSON_VERSION_MAJOR), '.',
20390                            std::to_string(NLOHMANN_JSON_VERSION_MINOR), '.',
20391                            std::to_string(NLOHMANN_JSON_VERSION_PATCH));
20392         result["version"]["major"] = NLOHMANN_JSON_VERSION_MAJOR;
20393         result["version"]["minor"] = NLOHMANN_JSON_VERSION_MINOR;
20394         result["version"]["patch"] = NLOHMANN_JSON_VERSION_PATCH;
20395
20396 #ifdef _WIN32
20397         result["platform"] = "win32";
20398 #elif defined __linux__
```

```
20399         result["platform"] = "linux";
20400 #elif defined __APPLE__
20401         result["platform"] = "apple";
20402 #elif defined __unix__
20403         result["platform"] = "unix";
20404 #else
20405         result["platform"] = "unknown";
20406 #endif
20407
20408 #if defined(__ICC) || defined(__INTEL_COMPILER)
20409         result["compiler"] = {{"family", "icc"}, {"version", __INTEL_COMPILER}};
20410 #elif defined(__clang__)
20411         result["compiler"] = {{"family", "clang"}, {"version", __clang_version__}};
20412 #elif defined(__GNUC__) || defined(__GNUG__)
20413         result["compiler"] = {{"family", "gcc"}, {"version", detail::concat(
20414                     std::to_string(__GNUC__), '.',
20415                     std::to_string(__GNUC_MINOR__), '.',
20416                     std::to_string(__GNUC_PATCHLEVEL__))
20417             }
20418         };
20419 #elif defined(__HP_cc) || defined(__HP_aCC)
20420         result["compiler"] = "hp"
20421 #elif defined(__IBMCPP__)
20422         result["compiler"] = {{"family", "ilecpp"}, {"version", __IBMCPP__}};
20423 #elif defined(_MSC_VER)
20424         result["compiler"] = {{"family", "msvc"}, {"version", _MSC_VER}};
20425 #elif defined(__PGI)
20426         result["compiler"] = {{"family", "pgcpp"}, {"version", __PGI}};
20427 #elif defined(__SUNPRO_CC)
20428         result["compiler"] = {{"family", "sunpro"}, {"version", __SUNPRO_CC}};
20429 #else
20430         result["compiler"] = {{"family", "unknown"}, {"version", "unknown"}};
20431 #endif
20432
20433 #if defined(_MSVC_LANG)
20434         result["compiler"]["c++"] = std::to_string(_MSVC_LANG);
20435 #elif defined(__cplusplus)
20436         result["compiler"]["c++"] = std::to_string(__cplusplus);
20437 #else
20438         result["compiler"]["c++"] = "unknown";
20439 #endif
20440         return result;
20441     }
20442
20444     // JSON value data types //
20446
20451
20456 #if defined(JSON_HAS_CPP_14)
20457     // use of transparent comparator avoids unnecessary repeated construction of temporaries
20458     // in functions involving lookup by key with types other than object_t::key_type (aka. StringType)
20459     using default_object_comparator_t = std::less<>;
20460 #else
20461     using default_object_comparator_t = std::less<StringType>;
20462 #endif
20463
20466     using object_t = ObjectType<StringType,
20467           basic_json,
20468           default_object_comparator_t,
20469           AllocatorType<std::pair<const StringType,
20470           basic_json>>;
20471
20474     using array_t = ArrayType<basic_json, AllocatorType<basic_json>;
20475
20478     using string_t = StringType;
20479
20482     using boolean_t = BooleanType;
20483
20486     using number_integer_t = NumberIntegerType;
20487
20490     using number_unsigned_t = NumberUnsignedType;
20491
20494     using number_float_t = NumberFloatType;
20495
20498     using binary_t = nlohmann::byte_container_with_subtype<BinaryType>;
20499
20502     using object_comparator_t = detail::actual_object_comparator_t<basic_json>;
20503
20505
20506   private:
20507
20509     template<typename T, typename... Args>
20510     JSON_HEDLEY_RETURNS_NON_NULL
20511     static T* create(Args&& ... args)
20512     {
20513         AllocatorType<T> alloc;
20514         using AllocatorTraits = std::allocator_traits<AllocatorType<T>;
20515
```

```
20516         auto deleter = [&](T * obj)
20517         {
20518             AllocatorTraits::deallocate(alloc, obj, 1);
20519         };
20520         std::unique_ptr<T, decltype(deleter)> obj(AllocatorTraits::allocate(alloc, 1), deleter);
20521         AllocatorTraits::construct(alloc, obj.get(), std::forward<Args>(args)...);
20522         JSON_ASSERT(obj != nullptr);
20523         return obj.release();
20524     }
20525
20527     // JSON value storage //
20529
20530   JSON_PRIVATE_UNLESS_TESTED:
20556     union json_value
20557     {
20559         object_t* object;
20561         array_t* array;
20563         string_t* string;
20565         binary_t* binary;
20567         boolean_t boolean;
20569         number_integer_t number_integer;
20571         number_unsigned_t number_unsigned;
20573         number_float_t number_float;
20574
20576         json_value() = default;
20578         json_value(boolean_t v) noexcept : boolean(v) {}
20580         json_value(number_integer_t v) noexcept : number_integer(v) {}
20582         json_value(number_unsigned_t v) noexcept : number_unsigned(v) {}
20584         json_value(number_float_t v) noexcept : number_float(v) {}
20586         json_value(value_t t)
20587         {
20588             switch (t)
20589             {
20590                 case value_t::object:
20591                 {
20592                     object = create<object_t>();
20593                     break;
20594                 }
20595
20596                 case value_t::array:
20597                 {
20598                     array = create<array_t>();
20599                     break;
20600                 }
20601
20602                 case value_t::string:
20603                 {
20604                     string = create<string_t>("");
20605                     break;
20606                 }
20607
20608                 case value_t::binary:
20609                 {
20610                     binary = create<binary_t>();
20611                     break;
20612                 }
20613
20614                 case value_t::boolean:
20615                 {
20616                     boolean = static_cast<boolean_t>(false);
20617                     break;
20618                 }
20619
20620                 case value_t::number_integer:
20621                 {
20622                     number_integer = static_cast<number_integer_t>(0);
20623                     break;
20624                 }
20625
20626                 case value_t::number_unsigned:
20627                 {
20628                     number_unsigned = static_cast<number_unsigned_t>(0);
20629                     break;
20630                 }
20631
20632                 case value_t::number_float:
20633                 {
20634                     number_float = static_cast<number_float_t>(0.0);
20635                     break;
20636                 }
20637
20638                 case value_t::null:
20639                 {
20640                     object = nullptr;  // silence warning, see #821
20641                     break;
20642                 }
20643
```

```
20644                 case value_t::discarded:
20645                 default:
20646                 {
20647                     object = nullptr;  // silence warning, see #821
20648                     if (JSON_HEDLEY_UNLIKELY(t == value_t::null))
20649                     {
20650                         JSON_THROW(other_error::create(500, "961c151d2e87f2686a955a9be24d316f1362bf21
      3.12.0", nullptr)); // LCOV_EXCL_LINE
20651                     }
20652                     break;
20653                 }
20654             }
20655         }
20656
20658         json_value(const string_t& value) : string(create<string_t>(value)) {}
20659
20661         json_value(string_t&& value) : string(create<string_t>(std::move(value))) {}
20662
20664         json_value(const object_t& value) : object(create<object_t>(value)) {}
20665
20667         json_value(object_t&& value) : object(create<object_t>(std::move(value))) {}
20668
20670         json_value(const array_t& value) : array(create<array_t>(value)) {}
20671
20673         json_value(array_t&& value) : array(create<array_t>(std::move(value))) {}
20674
20676         json_value(const typename binary_t::container_type& value) : binary(create<binary_t>(value))
      {}
20677
20679         json_value(typename binary_t::container_type&& value) :
      binary(create<binary_t>(std::move(value))) {}
20680
20682         json_value(const binary_t& value) : binary(create<binary_t>(value)) {}
20683
20685         json_value(binary_t&& value) : binary(create<binary_t>(std::move(value))) {}
20686
20687         void destroy(value_t t)
20688         {
20689             if (
20690                 (t == value_t::object && object == nullptr) ||
20691                 (t == value_t::array && array == nullptr) ||
20692                 (t == value_t::string && string == nullptr) ||
20693                 (t == value_t::binary && binary == nullptr)
20694             )
20695             {
20696                 // not initialized (e.g., due to exception in the ctor)
20697                 return;
20698             }
20699             if (t == value_t::array || t == value_t::object)
20700             {
20701                 // flatten the current json_value to a heap-allocated stack
20702                 std::vector<basic_json> stack;
20703
20704                 // move the top-level items to stack
20705                 if (t == value_t::array)
20706                 {
20707                     stack.reserve(array->size());
20708                     std::move(array->begin(), array->end(), std::back_inserter(stack));
20709                 }
20710                 else
20711                 {
20712                     stack.reserve(object->size());
20713                     for (auto&& it : *object)
20714                     {
20715                         stack.push_back(std::move(it.second));
20716                     }
20717                 }
20718
20719                 while (!stack.empty())
20720                 {
20721                     // move the last item to a local variable to be processed
20722                     basic_json current_item(std::move(stack.back()));
20723                     stack.pop_back();
20724
20725                     // if current_item is array/object, move
20726                     // its children to the stack to be processed later
20727                     if (current_item.is_array())
20728                     {
20729                         std::move(current_item.m_data.m_value.array->begin(),
      current_item.m_data.m_value.array->end(), std::back_inserter(stack));
20730
20731                         current_item.m_data.m_value.array->clear();
20732                     }
20733                     else if (current_item.is_object())
20734                     {
20735                         for (auto&& it : *current_item.m_data.m_value.object)
20736                         {
```

```
20737                             stack.push_back(std::move(it.second));
20738                         }
20739
20740                         current_item.m_data.m_value.object->clear();
20741                     }
20742
20743                     // it's now safe that current_item gets destructed
20744                     // since it doesn't have any children
20745                 }
20746             }
20747
20748             switch (t)
20749             {
20750                 case value_t::object:
20751                 {
20752                     AllocatorType<object_t> alloc;
20753                     std::allocator_traits<decltype(alloc)>::destroy(alloc, object);
20754                     std::allocator_traits<decltype(alloc)>::deallocate(alloc, object, 1);
20755                     break;
20756                 }
20757
20758                 case value_t::array:
20759                 {
20760                     AllocatorType<array_t> alloc;
20761                     std::allocator_traits<decltype(alloc)>::destroy(alloc, array);
20762                     std::allocator_traits<decltype(alloc)>::deallocate(alloc, array, 1);
20763                     break;
20764                 }
20765
20766                 case value_t::string:
20767                 {
20768                     AllocatorType<string_t> alloc;
20769                     std::allocator_traits<decltype(alloc)>::destroy(alloc, string);
20770                     std::allocator_traits<decltype(alloc)>::deallocate(alloc, string, 1);
20771                     break;
20772                 }
20773
20774                 case value_t::binary:
20775                 {
20776                     AllocatorType<binary_t> alloc;
20777                     std::allocator_traits<decltype(alloc)>::destroy(alloc, binary);
20778                     std::allocator_traits<decltype(alloc)>::deallocate(alloc, binary, 1);
20779                     break;
20780                 }
20781
20782                 case value_t::null:
20783                 case value_t::boolean:
20784                 case value_t::number_integer:
20785                 case value_t::number_unsigned:
20786                 case value_t::number_float:
20787                 case value_t::discarded:
20788                 default:
20789                 {
20790                     break;
20791                 }
20792             }
20793         }
20794     };
20795
20796   private:
20815     void assert_invariant(bool check_parents = true) const noexcept
20816     {
20817         JSON_ASSERT(m_data.m_type != value_t::object || m_data.m_value.object != nullptr);
20818         JSON_ASSERT(m_data.m_type != value_t::array || m_data.m_value.array != nullptr);
20819         JSON_ASSERT(m_data.m_type != value_t::string || m_data.m_value.string != nullptr);
20820         JSON_ASSERT(m_data.m_type != value_t::binary || m_data.m_value.binary != nullptr);
20821
20822 #if JSON_DIAGNOSTICS
20823         JSON_TRY
20824         {
20825             // cppcheck-suppress assertWithSideEffect
20826             JSON_ASSERT(!check_parents || !is_structured() || std::all_of(begin(), end(), [this](const
      basic_json & j)
20827             {
20828                 return j.m_parent == this;
20829             }));
20830         }
20831         JSON_CATCH(...) {} // LCOV_EXCL_LINE
20832 #endif
20833         static_cast<void>(check_parents);
20834     }
20835
20836     void set_parents()
20837     {
20838 #if JSON_DIAGNOSTICS
20839         switch (m_data.m_type)
20840         {
```

```
20841             case value_t::array:
20842             {
20843                 for (auto& element : *m_data.m_value.array)
20844                 {
20845                     element.m_parent = this;
20846                 }
20847                 break;
20848             }
20849
20850             case value_t::object:
20851             {
20852                 for (auto& element : *m_data.m_value.object)
20853                 {
20854                     element.second.m_parent = this;
20855                 }
20856                 break;
20857             }
20858
20859             case value_t::null:
20860             case value_t::string:
20861             case value_t::boolean:
20862             case value_t::number_integer:
20863             case value_t::number_unsigned:
20864             case value_t::number_float:
20865             case value_t::binary:
20866             case value_t::discarded:
20867             default:
20868                 break;
20869         }
20870 #endif
20871     }
20872
20873     iterator set_parents(iterator it, typename iterator::difference_type count_set_parents)
20874     {
20875 #if JSON_DIAGNOSTICS
20876         for (typename iterator::difference_type i = 0; i < count_set_parents; ++i)
20877         {
20878             (it + i)->m_parent = this;
20879         }
20880 #else
20881         static_cast<void>(count_set_parents);
20882 #endif
20883         return it;
20884     }
20885
20886     reference set_parent(reference j, std::size_t old_capacity = detail::unknown_size())
20887     {
20888 #if JSON_DIAGNOSTICS
20889         if (old_capacity != detail::unknown_size())
20890         {
20891             // see https://github.com/nlohmann/json/issues/2838
20892             JSON_ASSERT(type() == value_t::array);
20893             if (JSON_HEDLEY_UNLIKELY(m_data.m_value.array->capacity() != old_capacity))
20894             {
20895                 // capacity has changed: update all parents
20896                 set_parents();
20897                 return j;
20898             }
20899         }
20900
20901         // ordered_json uses a vector internally, so pointers could have
20902         // been invalidated; see https://github.com/nlohmann/json/issues/2962
20903 #ifdef JSON_HEDLEY_MSVC_VERSION
20904 #pragma warning(push )
20905 #pragma warning(disable : 4127) // ignore warning to replace if with if constexpr
20906 #endif
20907         if (detail::is_ordered_map<object_t>::value)
20908         {
20909             set_parents();
20910             return j;
20911         }
20912 #ifdef JSON_HEDLEY_MSVC_VERSION
20913 #pragma warning( pop )
20914 #endif
20915
20916         j.m_parent = this;
20917 #else
20918         static_cast<void>(j);
20919         static_cast<void>(old_capacity);
20920 #endif
20921         return j;
20922     }
20923
20924   public:
20926     // JSON parser callback //
20928
20931     using parse_event_t = detail::parse_event_t;
```

```
20932
20935     using parser_callback_t = detail::parser_callback_t<basic_json>;
20936
20938     // constructors //
20940
20945
20948     basic_json(const value_t v)
20949         : m_data(v)
20950     {
20951         assert_invariant();
20952     }
20953
20956     basic_json(std::nullptr_t = nullptr) noexcept // NOLINT(bugprone-exception-escape)
20957         : basic_json(value_t::null)
20958     {
20959         assert_invariant();
20960     }
20961
20964     template < typename CompatibleType,
20965                typename U = detail::uncvref_t<CompatibleType>,
20966                detail::enable_if_t <
20967                    !detail::is_basic_json<U>::value &&
      detail::is_compatible_type<basic_json_t, U>::value, int > = 0 >
20968     basic_json(CompatibleType && val) noexcept(noexcept( //
      NOLINT(bugprone-forwarding-reference-overload,bugprone-exception-escape)
20969             JSONSerializer<U>::to_json(std::declval<basic_json_t&>(),
20970                                        std::forward<CompatibleType>(val))))
20971     {
20972         JSONSerializer<U>::to_json(*this, std::forward<CompatibleType>(val));
20973         set_parents();
20974         assert_invariant();
20975     }
20976
20979     template < typename BasicJsonType,
20980                detail::enable_if_t <
20981                    detail::is_basic_json<BasicJsonType>::value&& !std::is_same<basic_json,
      BasicJsonType>::value, int > = 0 >
20982     basic_json(const BasicJsonType& val)
20983 #if JSON_DIAGNOSTIC_POSITIONS
20984         : start_position(val.start_pos()),
20985           end_position(val.end_pos())
20986 #endif
20987     {
20988         using other_boolean_t = typename BasicJsonType::boolean_t;
20989         using other_number_float_t = typename BasicJsonType::number_float_t;
20990         using other_number_integer_t = typename BasicJsonType::number_integer_t;
20991         using other_number_unsigned_t = typename BasicJsonType::number_unsigned_t;
20992         using other_string_t = typename BasicJsonType::string_t;
20993         using other_object_t = typename BasicJsonType::object_t;
20994         using other_array_t = typename BasicJsonType::array_t;
20995         using other_binary_t = typename BasicJsonType::binary_t;
20996
20997         switch (val.type())
20998         {
20999             case value_t::boolean:
21000                 JSONSerializer<other_boolean_t>::to_json(*this, val.template get<other_boolean_t>());
21001                 break;
21002             case value_t::number_float:
21003                 JSONSerializer<other_number_float_t>::to_json(*this, val.template
      get<other_number_float_t>());
21004                 break;
21005             case value_t::number_integer:
21006                 JSONSerializer<other_number_integer_t>::to_json(*this, val.template
      get<other_number_integer_t>());
21007                 break;
21008             case value_t::number_unsigned:
21009                 JSONSerializer<other_number_unsigned_t>::to_json(*this, val.template
      get<other_number_unsigned_t>());
21010                 break;
21011             case value_t::string:
21012                 JSONSerializer<other_string_t>::to_json(*this, val.template get_ref<const
      other_string_t&>());
21013                 break;
21014             case value_t::object:
21015                 JSONSerializer<other_object_t>::to_json(*this, val.template get_ref<const
      other_object_t&>());
21016                 break;
21017             case value_t::array:
21018                 JSONSerializer<other_array_t>::to_json(*this, val.template get_ref<const
      other_array_t&>());
21019                 break;
21020             case value_t::binary:
21021                 JSONSerializer<other_binary_t>::to_json(*this, val.template get_ref<const
      other_binary_t&>());
21022                 break;
21023             case value_t::null:
21024                 *this = nullptr;
```

```
21025                 break;
21026             case value_t::discarded:
21027                 m_data.m_type = value_t::discarded;
21028                 break;
21029             default:            // LCOV_EXCL_LINE
21030                 JSON_ASSERT(false); // NOLINT(cert-dcl03-c,hicpp-static-assert,misc-static-assert)
      LCOV_EXCL_LINE
21031         }
21032         JSON_ASSERT(m_data.m_type == val.type());
21033
21034         set_parents();
21035         assert_invariant();
21036     }
21037
21040     basic_json(initializer_list_t init,
21041                bool type_deduction = true,
21042                value_t manual_type = value_t::array)
21043     {
21044         // check if each element is an array with two elements whose first
21045         // element is a string
21046         bool is_an_object = std::all_of(init.begin(), init.end(),
21047                                         [](const detail::json_ref<basic_json>& element_ref)
21048         {
21049             // The cast is to ensure op[size_type] is called, bearing in mind size_type may not be
      int;
21050             // (many string types can be constructed from 0 via its null-pointer guise, so we get a
21051             // broken call to op[key_type], the wrong semantics, and a 4804 warning on Windows)
21052             return element_ref->is_array() && element_ref->size() == 2 &&
      (*element_ref)[static_cast<size_type>(0)].is_string();
21053         });
21054
21055         // adjust type if type deduction is not wanted
21056         if (!type_deduction)
21057         {
21058             // if an array is wanted, do not create an object though possible
21059             if (manual_type == value_t::array)
21060             {
21061                 is_an_object = false;
21062             }
21063
21064             // if an object is wanted but impossible, throw an exception
21065             if (JSON_HEDLEY_UNLIKELY(manual_type == value_t::object && !is_an_object))
21066             {
21067                 JSON_THROW(type_error::create(301, "cannot create object from initializer list",
      nullptr));
21068             }
21069         }
21070
21071         if (is_an_object)
21072         {
21073             // the initializer list is a list of pairs -> create an object
21074             m_data.m_type = value_t::object;
21075             m_data.m_value = value_t::object;
21076
21077             for (auto& element_ref : init)
21078             {
21079                 auto element = element_ref.moved_or_copied();
21080                 m_data.m_value.object->emplace(
21081                     std::move(*((*element.m_data.m_value.array)[0].m_data.m_value.string)),
21082                     std::move((*element.m_data.m_value.array)[1]));
21083             }
21084         }
21085         else
21086         {
21087             // the initializer list describes an array -> create an array
21088             m_data.m_type = value_t::array;
21089             m_data.m_value.array = create<array_t>(init.begin(), init.end());
21090         }
21091
21092         set_parents();
21093         assert_invariant();
21094     }
21095
21098     JSON_HEDLEY_WARN_UNUSED_RESULT
21099     static basic_json binary(const typename binary_t::container_type& init)
21100     {
21101         auto res = basic_json();
21102         res.m_data.m_type = value_t::binary;
21103         res.m_data.m_value = init;
21104         return res;
21105     }
21106
21109     JSON_HEDLEY_WARN_UNUSED_RESULT
21110     static basic_json binary(const typename binary_t::container_type& init, typename
      binary_t::subtype_type subtype)
21111     {
21112         auto res = basic_json();
```

```
21113         res.m_data.m_type = value_t::binary;
21114         res.m_data.m_value = binary_t(init, subtype);
21115         return res;
21116     }
21117
21120     JSON_HEDLEY_WARN_UNUSED_RESULT
21121     static basic_json binary(typename binary_t::container_type&& init)
21122     {
21123         auto res = basic_json();
21124         res.m_data.m_type = value_t::binary;
21125         res.m_data.m_value = std::move(init);
21126         return res;
21127     }
21128
21131     JSON_HEDLEY_WARN_UNUSED_RESULT
21132     static basic_json binary(typename binary_t::container_type&& init, typename binary_t::subtype_type
      subtype)
21133     {
21134         auto res = basic_json();
21135         res.m_data.m_type = value_t::binary;
21136         res.m_data.m_value = binary_t(std::move(init), subtype);
21137         return res;
21138     }
21139
21142     JSON_HEDLEY_WARN_UNUSED_RESULT
21143     static basic_json array(initializer_list_t init = {})
21144     {
21145         return basic_json(init, false, value_t::array);
21146     }
21147
21150     JSON_HEDLEY_WARN_UNUSED_RESULT
21151     static basic_json object(initializer_list_t init = {})
21152     {
21153         return basic_json(init, false, value_t::object);
21154     }
21155
21158     basic_json(size_type cnt, const basic_json& val):
21159         m_data{cnt, val}
21160     {
21161         set_parents();
21162         assert_invariant();
21163     }
21164
21167     template < class InputIT, typename std::enable_if <
21168                    std::is_same<InputIT, typename basic_json_t::iterator>::value ||
21169                    std::is_same<InputIT, typename basic_json_t::const_iterator>::value, int >::type =
      0 >
21170     basic_json(InputIT first, InputIT last) // NOLINT(performance-unnecessary-value-param)
21171     {
21172         JSON_ASSERT(first.m_object != nullptr);
21173         JSON_ASSERT(last.m_object != nullptr);
21174
21175         // make sure the iterator fits the current value
21176         if (JSON_HEDLEY_UNLIKELY(first.m_object != last.m_object))
21177         {
21178             JSON_THROW(invalid_iterator::create(201, "iterators are not compatible", nullptr));
21179         }
21180
21181         // copy type from the first iterator
21182         m_data.m_type = first.m_object->m_data.m_type;
21183
21184         // check if the iterator range is complete for primitive values
21185         switch (m_data.m_type)
21186         {
21187             case value_t::boolean:
21188             case value_t::number_float:
21189             case value_t::number_integer:
21190             case value_t::number_unsigned:
21191             case value_t::string:
21192             {
21193                 if (JSON_HEDLEY_UNLIKELY(!first.m_it.primitive_iterator.is_begin()
21194                                          || !last.m_it.primitive_iterator.is_end()))
21195                 {
21196                     JSON_THROW(invalid_iterator::create(204, "iterators out of range",
      first.m_object));
21197                 }
21198                 break;
21199             }
21200
21201             case value_t::null:
21202             case value_t::object:
21203             case value_t::array:
21204             case value_t::binary:
21205             case value_t::discarded:
21206             default:
21207                 break;
21208         }
```

```
21209
21210         switch (m_data.m_type)
21211         {
21212             case value_t::number_integer:
21213             {
21214                 m_data.m_value.number_integer = first.m_object->m_data.m_value.number_integer;
21215                 break;
21216             }
21217
21218             case value_t::number_unsigned:
21219             {
21220                 m_data.m_value.number_unsigned = first.m_object->m_data.m_value.number_unsigned;
21221                 break;
21222             }
21223
21224             case value_t::number_float:
21225             {
21226                 m_data.m_value.number_float = first.m_object->m_data.m_value.number_float;
21227                 break;
21228             }
21229
21230             case value_t::boolean:
21231             {
21232                 m_data.m_value.boolean = first.m_object->m_data.m_value.boolean;
21233                 break;
21234             }
21235
21236             case value_t::string:
21237             {
21238                 m_data.m_value = *first.m_object->m_data.m_value.string;
21239                 break;
21240             }
21241
21242             case value_t::object:
21243             {
21244                 m_data.m_value.object = create<object_t>(first.m_it.object_iterator,
21245                                         last.m_it.object_iterator);
21246                 break;
21247             }
21248
21249             case value_t::array:
21250             {
21251                 m_data.m_value.array = create<array_t>(first.m_it.array_iterator,
21252                                                        last.m_it.array_iterator);
21253                 break;
21254             }
21255
21256             case value_t::binary:
21257             {
21258                 m_data.m_value = *first.m_object->m_data.m_value.binary;
21259                 break;
21260             }
21261
21262             case value_t::null:
21263             case value_t::discarded:
21264             default:
21265                 JSON_THROW(invalid_iterator::create(206, detail::concat("cannot construct with
      iterators from ", first.m_object->type_name()), first.m_object));
21266         }
21267
21268         set_parents();
21269         assert_invariant();
21270     }
21271
21273     // other constructors and destructor //
21275
21276     template<typename JsonRef,
21277              detail::enable_if_t<detail::conjunction<detail::is_json_ref<JsonRef>,
21278                                  std::is_same<typename JsonRef::value_type, basic_json>::value, int> =
      0 >
21279     basic_json(const JsonRef& ref) : basic_json(ref.moved_or_copied()) {}
21280
21283     basic_json(const basic_json& other)
21284         : json_base_class_t(other)
21285 #if JSON_DIAGNOSTIC_POSITIONS
21286         , start_position(other.start_position)
21287         , end_position(other.end_position)
21288 #endif
21289     {
21290         m_data.m_type = other.m_data.m_type;
21291         // check of passed value is valid
21292         other.assert_invariant();
21293
21294         switch (m_data.m_type)
21295         {
21296             case value_t::object:
21297             {
```

```
21298                 m_data.m_value = *other.m_data.m_value.object;
21299                 break;
21300             }
21301
21302             case value_t::array:
21303             {
21304                 m_data.m_value = *other.m_data.m_value.array;
21305                 break;
21306             }
21307
21308             case value_t::string:
21309             {
21310                 m_data.m_value = *other.m_data.m_value.string;
21311                 break;
21312             }
21313
21314             case value_t::boolean:
21315             {
21316                 m_data.m_value = other.m_data.m_value.boolean;
21317                 break;
21318             }
21319
21320             case value_t::number_integer:
21321             {
21322                 m_data.m_value = other.m_data.m_value.number_integer;
21323                 break;
21324             }
21325
21326             case value_t::number_unsigned:
21327             {
21328                 m_data.m_value = other.m_data.m_value.number_unsigned;
21329                 break;
21330             }
21331
21332             case value_t::number_float:
21333             {
21334                 m_data.m_value = other.m_data.m_value.number_float;
21335                 break;
21336             }
21337
21338             case value_t::binary:
21339             {
21340                 m_data.m_value = *other.m_data.m_value.binary;
21341                 break;
21342             }
21343
21344             case value_t::null:
21345             case value_t::discarded:
21346             default:
21347                 break;
21348         }
21349
21350         set_parents();
21351         assert_invariant();
21352     }
21353
21356     basic_json(basic_json&& other) noexcept
21357         : json_base_class_t(std::forward<json_base_class_t>(other)),
21358           m_data(std::move(other.m_data)) // cppcheck-suppress[accessForwarded] TODO check
21359 #if JSON_DIAGNOSTIC_POSITIONS
21360         , start_position(other.start_position) // cppcheck-suppress[accessForwarded] TODO check
21361         , end_position(other.end_position) // cppcheck-suppress[accessForwarded] TODO check
21362 #endif
21363     {
21364         // check that the passed value is valid
21365         other.assert_invariant(false); // cppcheck-suppress[accessForwarded]
21366
21367         // invalidate payload
21368         other.m_data.m_type = value_t::null;
21369         other.m_data.m_value = {};
21370
21371 #if JSON_DIAGNOSTIC_POSITIONS
21372         other.start_position = std::string::npos;
21373         other.end_position = std::string::npos;
21374 #endif
21375
21376         set_parents();
21377         assert_invariant();
21378     }
21379
21382     basic_json& operator=(basic_json other) noexcept ( //
    NOLINT(cppcoreguidelines-c-copy-assignment-signature,misc-unconventional-assign-operator)
21383         std::is_nothrow_move_constructible<value_t>::value&&
21384         std::is_nothrow_move_assignable<value_t>::value&&
21385         std::is_nothrow_move_constructible<json_value>::value&&
21386         std::is_nothrow_move_assignable<json_value>::value&&
21387         std::is_nothrow_move_assignable<json_base_class_t>::value
```

```
21388     )
21389     {
21390         // check that the passed value is valid
21391         other.assert_invariant();
21392 
21393         using std::swap;
21394         swap(m_data.m_type, other.m_data.m_type);
21395         swap(m_data.m_value, other.m_data.m_value);
21396 
21397 #if JSON_DIAGNOSTIC_POSITIONS
21398         swap(start_position, other.start_position);
21399         swap(end_position, other.end_position);
21400 #endif
21401 
21402         json_base_class_t::operator=(std::move(other));
21403 
21404         set_parents();
21405         assert_invariant();
21406         return *this;
21407     }
21408 
21411     ~basic_json() noexcept
21412     {
21413         assert_invariant(false);
21414     }
21415 
21417 
21418   public:
21420     // object inspection //
21422 
21426 
21429     string_t dump(const int indent = -1,
21430                   const char indent_char = ' ',
21431                   const bool ensure_ascii = false,
21432                   const error_handler_t error_handler = error_handler_t::strict) const
21433     {
21434         string_t result;
21435         serializer s(detail::output_adapter<char, string_t>(result), indent_char, error_handler);
21436 
21437         if (indent >= 0)
21438         {
21439             s.dump(*this, true, ensure_ascii, static_cast<unsigned int>(indent));
21440         }
21441         else
21442         {
21443             s.dump(*this, false, ensure_ascii, 0);
21444         }
21445 
21446         return result;
21447     }
21448 
21451     constexpr value_t type() const noexcept
21452     {
21453         return m_data.m_type;
21454     }
21455 
21458     constexpr bool is_primitive() const noexcept
21459     {
21460         return is_null() || is_string() || is_boolean() || is_number() || is_binary();
21461     }
21462 
21465     constexpr bool is_structured() const noexcept
21466     {
21467         return is_array() || is_object();
21468     }
21469 
21472     constexpr bool is_null() const noexcept
21473     {
21474         return m_data.m_type == value_t::null;
21475     }
21476 
21479     constexpr bool is_boolean() const noexcept
21480     {
21481         return m_data.m_type == value_t::boolean;
21482     }
21483 
21486     constexpr bool is_number() const noexcept
21487     {
21488         return is_number_integer() || is_number_float();
21489     }
21490 
21493     constexpr bool is_number_integer() const noexcept
21494     {
21495         return m_data.m_type == value_t::number_integer || m_data.m_type == value_t::number_unsigned;
21496     }
21497 
21500     constexpr bool is_number_unsigned() const noexcept
```

```
21501     {
21502         return m_data.m_type == value_t::number_unsigned;
21503     }
21504
21507     constexpr bool is_number_float() const noexcept
21508     {
21509         return m_data.m_type == value_t::number_float;
21510     }
21511
21514     constexpr bool is_object() const noexcept
21515     {
21516         return m_data.m_type == value_t::object;
21517     }
21518
21521     constexpr bool is_array() const noexcept
21522     {
21523         return m_data.m_type == value_t::array;
21524     }
21525
21528     constexpr bool is_string() const noexcept
21529     {
21530         return m_data.m_type == value_t::string;
21531     }
21532
21535     constexpr bool is_binary() const noexcept
21536     {
21537         return m_data.m_type == value_t::binary;
21538     }
21539
21542     constexpr bool is_discarded() const noexcept
21543     {
21544         return m_data.m_type == value_t::discarded;
21545     }
21546
21549     constexpr operator value_t() const noexcept
21550     {
21551         return m_data.m_type;
21552     }
21553
21555
21556   private:
21558     // value access //
21560
21562     boolean_t get_impl(boolean_t* /*unused*/) const
21563     {
21564         if (JSON_HEDLEY_LIKELY(is_boolean()))
21565         {
21566             return m_data.m_value.boolean;
21567         }
21568
21569         JSON_THROW(type_error::create(302, detail::concat("type must be boolean, but is ",
      type_name()), this));
21570     }
21571
21573     object_t* get_impl_ptr(object_t* /*unused*/) noexcept
21574     {
21575         return is_object() ? m_data.m_value.object : nullptr;
21576     }
21577
21579     constexpr const object_t* get_impl_ptr(const object_t* /*unused*/) const noexcept
21580     {
21581         return is_object() ? m_data.m_value.object : nullptr;
21582     }
21583
21585     array_t* get_impl_ptr(array_t* /*unused*/) noexcept
21586     {
21587         return is_array() ? m_data.m_value.array : nullptr;
21588     }
21589
21591     constexpr const array_t* get_impl_ptr(const array_t* /*unused*/) const noexcept
21592     {
21593         return is_array() ? m_data.m_value.array : nullptr;
21594     }
21595
21597     string_t* get_impl_ptr(string_t* /*unused*/) noexcept
21598     {
21599         return is_string() ? m_data.m_value.string : nullptr;
21600     }
21601
21603     constexpr const string_t* get_impl_ptr(const string_t* /*unused*/) const noexcept
21604     {
21605         return is_string() ? m_data.m_value.string : nullptr;
21606     }
21607
21609     boolean_t* get_impl_ptr(boolean_t* /*unused*/) noexcept
21610     {
21611         return is_boolean() ? &m_data.m_value.boolean : nullptr;
```

```
21612     }
21613
21615     constexpr const boolean_t* get_impl_ptr(const boolean_t* /*unused*/) const noexcept
21616     {
21617         return is_boolean() ? &m_data.m_value.boolean : nullptr;
21618     }
21619
21621     number_integer_t* get_impl_ptr(number_integer_t* /*unused*/) noexcept
21622     {
21623         return m_data.m_type == value_t::number_integer ? &m_data.m_value.number_integer : nullptr;
21624     }
21625
21627     constexpr const number_integer_t* get_impl_ptr(const number_integer_t* /*unused*/) const noexcept
21628     {
21629         return m_data.m_type == value_t::number_integer ? &m_data.m_value.number_integer : nullptr;
21630     }
21631
21633     number_unsigned_t* get_impl_ptr(number_unsigned_t* /*unused*/) noexcept
21634     {
21635         return is_number_unsigned() ? &m_data.m_value.number_unsigned : nullptr;
21636     }
21637
21639     constexpr const number_unsigned_t* get_impl_ptr(const number_unsigned_t* /*unused*/) const
    noexcept
21640     {
21641         return is_number_unsigned() ? &m_data.m_value.number_unsigned : nullptr;
21642     }
21643
21645     number_float_t* get_impl_ptr(number_float_t* /*unused*/) noexcept
21646     {
21647         return is_number_float() ? &m_data.m_value.number_float : nullptr;
21648     }
21649
21651     constexpr const number_float_t* get_impl_ptr(const number_float_t* /*unused*/) const noexcept
21652     {
21653         return is_number_float() ? &m_data.m_value.number_float : nullptr;
21654     }
21655
21657     binary_t* get_impl_ptr(binary_t* /*unused*/) noexcept
21658     {
21659         return is_binary() ? m_data.m_value.binary : nullptr;
21660     }
21661
21663     constexpr const binary_t* get_impl_ptr(const binary_t* /*unused*/) const noexcept
21664     {
21665         return is_binary() ? m_data.m_value.binary : nullptr;
21666     }
21667
21679     template<typename ReferenceType, typename ThisType>
21680     static ReferenceType get_ref_impl(ThisType& obj)
21681     {
21682         // delegate the call to get_ptr<>()
21683         auto* ptr = obj.template get_ptr<typename std::add_pointer<ReferenceType>::type>();
21684
21685         if (JSON_HEDLEY_LIKELY(ptr != nullptr))
21686         {
21687             return *ptr;
21688         }
21689
21690         JSON_THROW(type_error::create(303, detail::concat("incompatible ReferenceType for get_ref,
    actual type is ", obj.type_name()), &obj));
21691     }
21692
21693   public:
21697
21700     template<typename PointerType, typename std::enable_if<
21701                   std::is_pointer<PointerType>::value, int>::type = 0>
21702     auto get_ptr() noexcept ->
    decltype(std::declval<basic_json_t&>().get_impl_ptr(std::declval<PointerType>()))
21703     {
21704         // delegate the call to get_impl_ptr<>()
21705         return get_impl_ptr(static_cast<PointerType>(nullptr));
21706     }
21707
21710     template < typename PointerType, typename std::enable_if <
21711                    std::is_pointer<PointerType>::value&&
21712                    std::is_const<typename std::remove_pointer<PointerType>::type>::value, int >::type
    = 0 >
21713     constexpr auto get_ptr() const noexcept -> decltype(std::declval<const
    basic_json_t&>().get_impl_ptr(std::declval<PointerType>()))
21714     {
21715         // delegate the call to get_impl_ptr<>() const
21716         return get_impl_ptr(static_cast<PointerType>(nullptr));
21717     }
21718
21719   private:
21758     template < typename ValueType,
```

```
21759                 detail::enable_if_t <
21760                     detail::is_default_constructible<ValueType>::value&&
21761                     detail::has_from_json<basic_json_t, ValueType>::value,
21762                     int > = 0 >
21763     ValueType get_impl(detail::priority_tag<0> /*unused*/) const noexcept(noexcept(
21764             JSONSerializer<ValueType>::from_json(std::declval<const basic_json_t&>(),
      std::declval<ValueType&>())))
21765     {
21766         auto ret = ValueType();
21767         JSONSerializer<ValueType>::from_json(*this, ret);
21768         return ret;
21769     }
21770
21801     template < typename ValueType,
21802                 detail::enable_if_t <
21803                     detail::has_non_default_from_json<basic_json_t, ValueType>::value,
21804                     int > = 0 >
21805     ValueType get_impl(detail::priority_tag<1> /*unused*/) const noexcept(noexcept(
21806             JSONSerializer<ValueType>::from_json(std::declval<const basic_json_t&>())))
21807     {
21808         return JSONSerializer<ValueType>::from_json(*this);
21809     }
21810
21826     template < typename BasicJsonType,
21827                 detail::enable_if_t <
21828                     detail::is_basic_json<BasicJsonType>::value,
21829                     int > = 0 >
21830     BasicJsonType get_impl(detail::priority_tag<2> /*unused*/) const
21831     {
21832         return *this;
21833     }
21834
21849     template<typename BasicJsonType,
21850              detail::enable_if_t<
21851                  std::is_same<BasicJsonType, basic_json_t>::value,
21852                  int> = 0>
21853     basic_json get_impl(detail::priority_tag<3> /*unused*/) const
21854     {
21855         return *this;
21856     }
21857
21862     template<typename PointerType,
21863              detail::enable_if_t<
21864                  std::is_pointer<PointerType>::value,
21865                  int> = 0>
21866     constexpr auto get_impl(detail::priority_tag<4> /*unused*/) const noexcept
21867     -> decltype(std::declval<const basic_json_t&>().template get_ptr<PointerType>())
21868     {
21869         // delegate the call to get_ptr
21870         return get_ptr<PointerType>();
21871     }
21872
21873   public:
21897     template < typename ValueTypeCV, typename ValueType = detail::uncvref_t<ValueTypeCV»
21898 #if defined(JSON_HAS_CPP_14)
21899     constexpr
21900 #endif
21901     auto get() const noexcept(
21902     noexcept(std::declval<const basic_json_t&>().template get_impl<ValueType>(detail::priority_tag<4>
      {})))
21903     -> decltype(std::declval<const basic_json_t&>().template
      get_impl<ValueType>(detail::priority_tag<4> {}))
21904     {
21905         // we cannot static_assert on ValueTypeCV being non-const, because
21906         // there is support for get<const basic_json_t>(), which is why we
21907         // still need the uncvref
21908         static_assert(!std::is_reference<ValueTypeCV>::value,
21909                       "get() cannot be used with reference types, you might want to use get_ref()");
21910         return get_impl<ValueType>(detail::priority_tag<4> {});
21911     }
21912
21940     template<typename PointerType, typename std::enable_if<
21941                  std::is_pointer<PointerType>::value, int>::type = 0>
21942     auto get() noexcept -> decltype(std::declval<basic_json_t&>().template get_ptr<PointerType>())
21943     {
21944         // delegate the call to get_ptr
21945         return get_ptr<PointerType>();
21946     }
21947
21950     template < typename ValueType,
21951                 detail::enable_if_t <
21952                     !detail::is_basic_json<ValueType>::value&&
21953                     detail::has_from_json<basic_json_t, ValueType>::value,
21954                     int > = 0 >
21955     ValueType & get_to(ValueType& v) const noexcept(noexcept(
21956             JSONSerializer<ValueType>::from_json(std::declval<const basic_json_t&>(), v)))
21957     {
```

```
21958         JSONSerializer<ValueType>::from_json(*this, v);
21959         return v;
21960     }
21961
21962     // specialization to allow calling get_to with a basic_json value
21963     // see https://github.com/nlohmann/json/issues/2175
21964     template<typename ValueType,
21965              detail::enable_if_t <
21966                  detail::is_basic_json<ValueType>::value,
21967                  int> = 0>
21968     ValueType & get_to(ValueType& v) const
21969     {
21970         v = *this;
21971         return v;
21972     }
21973
21974     template <
21975         typename T, std::size_t N,
21976         typename Array = T (&)[N], //
      NOLINT(cppcoreguidelines-avoid-c-arrays,hicpp-avoid-c-arrays,modernize-avoid-c-arrays)
21977         detail::enable_if_t <
21978             detail::has_from_json<basic_json_t, Array>::value, int > = 0 >
21979     Array get_to(T (&v)[N]) const //
      NOLINT(cppcoreguidelines-avoid-c-arrays,hicpp-avoid-c-arrays,modernize-avoid-c-arrays)
21980     noexcept(noexcept(JSONSerializer<Array>::from_json(
21981                           std::declval<const basic_json_t&>(), v)))
21982     {
21983         JSONSerializer<Array>::from_json(*this, v);
21984         return v;
21985     }
21986
21989     template<typename ReferenceType, typename std::enable_if<
21990                  std::is_reference<ReferenceType>::value, int>::type = 0>
21991     ReferenceType get_ref()
21992     {
21993         // delegate call to get_ref_impl
21994         return get_ref_impl<ReferenceType>(*this);
21995     }
21996
21999     template < typename ReferenceType, typename std::enable_if <
22000                    std::is_reference<ReferenceType>::value&&
22001                    std::is_const<typename std::remove_reference<ReferenceType>::type>::value, int
      >::type = 0 >
22002     ReferenceType get_ref() const
22003     {
22004         // delegate call to get_ref_impl
22005         return get_ref_impl<ReferenceType>(*this);
22006     }
22007
22037     template < typename ValueType, typename std::enable_if <
22038                    detail::conjunction <
22039                        detail::negation<std::is_pointer<ValueType>>,
22040                        detail::negation<std::is_same<ValueType, std::nullptr_t>>,
22041                        detail::negation<std::is_same<ValueType, detail::json_ref<basic_json>>»,
22042
      detail::negation<std::is_same<ValueType, typename string_t::value_type>>,
22043                                          detail::negation<detail::is_basic_json<ValueType>>,
22044
      detail::negation<std::is_same<ValueType, std::initializer_list<typename string_t::value_type>»,
22045 #if defined(JSON_HAS_CPP_17) && (defined(__GNUC__) || (defined(_MSC_VER) && _MSC_VER >= 1910 &&
      _MSC_VER <= 1914))
22046
      detail::negation<std::is_same<ValueType, std::string_view>>,
22047 #endif
22048 #if defined(JSON_HAS_CPP_17) && JSON_HAS_STATIC_RTTI
22049                                                  detail::negation<std::is_same<ValueType, std::any>>,
22050 #endif
22051
      detail::is_detected_lazy<detail::get_template_function, const basic_json_t&, ValueType>
22052                                                  >::value, int >::type = 0 >
22053                                          JSON_EXPLICIT operator ValueType() const
22054     {
22055         // delegate the call to get<>() const
22056         return get<ValueType>();
22057     }
22058
22061     binary_t& get_binary()
22062     {
22063         if (!is_binary())
22064         {
22065             JSON_THROW(type_error::create(302, detail::concat("type must be binary, but is ",
      type_name()), this));
22066         }
22067
22068         return *get_ptr<binary_t*>();
22069     }
22070
```

```
22073     const binary_t& get_binary() const
22074     {
22075         if (!is_binary())
22076         {
22077             JSON_THROW(type_error::create(302, detail::concat("type must be binary, but is ",
      type_name()), this));
22078         }
22079
22080         return *get_ptr<const binary_t*>();
22081     }
22082
22084
22086     // element access //
22088
22092
22095     reference at(size_type idx)
22096     {
22097         // at only works for arrays
22098         if (JSON_HEDLEY_LIKELY(is_array()))
22099         {
22100             JSON_TRY
22101             {
22102                 return set_parent(m_data.m_value.array->at(idx));
22103             }
22104             JSON_CATCH (std::out_of_range&)
22105             {
22106                 // create a better exception explanation
22107                 JSON_THROW(out_of_range::create(401, detail::concat("array index ",
      std::to_string(idx), " is out of range"), this));
22108             } // cppcheck-suppress[missingReturn]
22109         }
22110         else
22111         {
22112             JSON_THROW(type_error::create(304, detail::concat("cannot use at() with ", type_name()),
      this));
22113         }
22114     }
22115
22118     const_reference at(size_type idx) const
22119     {
22120         // at only works for arrays
22121         if (JSON_HEDLEY_LIKELY(is_array()))
22122         {
22123             JSON_TRY
22124             {
22125                 return m_data.m_value.array->at(idx);
22126             }
22127             JSON_CATCH (std::out_of_range&)
22128             {
22129                 // create a better exception explanation
22130                 JSON_THROW(out_of_range::create(401, detail::concat("array index ",
      std::to_string(idx), " is out of range"), this));
22131             } // cppcheck-suppress[missingReturn]
22132         }
22133         else
22134         {
22135             JSON_THROW(type_error::create(304, detail::concat("cannot use at() with ", type_name()),
      this));
22136         }
22137     }
22138
22141     reference at(const typename object_t::key_type& key)
22142     {
22143         // at only works for objects
22144         if (JSON_HEDLEY_UNLIKELY(!is_object()))
22145         {
22146             JSON_THROW(type_error::create(304, detail::concat("cannot use at() with ", type_name()),
      this));
22147         }
22148
22149         auto it = m_data.m_value.object->find(key);
22150         if (it == m_data.m_value.object->end())
22151         {
22152             JSON_THROW(out_of_range::create(403, detail::concat("key '", key, "' not found"), this));
22153         }
22154         return set_parent(it->second);
22155     }
22156
22159     template<class KeyType, detail::enable_if_t<
22160                  detail::is_usable_as_basic_json_key_type<basic_json_t, KeyType>::value, int> = 0>
22161     reference at(KeyType && key)
22162     {
22163         // at only works for objects
22164         if (JSON_HEDLEY_UNLIKELY(!is_object()))
22165         {
22166             JSON_THROW(type_error::create(304, detail::concat("cannot use at() with ", type_name()),
      this));
```

```
22167         }
22168
22169         auto it = m_data.m_value.object->find(std::forward<KeyType>(key));
22170         if (it == m_data.m_value.object->end())
22171         {
22172             JSON_THROW(out_of_range::create(403, detail::concat("key '",
      string_t(std::forward<KeyType>(key)), "' not found"), this));
22173         }
22174         return set_parent(it->second);
22175     }
22176
22179     const_reference at(const typename object_t::key_type& key) const
22180     {
22181         // at only works for objects
22182         if (JSON_HEDLEY_UNLIKELY(!is_object()))
22183         {
22184             JSON_THROW(type_error::create(304, detail::concat("cannot use at() with ", type_name()),
      this));
22185         }
22186
22187         auto it = m_data.m_value.object->find(key);
22188         if (it == m_data.m_value.object->end())
22189         {
22190             JSON_THROW(out_of_range::create(403, detail::concat("key '", key, "' not found"), this));
22191         }
22192         return it->second;
22193     }
22194
22197     template<class KeyType, detail::enable_if_t<
22198                  detail::is_usable_as_basic_json_key_type<basic_json_t, KeyType>::value, int> = 0>
22199     const_reference at(KeyType && key) const
22200     {
22201         // at only works for objects
22202         if (JSON_HEDLEY_UNLIKELY(!is_object()))
22203         {
22204             JSON_THROW(type_error::create(304, detail::concat("cannot use at() with ", type_name()),
      this));
22205         }
22206
22207         auto it = m_data.m_value.object->find(std::forward<KeyType>(key));
22208         if (it == m_data.m_value.object->end())
22209         {
22210             JSON_THROW(out_of_range::create(403, detail::concat("key '",
      string_t(std::forward<KeyType>(key)), "' not found"), this));
22211         }
22212         return it->second;
22213     }
22214
22217     reference operator[](size_type idx)
22218     {
22219         // implicitly convert a null value to an empty array
22220         if (is_null())
22221         {
22222             m_data.m_type = value_t::array;
22223             m_data.m_value.array = create<array_t>();
22224             assert_invariant();
22225         }
22226
22227         // operator[] only works for arrays
22228         if (JSON_HEDLEY_LIKELY(is_array()))
22229         {
22230             // fill up the array with null values if given idx is outside the range
22231             if (idx >= m_data.m_value.array->size())
22232             {
22233 #if JSON_DIAGNOSTICS
22234                 // remember array size & capacity before resizing
22235                 const auto old_size = m_data.m_value.array->size();
22236                 const auto old_capacity = m_data.m_value.array->capacity();
22237 #endif
22238                 m_data.m_value.array->resize(idx + 1);
22239
22240 #if JSON_DIAGNOSTICS
22241                 if (JSON_HEDLEY_UNLIKELY(m_data.m_value.array->capacity() != old_capacity))
22242                 {
22243                     // capacity has changed: update all parents
22244                     set_parents();
22245                 }
22246                 else
22247                 {
22248                     // set parent for values added above
22249                     set_parents(begin() + static_cast<typename iterator::difference_type>(old_size),
      static_cast<typename iterator::difference_type>(idx + 1 - old_size));
22250                 }
22251 #endif
22252                 assert_invariant();
22253             }
22254
```

```
22255             return m_data.m_value.array->operator[](idx);
22256         }
22257 
22258         JSON_THROW(type_error::create(305, detail::concat("cannot use operator[] with a numeric
      argument with ", type_name()), this));
22259     }
22260 
22263     const_reference operator[](size_type idx) const
22264     {
22265         // const operator[] only works for arrays
22266         if (JSON_HEDLEY_LIKELY(is_array()))
22267         {
22268             return m_data.m_value.array->operator[](idx);
22269         }
22270 
22271         JSON_THROW(type_error::create(305, detail::concat("cannot use operator[] with a numeric
      argument with ", type_name()), this));
22272     }
22273 
22276     reference operator[](typename object_t::key_type key) //
      NOLINT(performance-unnecessary-value-param)
22277     {
22278         // implicitly convert a null value to an empty object
22279         if (is_null())
22280         {
22281             m_data.m_type = value_t::object;
22282             m_data.m_value.object = create<object_t>();
22283             assert_invariant();
22284         }
22285 
22286         // operator[] only works for objects
22287         if (JSON_HEDLEY_LIKELY(is_object()))
22288         {
22289             auto result = m_data.m_value.object->emplace(std::move(key), nullptr);
22290             return set_parent(result.first->second);
22291         }
22292 
22293         JSON_THROW(type_error::create(305, detail::concat("cannot use operator[] with a string
      argument with ", type_name()), this));
22294     }
22295 
22298     const_reference operator[](const typename object_t::key_type& key) const
22299     {
22300         // const operator[] only works for objects
22301         if (JSON_HEDLEY_LIKELY(is_object()))
22302         {
22303             auto it = m_data.m_value.object->find(key);
22304             JSON_ASSERT(it != m_data.m_value.object->end());
22305             return it->second;
22306         }
22307 
22308         JSON_THROW(type_error::create(305, detail::concat("cannot use operator[] with a string
      argument with ", type_name()), this));
22309     }
22310 
22311     // these two functions resolve a (const) char * ambiguity affecting Clang and MSVC
22312     // (they seemingly cannot be constrained to resolve the ambiguity)
22313     template<typename T>
22314     reference operator[](T* key)
22315     {
22316         return operator[](typename object_t::key_type(key));
22317     }
22318 
22319     template<typename T>
22320     const_reference operator[](T* key) const
22321     {
22322         return operator[](typename object_t::key_type(key));
22323     }
22324 
22327     template<class KeyType, detail::enable_if_t<
22328                  detail::is_usable_as_basic_json_key_type<basic_json_t, KeyType>::value, int > = 0 >
22329     reference operator[](KeyType && key)
22330     {
22331         // implicitly convert a null value to an empty object
22332         if (is_null())
22333         {
22334             m_data.m_type = value_t::object;
22335             m_data.m_value.object = create<object_t>();
22336             assert_invariant();
22337         }
22338 
22339         // operator[] only works for objects
22340         if (JSON_HEDLEY_LIKELY(is_object()))
22341         {
22342             auto result = m_data.m_value.object->emplace(std::forward<KeyType>(key), nullptr);
22343             return set_parent(result.first->second);
22344         }
```

```
22345
22346         JSON_THROW(type_error::create(305, detail::concat("cannot use operator[] with a string
      argument with ", type_name()), this));
22347     }
22348
22351     template<class KeyType, detail::enable_if_t<
22352                  detail::is_usable_as_basic_json_key_type<basic_json_t, KeyType>::value, int > = 0 >
22353     const_reference operator[](KeyType && key) const
22354     {
22355         // const operator[] only works for objects
22356         if (JSON_HEDLEY_LIKELY(is_object()))
22357         {
22358             auto it = m_data.m_value.object->find(std::forward<KeyType>(key));
22359             JSON_ASSERT(it != m_data.m_value.object->end());
22360             return it->second;
22361         }
22362
22363         JSON_THROW(type_error::create(305, detail::concat("cannot use operator[] with a string
      argument with ", type_name()), this));
22364     }
22365
22366   private:
22367     template<typename KeyType>
22368     using is_comparable_with_object_key = detail::is_comparable <
22369         object_comparator_t, const typename object_t::key_type&, KeyType >;
22370
22371     template<typename ValueType>
22372     using value_return_type = std::conditional <
22373         detail::is_c_string_uncvref<ValueType>::value,
22374         string_t, typename std::decay<ValueType>::type >;
22375
22376   public:
22379     template < class ValueType, detail::enable_if_t <
22380                    !detail::is_transparent<object_comparator_t>::value
22381                    && detail::is_getable<basic_json_t, ValueType>::value
22382                    && !std::is_same<value_t, detail::uncvref_t<ValueType»::value, int > = 0 >
22383     ValueType value(const typename object_t::key_type& key, const ValueType& default_value) const
22384     {
22385         // value only works for objects
22386         if (JSON_HEDLEY_LIKELY(is_object()))
22387         {
22388             // If 'key' is found, return its value. Otherwise, return `default_value'.
22389             const auto it = find(key);
22390             if (it != end())
22391             {
22392                 return it->template get<ValueType>();
22393             }
22394
22395             return default_value;
22396         }
22397
22398         JSON_THROW(type_error::create(306, detail::concat("cannot use value() with ", type_name()),
      this));
22399     }
22400
22403     template < class ValueType, class ReturnType = typename value_return_type<ValueType>::type,
22404                detail::enable_if_t <
22405                    !detail::is_transparent<object_comparator_t>::value
22406                    && detail::is_getable<basic_json_t, ReturnType>::value
22407                    && !std::is_same<value_t, detail::uncvref_t<ValueType»::value, int > = 0 >
22408     ReturnType value(const typename object_t::key_type& key, ValueType && default_value) const
22409     {
22410         // value only works for objects
22411         if (JSON_HEDLEY_LIKELY(is_object()))
22412         {
22413             // If 'key' is found, return its value. Otherwise, return `default_value'.
22414             const auto it = find(key);
22415             if (it != end())
22416             {
22417                 return it->template get<ReturnType>();
22418             }
22419
22420             return std::forward<ValueType>(default_value);
22421         }
22422
22423         JSON_THROW(type_error::create(306, detail::concat("cannot use value() with ", type_name()),
      this));
22424     }
22425
22428     template < class ValueType, class KeyType, detail::enable_if_t <
22429                    detail::is_transparent<object_comparator_t>::value
22430                    && !detail::is_json_pointer<KeyType>::value
22431                    && is_comparable_with_object_key<KeyType>::value
22432                    && detail::is_getable<basic_json_t, ValueType>::value
22433                    && !std::is_same<value_t, detail::uncvref_t<ValueType»::value, int > = 0 >
22434     ValueType value(KeyType && key, const ValueType& default_value) const
22435     {
```

```
22436         // value only works for objects
22437         if (JSON_HEDLEY_LIKELY(is_object()))
22438         {
22439             // If 'key' is found, return its value. Otherwise, return `default_value'.
22440             const auto it = find(std::forward<KeyType>(key));
22441             if (it != end())
22442             {
22443                 return it->template get<ValueType>();
22444             }
22445 
22446             return default_value;
22447         }
22448 
22449         JSON_THROW(type_error::create(306, detail::concat("cannot use value() with ", type_name()),
      this));
22450     }
22451 
22454     template < class ValueType, class KeyType, class ReturnType = typename
      value_return_type<ValueType>::type,
22455                detail::enable_if_t <
22456                    detail::is_transparent<object_comparator_t>::value
22457                    && !detail::is_json_pointer<KeyType>::value
22458                    && is_comparable_with_object_key<KeyType>::value
22459                    && detail::is_getable<basic_json_t, ReturnType>::value
22460                    && !std::is_same<value_t, detail::uncvref_t<ValueType>>::value, int > = 0 >
22461     ReturnType value(KeyType && key, ValueType && default_value) const
22462     {
22463         // value only works for objects
22464         if (JSON_HEDLEY_LIKELY(is_object()))
22465         {
22466             // If 'key' is found, return its value. Otherwise, return `default_value'.
22467             const auto it = find(std::forward<KeyType>(key));
22468             if (it != end())
22469             {
22470                 return it->template get<ReturnType>();
22471             }
22472 
22473             return std::forward<ValueType>(default_value);
22474         }
22475 
22476         JSON_THROW(type_error::create(306, detail::concat("cannot use value() with ", type_name()),
      this));
22477     }
22478 
22481     template < class ValueType, detail::enable_if_t <
22482                     detail::is_getable<basic_json_t, ValueType>::value
22483                     && !std::is_same<value_t, detail::uncvref_t<ValueType>>::value, int > = 0 >
22484     ValueType value(const json_pointer& ptr, const ValueType& default_value) const
22485     {
22486         // value only works for objects
22487         if (JSON_HEDLEY_LIKELY(is_object()))
22488         {
22489             // If the pointer resolves to a value, return it. Otherwise, return
22490             // 'default_value'.
22491             JSON_TRY
22492             {
22493                 return ptr.get_checked(this).template get<ValueType>();
22494             }
22495             JSON_INTERNAL_CATCH (out_of_range&)
22496             {
22497                 return default_value;
22498             }
22499         }
22500 
22501         JSON_THROW(type_error::create(306, detail::concat("cannot use value() with ", type_name()),
      this));
22502     }
22503 
22506     template < class ValueType, class ReturnType = typename value_return_type<ValueType>::type,
22507                detail::enable_if_t <
22508                    detail::is_getable<basic_json_t, ReturnType>::value
22509                    && !std::is_same<value_t, detail::uncvref_t<ValueType>>::value, int > = 0 >
22510     ReturnType value(const json_pointer& ptr, ValueType && default_value) const
22511     {
22512         // value only works for objects
22513         if (JSON_HEDLEY_LIKELY(is_object()))
22514         {
22515             // If the pointer resolves to a value, return it. Otherwise, return
22516             // 'default_value'.
22517             JSON_TRY
22518             {
22519                 return ptr.get_checked(this).template get<ReturnType>();
22520             }
22521             JSON_INTERNAL_CATCH (out_of_range&)
22522             {
22523                 return std::forward<ValueType>(default_value);
22524             }
```

```
22525         }
22526
22527         JSON_THROW(type_error::create(306, detail::concat("cannot use value() with ", type_name()),
      this));
22528     }
22529
22530     template < class ValueType, class BasicJsonType, detail::enable_if_t <
22531                     detail::is_basic_json<BasicJsonType>::value
22532                     && detail::is_getable<basic_json_t, ValueType>::value
22533                     && !std::is_same<value_t, detail::uncvref_t<ValueType»::value, int > = 0 >
22534     JSON_HEDLEY_DEPRECATED_FOR(3.11.0, basic_json::json_pointer or
      nlohmann::json_pointer<basic_json::string_t>) // NOLINT(readability/alt_tokens)
22535     ValueType value(const ::nlohmann::json_pointer<BasicJsonType>& ptr, const ValueType&
      default_value) const
22536     {
22537         return value(ptr.convert(), default_value);
22538     }
22539
22540     template < class ValueType, class BasicJsonType, class ReturnType = typename
      value_return_type<ValueType>::type,
22541                detail::enable_if_t <
22542                     detail::is_basic_json<BasicJsonType>::value
22543                     && detail::is_getable<basic_json_t, ReturnType>::value
22544                     && !std::is_same<value_t, detail::uncvref_t<ValueType»::value, int > = 0 >
22545     JSON_HEDLEY_DEPRECATED_FOR(3.11.0, basic_json::json_pointer or
      nlohmann::json_pointer<basic_json::string_t>) // NOLINT(readability/alt_tokens)
22546     ReturnType value(const ::nlohmann::json_pointer<BasicJsonType>& ptr, ValueType && default_value)
      const
22547     {
22548         return value(ptr.convert(), std::forward<ValueType>(default_value));
22549     }
22550
22553     reference front()
22554     {
22555         return *begin();
22556     }
22557
22560     const_reference front() const
22561     {
22562         return *cbegin();
22563     }
22564
22567     reference back()
22568     {
22569         auto tmp = end();
22570         --tmp;
22571         return *tmp;
22572     }
22573
22576     const_reference back() const
22577     {
22578         auto tmp = cend();
22579         --tmp;
22580         return *tmp;
22581     }
22582
22585     template < class IteratorType, detail::enable_if_t <
22586                     std::is_same<IteratorType, typename basic_json_t::iterator>::value ||
22587                     std::is_same<IteratorType, typename basic_json_t::const_iterator>::value, int > = 0
      >
22588     IteratorType erase(IteratorType pos) // NOLINT(performance-unnecessary-value-param)
22589     {
22590         // make sure the iterator fits the current value
22591         if (JSON_HEDLEY_UNLIKELY(this != pos.m_object))
22592         {
22593             JSON_THROW(invalid_iterator::create(202, "iterator does not fit current value", this));
22594         }
22595
22596         IteratorType result = end();
22597
22598         switch (m_data.m_type)
22599         {
22600             case value_t::boolean:
22601             case value_t::number_float:
22602             case value_t::number_integer:
22603             case value_t::number_unsigned:
22604             case value_t::string:
22605             case value_t::binary:
22606             {
22607                 if (JSON_HEDLEY_UNLIKELY(!pos.m_it.primitive_iterator.is_begin()))
22608                 {
22609                     JSON_THROW(invalid_iterator::create(205, "iterator out of range", this));
22610                 }
22611
22612                 if (is_string())
22613                 {
22614                     AllocatorType<string_t> alloc;
```

```
22615                     std::allocator_traits<decltype(alloc)>::destroy(alloc, m_data.m_value.string);
22616                     std::allocator_traits<decltype(alloc)>::deallocate(alloc, m_data.m_value.string,
    1);
22617                     m_data.m_value.string = nullptr;
22618                 }
22619                 else if (is_binary())
22620                 {
22621                     AllocatorType<binary_t> alloc;
22622                     std::allocator_traits<decltype(alloc)>::destroy(alloc, m_data.m_value.binary);
22623                     std::allocator_traits<decltype(alloc)>::deallocate(alloc, m_data.m_value.binary,
    1);
22624                     m_data.m_value.binary = nullptr;
22625                 }
22626 
22627                 m_data.m_type = value_t::null;
22628                 assert_invariant();
22629                 break;
22630             }
22631 
22632             case value_t::object:
22633             {
22634                 result.m_it.object_iterator = m_data.m_value.object->erase(pos.m_it.object_iterator);
22635                 break;
22636             }
22637 
22638             case value_t::array:
22639             {
22640                 result.m_it.array_iterator = m_data.m_value.array->erase(pos.m_it.array_iterator);
22641                 break;
22642             }
22643 
22644             case value_t::null:
22645             case value_t::discarded:
22646             default:
22647                 JSON_THROW(type_error::create(307, detail::concat("cannot use erase() with ",
    type_name()), this));
22648         }
22649 
22650         return result;
22651     }
22652 
22655     template < class IteratorType, detail::enable_if_t <
22656                    std::is_same<IteratorType, typename basic_json_t::iterator>::value ||
22657                    std::is_same<IteratorType, typename basic_json_t::const_iterator>::value, int > = 0
    >
22658     IteratorType erase(IteratorType first, IteratorType last) //
    NOLINT(performance-unnecessary-value-param)
22659     {
22660         // make sure the iterator fits the current value
22661         if (JSON_HEDLEY_UNLIKELY(this != first.m_object || this != last.m_object))
22662         {
22663             JSON_THROW(invalid_iterator::create(203, "iterators do not fit current value", this));
22664         }
22665 
22666         IteratorType result = end();
22667 
22668         switch (m_data.m_type)
22669         {
22670             case value_t::boolean:
22671             case value_t::number_float:
22672             case value_t::number_integer:
22673             case value_t::number_unsigned:
22674             case value_t::string:
22675             case value_t::binary:
22676             {
22677                 if (JSON_HEDLEY_LIKELY(!first.m_it.primitive_iterator.is_begin()
22678                                        || !last.m_it.primitive_iterator.is_end()))
22679                 {
22680                     JSON_THROW(invalid_iterator::create(204, "iterators out of range", this));
22681                 }
22682 
22683                 if (is_string())
22684                 {
22685                     AllocatorType<string_t> alloc;
22686                     std::allocator_traits<decltype(alloc)>::destroy(alloc, m_data.m_value.string);
22687                     std::allocator_traits<decltype(alloc)>::deallocate(alloc, m_data.m_value.string,
    1);
22688                     m_data.m_value.string = nullptr;
22689                 }
22690                 else if (is_binary())
22691                 {
22692                     AllocatorType<binary_t> alloc;
22693                     std::allocator_traits<decltype(alloc)>::destroy(alloc, m_data.m_value.binary);
22694                     std::allocator_traits<decltype(alloc)>::deallocate(alloc, m_data.m_value.binary,
    1);
22695                     m_data.m_value.binary = nullptr;
22696                 }
```

```
22697
22698                 m_data.m_type = value_t::null;
22699                 assert_invariant();
22700                 break;
22701             }
22702
22703             case value_t::object:
22704             {
22705                 result.m_it.object_iterator = m_data.m_value.object->erase(first.m_it.object_iterator,
22706                                                 last.m_it.object_iterator);
22707                 break;
22708             }
22709
22710             case value_t::array:
22711             {
22712                 result.m_it.array_iterator = m_data.m_value.array->erase(first.m_it.array_iterator,
22713                                                last.m_it.array_iterator);
22714                 break;
22715             }
22716
22717             case value_t::null:
22718             case value_t::discarded:
22719             default:
22720                 JSON_THROW(type_error::create(307, detail::concat("cannot use erase() with ",
    type_name()), this));
22721         }
22722
22723         return result;
22724     }
22725
22726   private:
22727     template < typename KeyType, detail::enable_if_t <
22728                    detail::has_erase_with_key_type<basic_json_t, KeyType>::value, int > = 0 >
22729     size_type erase_internal(KeyType && key)
22730     {
22731         // this erase only works for objects
22732         if (JSON_HEDLEY_UNLIKELY(!is_object()))
22733         {
22734             JSON_THROW(type_error::create(307, detail::concat("cannot use erase() with ",
    type_name()), this));
22735         }
22736
22737         return m_data.m_value.object->erase(std::forward<KeyType>(key));
22738     }
22739
22740     template < typename KeyType, detail::enable_if_t <
22741                    !detail::has_erase_with_key_type<basic_json_t, KeyType>::value, int > = 0 >
22742     size_type erase_internal(KeyType && key)
22743     {
22744         // this erase only works for objects
22745         if (JSON_HEDLEY_UNLIKELY(!is_object()))
22746         {
22747             JSON_THROW(type_error::create(307, detail::concat("cannot use erase() with ",
    type_name()), this));
22748         }
22749
22750         const auto it = m_data.m_value.object->find(std::forward<KeyType>(key));
22751         if (it != m_data.m_value.object->end())
22752         {
22753             m_data.m_value.object->erase(it);
22754             return 1;
22755         }
22756         return 0;
22757     }
22758
22759   public:
22760
22763     size_type erase(const typename object_t::key_type& key)
22764     {
22765         // the indirection via erase_internal() is added to avoid making this
22766         // function a template and thus de-rank it during overload resolution
22767         return erase_internal(key);
22768     }
22769
22772     template<class KeyType, detail::enable_if_t<
22773                   detail::is_usable_as_basic_json_key_type<basic_json_t, KeyType>::value, int> = 0>
22774     size_type erase(KeyType && key)
22775     {
22776         return erase_internal(std::forward<KeyType>(key));
22777     }
22778
22781     void erase(const size_type idx)
22782     {
22783         // this erase only works for arrays
22784         if (JSON_HEDLEY_LIKELY(is_array()))
22785         {
22786             if (JSON_HEDLEY_UNLIKELY(idx >= size()))
```

```
22787             {
22788                 JSON_THROW(out_of_range::create(401, detail::concat("array index ",
      std::to_string(idx), " is out of range"), this));
22789             }
22790
22791             m_data.m_value.array->erase(m_data.m_value.array->begin() +
      static_cast<difference_type>(idx));
22792         }
22793         else
22794         {
22795             JSON_THROW(type_error::create(307, detail::concat("cannot use erase() with ",
      type_name()), this));
22796         }
22797     }
22798
22800
22802     // lookup //
22804
22807
22810     iterator find(const typename object_t::key_type& key)
22811     {
22812         auto result = end();
22813
22814         if (is_object())
22815         {
22816             result.m_it.object_iterator = m_data.m_value.object->find(key);
22817         }
22818
22819         return result;
22820     }
22821
22824     const_iterator find(const typename object_t::key_type& key) const
22825     {
22826         auto result = cend();
22827
22828         if (is_object())
22829         {
22830             result.m_it.object_iterator = m_data.m_value.object->find(key);
22831         }
22832
22833         return result;
22834     }
22835
22838     template<class KeyType, detail::enable_if_t<
22839                  detail::is_usable_as_basic_json_key_type<basic_json_t, KeyType>::value, int> = 0>
22840     iterator find(KeyType && key)
22841     {
22842         auto result = end();
22843
22844         if (is_object())
22845         {
22846             result.m_it.object_iterator = m_data.m_value.object->find(std::forward<KeyType>(key));
22847         }
22848
22849         return result;
22850     }
22851
22854     template<class KeyType, detail::enable_if_t<
22855                  detail::is_usable_as_basic_json_key_type<basic_json_t, KeyType>::value, int> = 0>
22856     const_iterator find(KeyType && key) const
22857     {
22858         auto result = cend();
22859
22860         if (is_object())
22861         {
22862             result.m_it.object_iterator = m_data.m_value.object->find(std::forward<KeyType>(key));
22863         }
22864
22865         return result;
22866     }
22867
22870     size_type count(const typename object_t::key_type& key) const
22871     {
22872         // return 0 for all nonobject types
22873         return is_object() ? m_data.m_value.object->count(key) : 0;
22874     }
22875
22878     template<class KeyType, detail::enable_if_t<
22879                  detail::is_usable_as_basic_json_key_type<basic_json_t, KeyType>::value, int> = 0>
22880     size_type count(KeyType && key) const
22881     {
22882         // return 0 for all nonobject types
22883         return is_object() ? m_data.m_value.object->count(std::forward<KeyType>(key)) : 0;
22884     }
22885
22888     bool contains(const typename object_t::key_type& key) const
22889     {
```

```
22890         return is_object() && m_data.m_value.object->find(key) != m_data.m_value.object->end();
22891     }
22892
22895     template<class KeyType, detail::enable_if_t<
22896                   detail::is_usable_as_basic_json_key_type<basic_json_t, KeyType>::value, int> = 0>
22897     bool contains(KeyType && key) const
22898     {
22899         return is_object() && m_data.m_value.object->find(std::forward<KeyType>(key)) !=
      m_data.m_value.object->end();
22900     }
22901
22904     bool contains(const json_pointer& ptr) const
22905     {
22906         return ptr.contains(this);
22907     }
22908
22909     template<typename BasicJsonType, detail::enable_if_t<detail::is_basic_json<BasicJsonType>::value,
      int> = 0>
22910     JSON_HEDLEY_DEPRECATED_FOR(3.11.0, basic_json::json_pointer or
      nlohmann::json_pointer<basic_json::string_t>) // NOLINT(readability/alt_tokens)
22911     bool contains(const typename ::nlohmann::json_pointer<BasicJsonType>& ptr) const
22912     {
22913         return ptr.contains(this);
22914     }
22915
22917
22919     // iterators //
22921
22924
22927     iterator begin() noexcept
22928     {
22929         iterator result(this);
22930         result.set_begin();
22931         return result;
22932     }
22933
22936     const_iterator begin() const noexcept
22937     {
22938         return cbegin();
22939     }
22940
22943     const_iterator cbegin() const noexcept
22944     {
22945         const_iterator result(this);
22946         result.set_begin();
22947         return result;
22948     }
22949
22952     iterator end() noexcept
22953     {
22954         iterator result(this);
22955         result.set_end();
22956         return result;
22957     }
22958
22961     const_iterator end() const noexcept
22962     {
22963         return cend();
22964     }
22965
22968     const_iterator cend() const noexcept
22969     {
22970         const_iterator result(this);
22971         result.set_end();
22972         return result;
22973     }
22974
22977     reverse_iterator rbegin() noexcept
22978     {
22979         return reverse_iterator(end());
22980     }
22981
22984     const_reverse_iterator rbegin() const noexcept
22985     {
22986         return crbegin();
22987     }
22988
22991     reverse_iterator rend() noexcept
22992     {
22993         return reverse_iterator(begin());
22994     }
22995
22998     const_reverse_iterator rend() const noexcept
22999     {
23000         return crend();
23001     }
23002
```

```
23005     const_reverse_iterator crbegin() const noexcept
23006     {
23007         return const_reverse_iterator(cend());
23008     }
23009
23012     const_reverse_iterator crend() const noexcept
23013     {
23014         return const_reverse_iterator(cbegin());
23015     }
23016
23017   public:
23023     JSON_HEDLEY_DEPRECATED_FOR(3.1.0, items())
23024     static iteration_proxy<iterator> iterator_wrapper(reference ref) noexcept
23025     {
23026         return ref.items();
23027     }
23028
23034     JSON_HEDLEY_DEPRECATED_FOR(3.1.0, items())
23035     static iteration_proxy<const_iterator> iterator_wrapper(const_reference ref) noexcept
23036     {
23037         return ref.items();
23038     }
23039
23042     iteration_proxy<iterator> items() noexcept
23043     {
23044         return iteration_proxy<iterator>(*this);
23045     }
23046
23049     iteration_proxy<const_iterator> items() const noexcept
23050     {
23051         return iteration_proxy<const_iterator>(*this);
23052     }
23053
23055
23057     // capacity //
23059
23062
23065     bool empty() const noexcept
23066     {
23067         switch (m_data.m_type)
23068         {
23069             case value_t::null:
23070             {
23071                 // null values are empty
23072                 return true;
23073             }
23074
23075             case value_t::array:
23076             {
23077                 // delegate call to array_t::empty()
23078                 return m_data.m_value.array->empty();
23079             }
23080
23081             case value_t::object:
23082             {
23083                 // delegate call to object_t::empty()
23084                 return m_data.m_value.object->empty();
23085             }
23086
23087             case value_t::string:
23088             case value_t::boolean:
23089             case value_t::number_integer:
23090             case value_t::number_unsigned:
23091             case value_t::number_float:
23092             case value_t::binary:
23093             case value_t::discarded:
23094             default:
23095             {
23096                 // all other types are nonempty
23097                 return false;
23098             }
23099         }
23100     }
23101
23104     size_type size() const noexcept
23105     {
23106         switch (m_data.m_type)
23107         {
23108             case value_t::null:
23109             {
23110                 // null values are empty
23111                 return 0;
23112             }
23113
23114             case value_t::array:
23115             {
23116                 // delegate call to array_t::size()
```

```
23117                 return m_data.m_value.array->size();
23118             }
23119
23120             case value_t::object:
23121             {
23122                 // delegate call to object_t::size()
23123                 return m_data.m_value.object->size();
23124             }
23125
23126             case value_t::string:
23127             case value_t::boolean:
23128             case value_t::number_integer:
23129             case value_t::number_unsigned:
23130             case value_t::number_float:
23131             case value_t::binary:
23132             case value_t::discarded:
23133             default:
23134             {
23135                 // all other types have size 1
23136                 return 1;
23137             }
23138         }
23139     }
23140
23143     size_type max_size() const noexcept
23144     {
23145         switch (m_data.m_type)
23146         {
23147             case value_t::array:
23148             {
23149                 // delegate call to array_t::max_size()
23150                 return m_data.m_value.array->max_size();
23151             }
23152
23153             case value_t::object:
23154             {
23155                 // delegate call to object_t::max_size()
23156                 return m_data.m_value.object->max_size();
23157             }
23158
23159             case value_t::null:
23160             case value_t::string:
23161             case value_t::boolean:
23162             case value_t::number_integer:
23163             case value_t::number_unsigned:
23164             case value_t::number_float:
23165             case value_t::binary:
23166             case value_t::discarded:
23167             default:
23168             {
23169                 // all other types have max_size() == size()
23170                 return size();
23171             }
23172         }
23173     }
23174
23176
23178     // modifiers //
23180
23183
23186     void clear() noexcept
23187     {
23188         switch (m_data.m_type)
23189         {
23190             case value_t::number_integer:
23191             {
23192                 m_data.m_value.number_integer = 0;
23193                 break;
23194             }
23195
23196             case value_t::number_unsigned:
23197             {
23198                 m_data.m_value.number_unsigned = 0;
23199                 break;
23200             }
23201
23202             case value_t::number_float:
23203             {
23204                 m_data.m_value.number_float = 0.0;
23205                 break;
23206             }
23207
23208             case value_t::boolean:
23209             {
23210                 m_data.m_value.boolean = false;
23211                 break;
23212             }
```

```
23213
23214             case value_t::string:
23215             {
23216                 m_data.m_value.string->clear();
23217                 break;
23218             }
23219
23220             case value_t::binary:
23221             {
23222                 m_data.m_value.binary->clear();
23223                 break;
23224             }
23225
23226             case value_t::array:
23227             {
23228                 m_data.m_value.array->clear();
23229                 break;
23230             }
23231
23232             case value_t::object:
23233             {
23234                 m_data.m_value.object->clear();
23235                 break;
23236             }
23237
23238             case value_t::null:
23239             case value_t::discarded:
23240             default:
23241                 break;
23242         }
23243     }
23244
23247     void push_back(basic_json&& val)
23248     {
23249         // push_back only works for null objects or arrays
23250         if (JSON_HEDLEY_UNLIKELY(!(is_null() || is_array())))
23251         {
23252             JSON_THROW(type_error::create(308, detail::concat("cannot use push_back() with ",
      type_name()), this));
23253         }
23254
23255         // transform a null object into an array
23256         if (is_null())
23257         {
23258             m_data.m_type = value_t::array;
23259             m_data.m_value = value_t::array;
23260             assert_invariant();
23261         }
23262
23263         // add the element to the array (move semantics)
23264         const auto old_capacity = m_data.m_value.array->capacity();
23265         m_data.m_value.array->push_back(std::move(val));
23266         set_parent(m_data.m_value.array->back(), old_capacity);
23267         // if val is moved from, basic_json move constructor marks it null, so we do not call the
      destructor
23268     }
23269
23272     reference operator+=(basic_json&& val)
23273     {
23274         push_back(std::move(val));
23275         return *this;
23276     }
23277
23280     void push_back(const basic_json& val)
23281     {
23282         // push_back only works for null objects or arrays
23283         if (JSON_HEDLEY_UNLIKELY(!(is_null() || is_array())))
23284         {
23285             JSON_THROW(type_error::create(308, detail::concat("cannot use push_back() with ",
      type_name()), this));
23286         }
23287
23288         // transform a null object into an array
23289         if (is_null())
23290         {
23291             m_data.m_type = value_t::array;
23292             m_data.m_value = value_t::array;
23293             assert_invariant();
23294         }
23295
23296         // add the element to the array
23297         const auto old_capacity = m_data.m_value.array->capacity();
23298         m_data.m_value.array->push_back(val);
23299         set_parent(m_data.m_value.array->back(), old_capacity);
23300     }
23301
23304     reference operator+=(const basic_json& val)
```

```
23305     {
23306         push_back(val);
23307         return *this;
23308     }
23309
23312     void push_back(const typename object_t::value_type& val)
23313     {
23314         // push_back only works for null objects or objects
23315         if (JSON_HEDLEY_UNLIKELY(!(is_null() || is_object())))
23316         {
23317             JSON_THROW(type_error::create(308, detail::concat("cannot use push_back() with ",
      type_name()), this));
23318         }
23319
23320         // transform a null object into an object
23321         if (is_null())
23322         {
23323             m_data.m_type = value_t::object;
23324             m_data.m_value = value_t::object;
23325             assert_invariant();
23326         }
23327
23328         // add the element to the object
23329         auto res = m_data.m_value.object->insert(val);
23330         set_parent(res.first->second);
23331     }
23332
23335     reference operator+=(const typename object_t::value_type& val)
23336     {
23337         push_back(val);
23338         return *this;
23339     }
23340
23343     void push_back(initializer_list_t init)
23344     {
23345         if (is_object() && init.size() == 2 && (*init.begin())->is_string())
23346         {
23347             basic_json&& key = init.begin()->moved_or_copied();
23348             push_back(typename object_t::value_type(
23349                           std::move(key.get_ref<string_t&>()), (init.begin() +
      1)->moved_or_copied()));
23350         }
23351         else
23352         {
23353             push_back(basic_json(init));
23354         }
23355     }
23356
23359     reference operator+=(initializer_list_t init)
23360     {
23361         push_back(init);
23362         return *this;
23363     }
23364
23367     template<class... Args>
23368     reference emplace_back(Args&& ... args)
23369     {
23370         // emplace_back only works for null objects or arrays
23371         if (JSON_HEDLEY_UNLIKELY(!(is_null() || is_array())))
23372         {
23373             JSON_THROW(type_error::create(311, detail::concat("cannot use emplace_back() with ",
      type_name()), this));
23374         }
23375
23376         // transform a null object into an array
23377         if (is_null())
23378         {
23379             m_data.m_type = value_t::array;
23380             m_data.m_value = value_t::array;
23381             assert_invariant();
23382         }
23383
23384         // add the element to the array (perfect forwarding)
23385         const auto old_capacity = m_data.m_value.array->capacity();
23386         m_data.m_value.array->emplace_back(std::forward<Args>(args)...);
23387         return set_parent(m_data.m_value.array->back(), old_capacity);
23388     }
23389
23392     template<class... Args>
23393     std::pair<iterator, bool> emplace(Args&& ... args)
23394     {
23395         // emplace only works for null objects or arrays
23396         if (JSON_HEDLEY_UNLIKELY(!(is_null() || is_object())))
23397         {
23398             JSON_THROW(type_error::create(311, detail::concat("cannot use emplace() with ",
      type_name()), this));
23399         }
```

```
23400
23401         // transform a null object into an object
23402         if (is_null())
23403         {
23404             m_data.m_type = value_t::object;
23405             m_data.m_value = value_t::object;
23406             assert_invariant();
23407         }
23408
23409         // add the element to the array (perfect forwarding)
23410         auto res = m_data.m_value.object->emplace(std::forward<Args>(args)...);
23411         set_parent(res.first->second);
23412
23413         // create a result iterator and set iterator to the result of emplace
23414         auto it = begin();
23415         it.m_it.object_iterator = res.first;
23416
23417         // return pair of iterator and boolean
23418         return {it, res.second};
23419     }
23420
23424     template<typename... Args>
23425     iterator insert_iterator(const_iterator pos, Args&& ... args) //
    NOLINT(performance-unnecessary-value-param)
23426     {
23427         iterator result(this);
23428         JSON_ASSERT(m_data.m_value.array != nullptr);
23429
23430         auto insert_pos = std::distance(m_data.m_value.array->begin(), pos.m_it.array_iterator);
23431         m_data.m_value.array->insert(pos.m_it.array_iterator, std::forward<Args>(args)...);
23432         result.m_it.array_iterator = m_data.m_value.array->begin() + insert_pos;
23433
23434         // This could have been written as:
23435         // result.m_it.array_iterator = m_data.m_value.array->insert(pos.m_it.array_iterator, cnt,
    val);
23436         // but the return value of insert is missing in GCC 4.8, so it is written this way instead.
23437
23438         set_parents();
23439         return result;
23440     }
23441
23444     iterator insert(const_iterator pos, const basic_json& val) //
    NOLINT(performance-unnecessary-value-param)
23445     {
23446         // insert only works for arrays
23447         if (JSON_HEDLEY_LIKELY(is_array()))
23448         {
23449             // check if iterator pos fits to this JSON value
23450             if (JSON_HEDLEY_UNLIKELY(pos.m_object != this))
23451             {
23452                 JSON_THROW(invalid_iterator::create(202, "iterator does not fit current value",
    this));
23453             }
23454
23455             // insert to array and return iterator
23456             return insert_iterator(pos, val);
23457         }
23458
23459         JSON_THROW(type_error::create(309, detail::concat("cannot use insert() with ", type_name()),
    this));
23460     }
23461
23464     iterator insert(const_iterator pos, basic_json&& val) //
    NOLINT(performance-unnecessary-value-param)
23465     {
23466         return insert(pos, val);
23467     }
23468
23471     iterator insert(const_iterator pos, size_type cnt, const basic_json& val) //
    NOLINT(performance-unnecessary-value-param)
23472     {
23473         // insert only works for arrays
23474         if (JSON_HEDLEY_LIKELY(is_array()))
23475         {
23476             // check if iterator pos fits to this JSON value
23477             if (JSON_HEDLEY_UNLIKELY(pos.m_object != this))
23478             {
23479                 JSON_THROW(invalid_iterator::create(202, "iterator does not fit current value",
    this));
23480             }
23481
23482             // insert to array and return iterator
23483             return insert_iterator(pos, cnt, val);
23484         }
23485
23486         JSON_THROW(type_error::create(309, detail::concat("cannot use insert() with ", type_name()),
    this));
```

```
23487     }
23488
23491     iterator insert(const_iterator pos, const_iterator first, const_iterator last) //
      NOLINT(performance-unnecessary-value-param)
23492     {
23493         // insert only works for arrays
23494         if (JSON_HEDLEY_UNLIKELY(!is_array()))
23495         {
23496             JSON_THROW(type_error::create(309, detail::concat("cannot use insert() with ",
      type_name()), this));
23497         }
23498
23499         // check if iterator pos fits to this JSON value
23500         if (JSON_HEDLEY_UNLIKELY(pos.m_object != this))
23501         {
23502             JSON_THROW(invalid_iterator::create(202, "iterator does not fit current value", this));
23503         }
23504
23505         // check if range iterators belong to the same JSON object
23506         if (JSON_HEDLEY_UNLIKELY(first.m_object != last.m_object))
23507         {
23508             JSON_THROW(invalid_iterator::create(210, "iterators do not fit", this));
23509         }
23510
23511         if (JSON_HEDLEY_UNLIKELY(first.m_object == this))
23512         {
23513             JSON_THROW(invalid_iterator::create(211, "passed iterators may not belong to container",
      this));
23514         }
23515
23516         // insert to array and return iterator
23517         return insert_iterator(pos, first.m_it.array_iterator, last.m_it.array_iterator);
23518     }
23519
23522     iterator insert(const_iterator pos, initializer_list_t ilist) //
      NOLINT(performance-unnecessary-value-param)
23523     {
23524         // insert only works for arrays
23525         if (JSON_HEDLEY_UNLIKELY(!is_array()))
23526         {
23527             JSON_THROW(type_error::create(309, detail::concat("cannot use insert() with ",
      type_name()), this));
23528         }
23529
23530         // check if iterator pos fits to this JSON value
23531         if (JSON_HEDLEY_UNLIKELY(pos.m_object != this))
23532         {
23533             JSON_THROW(invalid_iterator::create(202, "iterator does not fit current value", this));
23534         }
23535
23536         // insert to array and return iterator
23537         return insert_iterator(pos, ilist.begin(), ilist.end());
23538     }
23539
23542     void insert(const_iterator first, const_iterator last) //
      NOLINT(performance-unnecessary-value-param)
23543     {
23544         // insert only works for objects
23545         if (JSON_HEDLEY_UNLIKELY(!is_object()))
23546         {
23547             JSON_THROW(type_error::create(309, detail::concat("cannot use insert() with ",
      type_name()), this));
23548         }
23549
23550         // check if range iterators belong to the same JSON object
23551         if (JSON_HEDLEY_UNLIKELY(first.m_object != last.m_object))
23552         {
23553             JSON_THROW(invalid_iterator::create(210, "iterators do not fit", this));
23554         }
23555
23556         // passed iterators must belong to objects
23557         if (JSON_HEDLEY_UNLIKELY(!first.m_object->is_object()))
23558         {
23559             JSON_THROW(invalid_iterator::create(202, "iterators first and last must point to objects",
      this));
23560         }
23561
23562         m_data.m_value.object->insert(first.m_it.object_iterator, last.m_it.object_iterator);
23563         set_parents();
23564     }
23565
23568     void update(const_reference j, bool merge_objects = false)
23569     {
23570         update(j.begin(), j.end(), merge_objects);
23571     }
23572
23575     void update(const_iterator first, const_iterator last, bool merge_objects = false) //
```

```
      NOLINT(performance-unnecessary-value-param)
23576     {
23577         // implicitly convert a null value to an empty object
23578         if (is_null())
23579         {
23580             m_data.m_type = value_t::object;
23581             m_data.m_value.object = create<object_t>();
23582             assert_invariant();
23583         }
23584 
23585         if (JSON_HEDLEY_UNLIKELY(!is_object()))
23586         {
23587             JSON_THROW(type_error::create(312, detail::concat("cannot use update() with ",
      type_name()), this));
23588         }
23589 
23590         // check if range iterators belong to the same JSON object
23591         if (JSON_HEDLEY_UNLIKELY(first.m_object != last.m_object))
23592         {
23593             JSON_THROW(invalid_iterator::create(210, "iterators do not fit", this));
23594         }
23595 
23596         // passed iterators must belong to objects
23597         if (JSON_HEDLEY_UNLIKELY(!first.m_object->is_object()))
23598         {
23599             JSON_THROW(type_error::create(312, detail::concat("cannot use update() with ",
      first.m_object->type_name()), first.m_object));
23600         }
23601 
23602         for (auto it = first; it != last; ++it)
23603         {
23604             if (merge_objects && it.value().is_object())
23605             {
23606                 auto it2 = m_data.m_value.object->find(it.key());
23607                 if (it2 != m_data.m_value.object->end())
23608                 {
23609                     it2->second.update(it.value(), true);
23610                     continue;
23611                 }
23612             }
23613             m_data.m_value.object->operator[](it.key()) = it.value();
23614 #if JSON_DIAGNOSTICS
23615             m_data.m_value.object->operator[](it.key()).m_parent = this;
23616 #endif
23617         }
23618     }
23619 
23622     void swap(reference other) noexcept (
23623         std::is_nothrow_move_constructible<value_t>::value&&
23624         std::is_nothrow_move_assignable<value_t>::value&&
23625         std::is_nothrow_move_constructible<json_value>::value&& //
      NOLINT(cppcoreguidelines-noexcept-swap,performance-noexcept-swap)
23626         std::is_nothrow_move_assignable<json_value>::value
23627     )
23628     {
23629         std::swap(m_data.m_type, other.m_data.m_type);
23630         std::swap(m_data.m_value, other.m_data.m_value);
23631 
23632         set_parents();
23633         other.set_parents();
23634         assert_invariant();
23635     }
23636 
23639     friend void swap(reference left, reference right) noexcept (
23640         std::is_nothrow_move_constructible<value_t>::value&&
23641         std::is_nothrow_move_assignable<value_t>::value&&
23642         std::is_nothrow_move_constructible<json_value>::value&& //
      NOLINT(cppcoreguidelines-noexcept-swap,performance-noexcept-swap)
23643         std::is_nothrow_move_assignable<json_value>::value
23644     )
23645     {
23646         left.swap(right);
23647     }
23648 
23651     void swap(array_t& other) //
      NOLINT(bugprone-exception-escape,cppcoreguidelines-noexcept-swap,performance-noexcept-swap)
23652     {
23653         // swap only works for arrays
23654         if (JSON_HEDLEY_LIKELY(is_array()))
23655         {
23656             using std::swap;
23657             swap(*(m_data.m_value.array), other);
23658         }
23659         else
23660         {
23661             JSON_THROW(type_error::create(310, detail::concat("cannot use swap(array_t&) with ",
      type_name()), this));
```

```
23662         }
23663     }
23664
23667     void swap(object_t& other) //
      NOLINT(bugprone-exception-escape,cppcoreguidelines-noexcept-swap,performance-noexcept-swap)
23668     {
23669         // swap only works for objects
23670         if (JSON_HEDLEY_LIKELY(is_object()))
23671         {
23672             using std::swap;
23673             swap(*(m_data.m_value.object), other);
23674         }
23675         else
23676         {
23677             JSON_THROW(type_error::create(310, detail::concat("cannot use swap(object_t&) with ",
      type_name()), this));
23678         }
23679     }
23680
23683     void swap(string_t& other) //
      NOLINT(bugprone-exception-escape,cppcoreguidelines-noexcept-swap,performance-noexcept-swap)
23684     {
23685         // swap only works for strings
23686         if (JSON_HEDLEY_LIKELY(is_string()))
23687         {
23688             using std::swap;
23689             swap(*(m_data.m_value.string), other);
23690         }
23691         else
23692         {
23693             JSON_THROW(type_error::create(310, detail::concat("cannot use swap(string_t&) with ",
      type_name()), this));
23694         }
23695     }
23696
23699     void swap(binary_t& other) //
      NOLINT(bugprone-exception-escape,cppcoreguidelines-noexcept-swap,performance-noexcept-swap)
23700     {
23701         // swap only works for strings
23702         if (JSON_HEDLEY_LIKELY(is_binary()))
23703         {
23704             using std::swap;
23705             swap(*(m_data.m_value.binary), other);
23706         }
23707         else
23708         {
23709             JSON_THROW(type_error::create(310, detail::concat("cannot use swap(binary_t&) with ",
      type_name()), this));
23710         }
23711     }
23712
23715     void swap(typename binary_t::container_type& other) // NOLINT(bugprone-exception-escape)
23716     {
23717         // swap only works for strings
23718         if (JSON_HEDLEY_LIKELY(is_binary()))
23719         {
23720             using std::swap;
23721             swap(*(m_data.m_value.binary), other);
23722         }
23723         else
23724         {
23725             JSON_THROW(type_error::create(310, detail::concat("cannot use
      swap(binary_t::container_type&) with ", type_name()), this));
23726         }
23727     }
23728
23730
23732     // lexicographical comparison operators //
23734
23737
23738     // note parentheses around operands are necessary; see
23739     // https://github.com/nlohmann/json/issues/1530
23740 #define JSON_IMPLEMENT_OPERATOR(op, null_result, unordered_result, default_result)
      \
23741     const auto lhs_type = lhs.type();
      \
23742     const auto rhs_type = rhs.type();
      \
23743     \
23744     if (lhs_type == rhs_type) /* NOLINT(readability/braces) */
      \
23745     {
      \
23746         switch (lhs_type)
      \
23747         {
      \
```

```
23748             case value_t::array:
\
23749                 return (*lhs.m_data.m_value.array) op (*rhs.m_data.m_value.array);
\
23750                 \
23751             case value_t::object:
\
23752                 return (*lhs.m_data.m_value.object) op (*rhs.m_data.m_value.object);
\
23753                 \
23754             case value_t::null:
\
23755                 return (null_result);
\
23756                 \
23757             case value_t::string:
\
23758                 return (*lhs.m_data.m_value.string) op (*rhs.m_data.m_value.string);
\
23759                 \
23760             case value_t::boolean:
\
23761                 return (lhs.m_data.m_value.boolean) op (rhs.m_data.m_value.boolean);
\
23762                 \
23763             case value_t::number_integer:
\
23764                 return (lhs.m_data.m_value.number_integer) op (rhs.m_data.m_value.number_integer);
\
23765                 \
23766             case value_t::number_unsigned:
\
23767                 return (lhs.m_data.m_value.number_unsigned) op (rhs.m_data.m_value.number_unsigned);
\
23768                 \
23769             case value_t::number_float:
\
23770                 return (lhs.m_data.m_value.number_float) op (rhs.m_data.m_value.number_float);
\
23771                 \
23772             case value_t::binary:
\
23773                 return (*lhs.m_data.m_value.binary) op (*rhs.m_data.m_value.binary);
\
23774                 \
23775             case value_t::discarded:
\
23776             default:
\
23777                 return (unordered_result);
\
23778         }
\
23779     }
\
23780     else if (lhs_type == value_t::number_integer && rhs_type == value_t::number_float)
\
23781     {
\
23782         return static_cast<number_float_t>(lhs.m_data.m_value.number_integer) op
      rhs.m_data.m_value.number_float;      \
23783     }
\
23784     else if (lhs_type == value_t::number_float && rhs_type == value_t::number_integer)
\
23785     {
\
23786         return lhs.m_data.m_value.number_float op
      static_cast<number_float_t>(rhs.m_data.m_value.number_integer);      \
23787     }
\
23788     else if (lhs_type == value_t::number_unsigned && rhs_type == value_t::number_float)
\
23789     {
\
23790         return static_cast<number_float_t>(lhs.m_data.m_value.number_unsigned) op
      rhs.m_data.m_value.number_float;     \
23791     }
\
23792     else if (lhs_type == value_t::number_float && rhs_type == value_t::number_unsigned)
\
23793     {
\
23794         return lhs.m_data.m_value.number_float op
      static_cast<number_float_t>(rhs.m_data.m_value.number_unsigned);     \
23795     }
\
```

```
23796     else if (lhs_type == value_t::number_unsigned && rhs_type == value_t::number_integer)
      \
23797     {
      \
23798         return static_cast<number_integer_t>(lhs.m_data.m_value.number_unsigned) op
      rhs.m_data.m_value.number_integer; \
23799     }
      \
23800     else if (lhs_type == value_t::number_integer && rhs_type == value_t::number_unsigned)
      \
23801     {
      \
23802         return lhs.m_data.m_value.number_integer op
      static_cast<number_integer_t>(rhs.m_data.m_value.number_unsigned); \
23803     }
      \
23804     else if(compares_unordered(lhs, rhs))\
23805     {\
23806         return (unordered_result);\
23807     }\
23808     \
23809     return (default_result);
23810
23811   JSON_PRIVATE_UNLESS_TESTED:
23812     // returns true if:
23813     // - any operand is NaN and the other operand is of number type
23814     // - any operand is discarded
23815     // in legacy mode, discarded values are considered ordered if
23816     // an operation is computed as an odd number of inverses of others
23817     static bool compares_unordered(const_reference lhs, const_reference rhs, bool inverse = false)
      noexcept
23818     {
23819         if ((lhs.is_number_float() && std::isnan(lhs.m_data.m_value.number_float) && rhs.is_number())
23820                 || (rhs.is_number_float() && std::isnan(rhs.m_data.m_value.number_float) &&
      lhs.is_number()))
23821         {
23822             return true;
23823         }
23824 #if JSON_USE_LEGACY_DISCARDED_VALUE_COMPARISON
23825         return (lhs.is_discarded() || rhs.is_discarded()) && !inverse;
23826 #else
23827         static_cast<void>(inverse);
23828         return lhs.is_discarded() || rhs.is_discarded();
23829 #endif
23830     }
23831
23832   private:
23833     bool compares_unordered(const_reference rhs, bool inverse = false) const noexcept
23834     {
23835         return compares_unordered(*this, rhs, inverse);
23836     }
23837
23838   public:
23839 #if JSON_HAS_THREE_WAY_COMPARISON
23842     bool operator==(const_reference rhs) const noexcept
23843     {
23844 #ifdef __GNUC__
23845 #pragma GCC diagnostic push
23846 #pragma GCC diagnostic ignored "-Wfloat-equal"
23847 #endif
23848         const_reference lhs = *this;
23849         JSON_IMPLEMENT_OPERATOR( ==, true, false, false)
23850 #ifdef __GNUC__
23851 #pragma GCC diagnostic pop
23852 #endif
23853     }
23854
23857     template<typename ScalarType>
23858     requires std::is_scalar_v<ScalarType>
23859     bool operator==(ScalarType rhs) const noexcept
23860     {
23861         return *this == basic_json(rhs);
23862     }
23863
23866     bool operator!=(const_reference rhs) const noexcept
23867     {
23868         if (compares_unordered(rhs, true))
23869         {
23870             return false;
23871         }
23872         return !operator==(rhs);
23873     }
23874
23877     std::partial_ordering operator<=>(const_reference rhs) const noexcept // *NOPAD*
23878     {
23879         const_reference lhs = *this;
23880         // default_result is used if we cannot compare values. In that case,
```

```
23881         // we compare types.
23882         JSON_IMPLEMENT_OPERATOR(<=>, // *NOPAD*
23883                                 std::partial_ordering::equivalent,
23884                                 std::partial_ordering::unordered,
23885                                 lhs_type <=> rhs_type) // *NOPAD*
23886     }
23887
23890     template<typename ScalarType>
23891     requires std::is_scalar_v<ScalarType>
23892     std::partial_ordering operator<=>(ScalarType rhs) const noexcept // *NOPAD*
23893     {
23894         return *this <=> basic_json(rhs); // *NOPAD*
23895     }
23896
23897 #if JSON_USE_LEGACY_DISCARDED_VALUE_COMPARISON
23898     // all operators that are computed as an odd number of inverses of others
23899     // need to be overloaded to emulate the legacy comparison behavior
23900
23903     JSON_HEDLEY_DEPRECATED_FOR(3.11.0, undef JSON_USE_LEGACY_DISCARDED_VALUE_COMPARISON)
23904     bool operator<=(const_reference rhs) const noexcept
23905     {
23906         if (compares_unordered(rhs, true))
23907         {
23908             return false;
23909         }
23910         return !(rhs < *this);
23911     }
23912
23915     template<typename ScalarType>
23916     requires std::is_scalar_v<ScalarType>
23917     bool operator<=(ScalarType rhs) const noexcept
23918     {
23919         return *this <= basic_json(rhs);
23920     }
23921
23924     JSON_HEDLEY_DEPRECATED_FOR(3.11.0, undef JSON_USE_LEGACY_DISCARDED_VALUE_COMPARISON)
23925     bool operator>=(const_reference rhs) const noexcept
23926     {
23927         if (compares_unordered(rhs, true))
23928         {
23929             return false;
23930         }
23931         return !(*this < rhs);
23932     }
23933
23936     template<typename ScalarType>
23937     requires std::is_scalar_v<ScalarType>
23938     bool operator>=(ScalarType rhs) const noexcept
23939     {
23940         return *this >= basic_json(rhs);
23941     }
23942 #endif
23943 #else
23946     friend bool operator==(const_reference lhs, const_reference rhs) noexcept
23947     {
23948 #ifdef __GNUC__
23949 #pragma GCC diagnostic push
23950 #pragma GCC diagnostic ignored "-Wfloat-equal"
23951 #endif
23952         JSON_IMPLEMENT_OPERATOR( ==, true, false, false)
23953 #ifdef __GNUC__
23954 #pragma GCC diagnostic pop
23955 #endif
23956     }
23957
23960     template<typename ScalarType, typename std::enable_if<
23961                  std::is_scalar<ScalarType>::value, int>::type = 0>
23962     friend bool operator==(const_reference lhs, ScalarType rhs) noexcept
23963     {
23964         return lhs == basic_json(rhs);
23965     }
23966
23969     template<typename ScalarType, typename std::enable_if<
23970                  std::is_scalar<ScalarType>::value, int>::type = 0>
23971     friend bool operator==(ScalarType lhs, const_reference rhs) noexcept
23972     {
23973         return basic_json(lhs) == rhs;
23974     }
23975
23978     friend bool operator!=(const_reference lhs, const_reference rhs) noexcept
23979     {
23980         if (compares_unordered(lhs, rhs, true))
23981         {
23982             return false;
23983         }
23984         return !(lhs == rhs);
23985     }
```

```
23986
23989     template<typename ScalarType, typename std::enable_if<
23990                 std::is_scalar<ScalarType>::value, int>::type = 0>
23991     friend bool operator!=(const_reference lhs, ScalarType rhs) noexcept
23992     {
23993         return lhs != basic_json(rhs);
23994     }
23995
23998     template<typename ScalarType, typename std::enable_if<
23999                 std::is_scalar<ScalarType>::value, int>::type = 0>
24000     friend bool operator!=(ScalarType lhs, const_reference rhs) noexcept
24001     {
24002         return basic_json(lhs) != rhs;
24003     }
24004
24007     friend bool operator<(const_reference lhs, const_reference rhs) noexcept
24008     {
24009         // default_result is used if we cannot compare values. In that case,
24010         // we compare types. Note we have to call the operator explicitly,
24011         // because MSVC has problems otherwise.
24012         JSON_IMPLEMENT_OPERATOR( <, false, false, operator<(lhs_type, rhs_type))
24013     }
24014
24017     template<typename ScalarType, typename std::enable_if<
24018                 std::is_scalar<ScalarType>::value, int>::type = 0>
24019     friend bool operator<(const_reference lhs, ScalarType rhs) noexcept
24020     {
24021         return lhs < basic_json(rhs);
24022     }
24023
24026     template<typename ScalarType, typename std::enable_if<
24027                 std::is_scalar<ScalarType>::value, int>::type = 0>
24028     friend bool operator<(ScalarType lhs, const_reference rhs) noexcept
24029     {
24030         return basic_json(lhs) < rhs;
24031     }
24032
24035     friend bool operator<=(const_reference lhs, const_reference rhs) noexcept
24036     {
24037         if (compares_unordered(lhs, rhs, true))
24038         {
24039             return false;
24040         }
24041         return !(rhs < lhs);
24042     }
24043
24046     template<typename ScalarType, typename std::enable_if<
24047                 std::is_scalar<ScalarType>::value, int>::type = 0>
24048     friend bool operator<=(const_reference lhs, ScalarType rhs) noexcept
24049     {
24050         return lhs <= basic_json(rhs);
24051     }
24052
24055     template<typename ScalarType, typename std::enable_if<
24056                 std::is_scalar<ScalarType>::value, int>::type = 0>
24057     friend bool operator<=(ScalarType lhs, const_reference rhs) noexcept
24058     {
24059         return basic_json(lhs) <= rhs;
24060     }
24061
24064     friend bool operator>(const_reference lhs, const_reference rhs) noexcept
24065     {
24066         // double inverse
24067         if (compares_unordered(lhs, rhs))
24068         {
24069             return false;
24070         }
24071         return !(lhs <= rhs);
24072     }
24073
24076     template<typename ScalarType, typename std::enable_if<
24077                 std::is_scalar<ScalarType>::value, int>::type = 0>
24078     friend bool operator>(const_reference lhs, ScalarType rhs) noexcept
24079     {
24080         return lhs > basic_json(rhs);
24081     }
24082
24085     template<typename ScalarType, typename std::enable_if<
24086                 std::is_scalar<ScalarType>::value, int>::type = 0>
24087     friend bool operator>(ScalarType lhs, const_reference rhs) noexcept
24088     {
24089         return basic_json(lhs) > rhs;
24090     }
24091
24094     friend bool operator>=(const_reference lhs, const_reference rhs) noexcept
24095     {
24096         if (compares_unordered(lhs, rhs, true))
```

```
24097         {
24098             return false;
24099         }
24100         return !(lhs < rhs);
24101     }
24102
24105     template<typename ScalarType, typename std::enable_if<
24106                 std::is_scalar<ScalarType>::value, int>::type = 0>
24107     friend bool operator>=(const_reference lhs, ScalarType rhs) noexcept
24108     {
24109         return lhs >= basic_json(rhs);
24110     }
24111
24114     template<typename ScalarType, typename std::enable_if<
24115                 std::is_scalar<ScalarType>::value, int>::type = 0>
24116     friend bool operator>=(ScalarType lhs, const_reference rhs) noexcept
24117     {
24118         return basic_json(lhs) >= rhs;
24119     }
24120 #endif
24121
24122 #undef JSON_IMPLEMENT_OPERATOR
24123
24125
24127     // serialization //
24129
24132 #ifndef JSON_NO_IO
24135     friend std::ostream& operator<<(std::ostream& o, const basic_json& j)
24136     {
24137         // read width member and use it as the indentation parameter if nonzero
24138         const bool pretty_print = o.width() > 0;
24139         const auto indentation = pretty_print ? o.width() : 0;
24140
24141         // reset width to 0 for subsequent calls to this stream
24142         o.width(0);
24143
24144         // do the actual serialization
24145         serializer s(detail::output_adapter<char>(o), o.fill());
24146         s.dump(j, pretty_print, false, static_cast<unsigned int>(indentation));
24147         return o;
24148     }
24149
24156     JSON_HEDLEY_DEPRECATED_FOR(3.0.0, operator<<(std::ostream&, const basic_json&))
24157     friend std::ostream& operator>>(const basic_json& j, std::ostream& o)
24158     {
24159         return o << j;
24160     }
24161 #endif  // JSON_NO_IO
24163
24165     // deserialization //
24167
24170
24173     template<typename InputType>
24174     JSON_HEDLEY_WARN_UNUSED_RESULT
24175     static basic_json parse(InputType&& i,
24176                             parser_callback_t cb = nullptr,
24177                             const bool allow_exceptions = true,
24178                             const bool ignore_comments = false,
24179                             const bool ignore_trailing_commas = false)
24180     {
24181         basic_json result;
24182         parser(detail::input_adapter(std::forward<InputType>(i)), std::move(cb), allow_exceptions,
      ignore_comments, ignore_trailing_commas).parse(true, result); //
      cppcheck-suppress[accessMoved,accessForwarded]
24183         return result;
24184     }
24185
24188     template<typename IteratorType>
24189     JSON_HEDLEY_WARN_UNUSED_RESULT
24190     static basic_json parse(IteratorType first,
24191                             IteratorType last,
24192                             parser_callback_t cb = nullptr,
24193                             const bool allow_exceptions = true,
24194                             const bool ignore_comments = false,
24195                             const bool ignore_trailing_commas = false)
24196     {
24197         basic_json result;
24198         parser(detail::input_adapter(std::move(first), std::move(last)), std::move(cb),
      allow_exceptions, ignore_comments, ignore_trailing_commas).parse(true, result); //
      cppcheck-suppress[accessMoved]
24199         return result;
24200     }
24201
24202     JSON_HEDLEY_WARN_UNUSED_RESULT
24203     JSON_HEDLEY_DEPRECATED_FOR(3.8.0, parse(ptr, ptr + len))
24204     static basic_json parse(detail::span_input_adapter&& i,
24205                             parser_callback_t cb = nullptr,
```

```
24206                            const bool allow_exceptions = true,
24207                            const bool ignore_comments = false,
24208                            const bool ignore_trailing_commas = false)
24209     {
24210         basic_json result;
24211         parser(i.get(), std::move(cb), allow_exceptions, ignore_comments,
      ignore_trailing_commas).parse(true, result); // cppcheck-suppress[accessMoved]
24212         return result;
24213     }
24214
24217     template<typename InputType>
24218     static bool accept(InputType&& i,
24219                        const bool ignore_comments = false,
24220                        const bool ignore_trailing_commas = false)
24221     {
24222         return parser(detail::input_adapter(std::forward<InputType>(i)), nullptr, false,
      ignore_comments, ignore_trailing_commas).accept(true);
24223     }
24224
24227     template<typename IteratorType>
24228     static bool accept(IteratorType first, IteratorType last,
24229                        const bool ignore_comments = false,
24230                        const bool ignore_trailing_commas = false)
24231     {
24232         return parser(detail::input_adapter(std::move(first), std::move(last)), nullptr, false,
      ignore_comments, ignore_trailing_commas).accept(true);
24233     }
24234
24235     JSON_HEDLEY_WARN_UNUSED_RESULT
24236     JSON_HEDLEY_DEPRECATED_FOR(3.8.0, accept(ptr, ptr + len))
24237     static bool accept(detail::span_input_adapter&& i,
24238                        const bool ignore_comments = false,
24239                        const bool ignore_trailing_commas = false)
24240     {
24241         return parser(i.get(), nullptr, false, ignore_comments, ignore_trailing_commas).accept(true);
24242     }
24243
24246     template <typename InputType, typename SAX>
24247     JSON_HEDLEY_NON_NULL(2)
24248     static bool sax_parse(InputType&& i, SAX* sax,
24249                           input_format_t format = input_format_t::json,
24250                           const bool strict = true,
24251                           const bool ignore_comments = false,
24252                           const bool ignore_trailing_commas = false)
24253     {
24254 #if defined(__clang__)
24255 #pragma clang diagnostic push
24256 #pragma clang diagnostic ignored "-Wtautological-pointer-compare"
24257 #elif defined(__GNUC__)
24258 #pragma GCC diagnostic push
24259 #pragma GCC diagnostic ignored "-Wnonnull-compare"
24260 #endif
24261         if (sax == nullptr)
24262         {
24263             JSON_THROW(other_error::create(502, "SAX handler must not be null", nullptr));
24264         }
24265 #if defined(__clang__)
24266 #pragma clang diagnostic pop
24267 #elif defined(__GNUC__)
24268 #pragma GCC diagnostic pop
24269 #endif
24270         auto ia = detail::input_adapter(std::forward<InputType>(i));
24271         return format == input_format_t::json
24272                ? parser(std::move(ia), nullptr, true, ignore_comments,
      ignore_trailing_commas).sax_parse(sax, strict)
24273                : detail::binary_reader<basic_json, decltype(ia), SAX>(std::move(ia),
      format).sax_parse(format, sax, strict);
24274     }
24275
24278     template<class IteratorType, class SAX>
24279     JSON_HEDLEY_NON_NULL(3)
24280     static bool sax_parse(IteratorType first, IteratorType last, SAX* sax,
24281                           input_format_t format = input_format_t::json,
24282                           const bool strict = true,
24283                           const bool ignore_comments = false,
24284                           const bool ignore_trailing_commas = false)
24285     {
24286 #if defined(__clang__)
24287 #pragma clang diagnostic push
24288 #pragma clang diagnostic ignored "-Wtautological-pointer-compare"
24289 #elif defined(__GNUC__)
24290 #pragma GCC diagnostic push
24291 #pragma GCC diagnostic ignored "-Wnonnull-compare"
24292 #endif
24293         if (sax == nullptr)
24294         {
24295             JSON_THROW(other_error::create(502, "SAX handler must not be null", nullptr));
```

```
24296         }
24297 #if defined(__clang__)
24298 #pragma clang diagnostic pop
24299 #elif defined(__GNUC__)
24300 #pragma GCC diagnostic pop
24301 #endif
24302         auto ia = detail::input_adapter(std::move(first), std::move(last));
24303         return format == input_format_t::json
24304                ? parser(std::move(ia), nullptr, true, ignore_comments,
      ignore_trailing_commas).sax_parse(sax, strict)
24305                : detail::binary_reader<basic_json, decltype(ia), SAX>(std::move(ia),
      format).sax_parse(format, sax, strict);
24306     }
24307
24313     template <typename SAX>
24314     JSON_HEDLEY_DEPRECATED_FOR(3.8.0, sax_parse(ptr, ptr + len, ...))
24315     JSON_HEDLEY_NON_NULL(2)
24316     static bool sax_parse(detail::span_input_adapter&& i, SAX* sax,
24317                           input_format_t format = input_format_t::json,
24318                           const bool strict = true,
24319                           const bool ignore_comments = false,
24320                           const bool ignore_trailing_commas = false)
24321     {
24322 #if defined(__clang__)
24323 #pragma clang diagnostic push
24324 #pragma clang diagnostic ignored "-Wtautological-pointer-compare"
24325 #elif defined(__GNUC__)
24326 #pragma GCC diagnostic push
24327 #pragma GCC diagnostic ignored "-Wnonnull-compare"
24328 #endif
24329         if (sax == nullptr)
24330         {
24331             JSON_THROW(other_error::create(502, "SAX handler must not be null", nullptr));
24332         }
24333 #if defined(__clang__)
24334 #pragma clang diagnostic pop
24335 #elif defined(__GNUC__)
24336 #pragma GCC diagnostic pop
24337 #endif
24338         auto ia = i.get();
24339         return format == input_format_t::json
24340                // NOLINTNEXTLINE(hicpp-move-const-arg,performance-move-const-arg)
24341                ? parser(std::move(ia), nullptr, true, ignore_comments,
      ignore_trailing_commas).sax_parse(sax, strict)
24342                // NOLINTNEXTLINE(hicpp-move-const-arg,performance-move-const-arg)
24343                : detail::binary_reader<basic_json, decltype(ia), SAX>(std::move(ia),
      format).sax_parse(format, sax, strict);
24344     }
24345 #ifndef JSON_NO_IO
24352     JSON_HEDLEY_DEPRECATED_FOR(3.0.0, operator»(std::istream&, basic_json&))
24353     friend std::istream& operator«(basic_json& j, std::istream& i)
24354     {
24355         return operator»(i, j);
24356     }
24357
24360     friend std::istream& operator»(std::istream& i, basic_json& j)
24361     {
24362         parser(detail::input_adapter(i)).parse(false, j);
24363         return i;
24364     }
24365 #endif  // JSON_NO_IO
24367
24369     // convenience functions //
24371
24374     JSON_HEDLEY_RETURNS_NON_NULL
24375     const char* type_name() const noexcept
24376     {
24377         switch (m_data.m_type)
24378         {
24379             case value_t::null:
24380                 return "null";
24381             case value_t::object:
24382                 return "object";
24383             case value_t::array:
24384                 return "array";
24385             case value_t::string:
24386                 return "string";
24387             case value_t::boolean:
24388                 return "boolean";
24389             case value_t::binary:
24390                 return "binary";
24391             case value_t::discarded:
24392                 return "discarded";
24393             case value_t::number_integer:
24394             case value_t::number_unsigned:
24395             case value_t::number_float:
24396                 return "number";
```

```
24397             default:
24398                 return "invalid";
24399         }
24400     }
24401
24402   JSON_PRIVATE_UNLESS_TESTED:
24403     // member variables //
24404
24405     struct data
24406     {
24407         value_t m_type = value_t::null;
24408
24409         json_value m_value = {};
24410
24411         data(const value_t v)
24412             : m_type(v), m_value(v)
24413         {
24414         }
24415
24416         data(size_type cnt, const basic_json& val)
24417             : m_type(value_t::array)
24418         {
24419             m_value.array = create<array_t>(cnt, val);
24420         }
24421
24422         data() noexcept = default;
24423         data(data&&) noexcept = default;
24424         data(const data&) noexcept = delete;
24425         data& operator=(data&&) noexcept = delete;
24426         data& operator=(const data&) noexcept = delete;
24427
24428         ~data() noexcept
24429         {
24430             m_value.destroy(m_type);
24431         }
24432     };
24433
24434     data m_data = {};
24435
24436 #if JSON_DIAGNOSTICS
24437     basic_json* m_parent = nullptr;
24438 #endif
24439
24440 #if JSON_DIAGNOSTIC_POSITIONS
24441     std::size_t start_position = std::string::npos;
24442     std::size_t end_position = std::string::npos;
24443   public:
24444     constexpr std::size_t start_pos() const noexcept
24445     {
24446         return start_position;
24447     }
24448
24449     constexpr std::size_t end_pos() const noexcept
24450     {
24451         return end_position;
24452     }
24453 #endif
24454
24455     // binary serialization/deserialization //
24456
24457
24458   public:
24459     static std::vector<std::uint8_t> to_cbor(const basic_json& j)
24460     {
24461         std::vector<std::uint8_t> result;
24462         to_cbor(j, result);
24463         return result;
24464     }
24465
24466     static void to_cbor(const basic_json& j, detail::output_adapter<std::uint8_t> o)
24467     {
24468         binary_writer<std::uint8_t>(o).write_cbor(j);
24469     }
24470
24471     static void to_cbor(const basic_json& j, detail::output_adapter<char> o)
24472     {
24473         binary_writer<char>(o).write_cbor(j);
24474     }
24475
24476     static std::vector<std::uint8_t> to_msgpack(const basic_json& j)
24477     {
24478         std::vector<std::uint8_t> result;
24479         to_msgpack(j, result);
24480         return result;
24481     }
24482
24483     static void to_msgpack(const basic_json& j, detail::output_adapter<std::uint8_t> o)
```

```
24505     {
24506         binary_writer<std::uint8_t>(o).write_msgpack(j);
24507     }
24508
24511     static void to_msgpack(const basic_json& j, detail::output_adapter<char> o)
24512     {
24513         binary_writer<char>(o).write_msgpack(j);
24514     }
24515
24518     static std::vector<std::uint8_t> to_ubjson(const basic_json& j,
24519             const bool use_size = false,
24520             const bool use_type = false)
24521     {
24522         std::vector<std::uint8_t> result;
24523         to_ubjson(j, result, use_size, use_type);
24524         return result;
24525     }
24526
24529     static void to_ubjson(const basic_json& j, detail::output_adapter<std::uint8_t> o,
24530                           const bool use_size = false, const bool use_type = false)
24531     {
24532         binary_writer<std::uint8_t>(o).write_ubjson(j, use_size, use_type);
24533     }
24534
24537     static void to_ubjson(const basic_json& j, detail::output_adapter<char> o,
24538                           const bool use_size = false, const bool use_type = false)
24539     {
24540         binary_writer<char>(o).write_ubjson(j, use_size, use_type);
24541     }
24542
24545     static std::vector<std::uint8_t> to_bjdata(const basic_json& j,
24546             const bool use_size = false,
24547             const bool use_type = false,
24548             const bjdata_version_t version = bjdata_version_t::draft2)
24549     {
24550         std::vector<std::uint8_t> result;
24551         to_bjdata(j, result, use_size, use_type, version);
24552         return result;
24553     }
24554
24557     static void to_bjdata(const basic_json& j, detail::output_adapter<std::uint8_t> o,
24558                           const bool use_size = false, const bool use_type = false,
24559                           const bjdata_version_t version = bjdata_version_t::draft2)
24560     {
24561         binary_writer<std::uint8_t>(o).write_ubjson(j, use_size, use_type, true, true, version);
24562     }
24563
24566     static void to_bjdata(const basic_json& j, detail::output_adapter<char> o,
24567                           const bool use_size = false, const bool use_type = false,
24568                           const bjdata_version_t version = bjdata_version_t::draft2)
24569     {
24570         binary_writer<char>(o).write_ubjson(j, use_size, use_type, true, true, version);
24571     }
24572
24575     static std::vector<std::uint8_t> to_bson(const basic_json& j)
24576     {
24577         std::vector<std::uint8_t> result;
24578         to_bson(j, result);
24579         return result;
24580     }
24581
24584     static void to_bson(const basic_json& j, detail::output_adapter<std::uint8_t> o)
24585     {
24586         binary_writer<std::uint8_t>(o).write_bson(j);
24587     }
24588
24591     static void to_bson(const basic_json& j, detail::output_adapter<char> o)
24592     {
24593         binary_writer<char>(o).write_bson(j);
24594     }
24595
24598     template<typename InputType>
24599     JSON_HEDLEY_WARN_UNUSED_RESULT
24600     static basic_json from_cbor(InputType&& i,
24601                                 const bool strict = true,
24602                                 const bool allow_exceptions = true,
24603                                 const cbor_tag_handler_t tag_handler = cbor_tag_handler_t::error)
24604     {
24605         basic_json result;
24606         auto ia = detail::input_adapter(std::forward<InputType>(i));
24607         detail::json_sax_dom_parser<basic_json, decltype(ia)> sdp(result, allow_exceptions);
24608         const bool res = binary_reader<decltype(ia)>(std::move(ia),
      input_format_t::cbor).sax_parse(input_format_t::cbor, &sdp, strict, tag_handler); //
      cppcheck-suppress[accessMoved]
24609         return res ? result : basic_json(value_t::discarded);
24610     }
24611
```

```
24614     template<typename IteratorType>
24615     JSON_HEDLEY_WARN_UNUSED_RESULT
24616     static basic_json from_cbor(IteratorType first, IteratorType last,
24617                                 const bool strict = true,
24618                                 const bool allow_exceptions = true,
24619                                 const cbor_tag_handler_t tag_handler = cbor_tag_handler_t::error)
24620     {
24621         basic_json result;
24622         auto ia = detail::input_adapter(std::move(first), std::move(last));
24623         detail::json_sax_dom_parser<basic_json, decltype(ia)> sdp(result, allow_exceptions);
24624         const bool res = binary_reader<decltype(ia)>(std::move(ia),
      input_format_t::cbor).sax_parse(input_format_t::cbor, &sdp, strict, tag_handler); //
      cppcheck-suppress[accessMoved]
24625         return res ? result : basic_json(value_t::discarded);
24626     }
24627
24628     template<typename T>
24629     JSON_HEDLEY_WARN_UNUSED_RESULT
24630     JSON_HEDLEY_DEPRECATED_FOR(3.8.0, from_cbor(ptr, ptr + len))
24631     static basic_json from_cbor(const T* ptr, std::size_t len,
24632                                 const bool strict = true,
24633                                 const bool allow_exceptions = true,
24634                                 const cbor_tag_handler_t tag_handler = cbor_tag_handler_t::error)
24635     {
24636         return from_cbor(ptr, ptr + len, strict, allow_exceptions, tag_handler);
24637     }
24638
24639     JSON_HEDLEY_WARN_UNUSED_RESULT
24640     JSON_HEDLEY_DEPRECATED_FOR(3.8.0, from_cbor(ptr, ptr + len))
24641     static basic_json from_cbor(detail::span_input_adapter&& i,
24642                                 const bool strict = true,
24643                                 const bool allow_exceptions = true,
24644                                 const cbor_tag_handler_t tag_handler = cbor_tag_handler_t::error)
24645     {
24646         basic_json result;
24647         auto ia = i.get();
24648         detail::json_sax_dom_parser<basic_json, decltype(ia)> sdp(result, allow_exceptions);
24649         // NOLINTNEXTLINE(hicpp-move-const-arg,performance-move-const-arg)
24650         const bool res = binary_reader<decltype(ia)>(std::move(ia),
      input_format_t::cbor).sax_parse(input_format_t::cbor, &sdp, strict, tag_handler); //
      cppcheck-suppress[accessMoved]
24651         return res ? result : basic_json(value_t::discarded);
24652     }
24653
24656     template<typename InputType>
24657     JSON_HEDLEY_WARN_UNUSED_RESULT
24658     static basic_json from_msgpack(InputType&& i,
24659                                    const bool strict = true,
24660                                    const bool allow_exceptions = true)
24661     {
24662         basic_json result;
24663         auto ia = detail::input_adapter(std::forward<InputType>(i));
24664         detail::json_sax_dom_parser<basic_json, decltype(ia)> sdp(result, allow_exceptions);
24665         const bool res = binary_reader<decltype(ia)>(std::move(ia),
      input_format_t::msgpack).sax_parse(input_format_t::msgpack, &sdp, strict); //
      cppcheck-suppress[accessMoved]
24666         return res ? result : basic_json(value_t::discarded);
24667     }
24668
24671     template<typename IteratorType>
24672     JSON_HEDLEY_WARN_UNUSED_RESULT
24673     static basic_json from_msgpack(IteratorType first, IteratorType last,
24674                                    const bool strict = true,
24675                                    const bool allow_exceptions = true)
24676     {
24677         basic_json result;
24678         auto ia = detail::input_adapter(std::move(first), std::move(last));
24679         detail::json_sax_dom_parser<basic_json, decltype(ia)> sdp(result, allow_exceptions);
24680         const bool res = binary_reader<decltype(ia)>(std::move(ia),
      input_format_t::msgpack).sax_parse(input_format_t::msgpack, &sdp, strict); //
      cppcheck-suppress[accessMoved]
24681         return res ? result : basic_json(value_t::discarded);
24682     }
24683
24684     template<typename T>
24685     JSON_HEDLEY_WARN_UNUSED_RESULT
24686     JSON_HEDLEY_DEPRECATED_FOR(3.8.0, from_msgpack(ptr, ptr + len))
24687     static basic_json from_msgpack(const T* ptr, std::size_t len,
24688                                    const bool strict = true,
24689                                    const bool allow_exceptions = true)
24690     {
24691         return from_msgpack(ptr, ptr + len, strict, allow_exceptions);
24692     }
24693
24694     JSON_HEDLEY_WARN_UNUSED_RESULT
24695     JSON_HEDLEY_DEPRECATED_FOR(3.8.0, from_msgpack(ptr, ptr + len))
24696     static basic_json from_msgpack(detail::span_input_adapter&& i,
```

```
24697                                    const bool strict = true,
24698                                    const bool allow_exceptions = true)
24699     {
24700         basic_json result;
24701         auto ia = i.get();
24702         detail::json_sax_dom_parser<basic_json, decltype(ia)> sdp(result, allow_exceptions);
24703         // NOLINTNEXTLINE(hicpp-move-const-arg,performance-move-const-arg)
24704         const bool res = binary_reader<decltype(ia)>(std::move(ia),
      input_format_t::msgpack).sax_parse(input_format_t::msgpack, &sdp, strict); //
      cppcheck-suppress[accessMoved]
24705         return res ? result : basic_json(value_t::discarded);
24706     }
24707
24710     template<typename InputType>
24711     JSON_HEDLEY_WARN_UNUSED_RESULT
24712     static basic_json from_ubjson(InputType&& i,
24713                                   const bool strict = true,
24714                                   const bool allow_exceptions = true)
24715     {
24716         basic_json result;
24717         auto ia = detail::input_adapter(std::forward<InputType>(i));
24718         detail::json_sax_dom_parser<basic_json, decltype(ia)> sdp(result, allow_exceptions);
24719         const bool res = binary_reader<decltype(ia)>(std::move(ia),
      input_format_t::ubjson).sax_parse(input_format_t::ubjson, &sdp, strict); //
      cppcheck-suppress[accessMoved]
24720         return res ? result : basic_json(value_t::discarded);
24721     }
24722
24725     template<typename IteratorType>
24726     JSON_HEDLEY_WARN_UNUSED_RESULT
24727     static basic_json from_ubjson(IteratorType first, IteratorType last,
24728                                   const bool strict = true,
24729                                   const bool allow_exceptions = true)
24730     {
24731         basic_json result;
24732         auto ia = detail::input_adapter(std::move(first), std::move(last));
24733         detail::json_sax_dom_parser<basic_json, decltype(ia)> sdp(result, allow_exceptions);
24734         const bool res = binary_reader<decltype(ia)>(std::move(ia),
      input_format_t::ubjson).sax_parse(input_format_t::ubjson, &sdp, strict); //
      cppcheck-suppress[accessMoved]
24735         return res ? result : basic_json(value_t::discarded);
24736     }
24737
24738     template<typename T>
24739     JSON_HEDLEY_WARN_UNUSED_RESULT
24740     JSON_HEDLEY_DEPRECATED_FOR(3.8.0, from_ubjson(ptr, ptr + len))
24741     static basic_json from_ubjson(const T* ptr, std::size_t len,
24742                                   const bool strict = true,
24743                                   const bool allow_exceptions = true)
24744     {
24745         return from_ubjson(ptr, ptr + len, strict, allow_exceptions);
24746     }
24747
24748     JSON_HEDLEY_WARN_UNUSED_RESULT
24749     JSON_HEDLEY_DEPRECATED_FOR(3.8.0, from_ubjson(ptr, ptr + len))
24750     static basic_json from_ubjson(detail::span_input_adapter&& i,
24751                                   const bool strict = true,
24752                                   const bool allow_exceptions = true)
24753     {
24754         basic_json result;
24755         auto ia = i.get();
24756         detail::json_sax_dom_parser<basic_json, decltype(ia)> sdp(result, allow_exceptions);
24757         // NOLINTNEXTLINE(hicpp-move-const-arg,performance-move-const-arg)
24758         const bool res = binary_reader<decltype(ia)>(std::move(ia),
      input_format_t::ubjson).sax_parse(input_format_t::ubjson, &sdp, strict); //
      cppcheck-suppress[accessMoved]
24759         return res ? result : basic_json(value_t::discarded);
24760     }
24761
24764     template<typename InputType>
24765     JSON_HEDLEY_WARN_UNUSED_RESULT
24766     static basic_json from_bjdata(InputType&& i,
24767                                   const bool strict = true,
24768                                   const bool allow_exceptions = true)
24769     {
24770         basic_json result;
24771         auto ia = detail::input_adapter(std::forward<InputType>(i));
24772         detail::json_sax_dom_parser<basic_json, decltype(ia)> sdp(result, allow_exceptions);
24773         const bool res = binary_reader<decltype(ia)>(std::move(ia),
      input_format_t::bjdata).sax_parse(input_format_t::bjdata, &sdp, strict); //
      cppcheck-suppress[accessMoved]
24774         return res ? result : basic_json(value_t::discarded);
24775     }
24776
24779     template<typename IteratorType>
24780     JSON_HEDLEY_WARN_UNUSED_RESULT
24781     static basic_json from_bjdata(IteratorType first, IteratorType last,
```

```
24782                                 const bool strict = true,
24783                                 const bool allow_exceptions = true)
24784     {
24785         basic_json result;
24786         auto ia = detail::input_adapter(std::move(first), std::move(last));
24787         detail::json_sax_dom_parser<basic_json, decltype(ia)> sdp(result, allow_exceptions);
24788         const bool res = binary_reader<decltype(ia)>(std::move(ia),
      input_format_t::bjdata).sax_parse(input_format_t::bjdata, &sdp, strict); //
      cppcheck-suppress[accessMoved]
24789         return res ? result : basic_json(value_t::discarded);
24790     }
24791
24794     template<typename InputType>
24795     JSON_HEDLEY_WARN_UNUSED_RESULT
24796     static basic_json from_bson(InputType&& i,
24797                                 const bool strict = true,
24798                                 const bool allow_exceptions = true)
24799     {
24800         basic_json result;
24801         auto ia = detail::input_adapter(std::forward<InputType>(i));
24802         detail::json_sax_dom_parser<basic_json, decltype(ia)> sdp(result, allow_exceptions);
24803         const bool res = binary_reader<decltype(ia)>(std::move(ia),
      input_format_t::bson).sax_parse(input_format_t::bson, &sdp, strict); // cppcheck-suppress[accessMoved]
24804         return res ? result : basic_json(value_t::discarded);
24805     }
24806
24809     template<typename IteratorType>
24810     JSON_HEDLEY_WARN_UNUSED_RESULT
24811     static basic_json from_bson(IteratorType first, IteratorType last,
24812                                 const bool strict = true,
24813                                 const bool allow_exceptions = true)
24814     {
24815         basic_json result;
24816         auto ia = detail::input_adapter(std::move(first), std::move(last));
24817         detail::json_sax_dom_parser<basic_json, decltype(ia)> sdp(result, allow_exceptions);
24818         const bool res = binary_reader<decltype(ia)>(std::move(ia),
      input_format_t::bson).sax_parse(input_format_t::bson, &sdp, strict); // cppcheck-suppress[accessMoved]
24819         return res ? result : basic_json(value_t::discarded);
24820     }
24821
24822     template<typename T>
24823     JSON_HEDLEY_WARN_UNUSED_RESULT
24824     JSON_HEDLEY_DEPRECATED_FOR(3.8.0, from_bson(ptr, ptr + len))
24825     static basic_json from_bson(const T* ptr, std::size_t len,
24826                                 const bool strict = true,
24827                                 const bool allow_exceptions = true)
24828     {
24829         return from_bson(ptr, ptr + len, strict, allow_exceptions);
24830     }
24831
24832     JSON_HEDLEY_WARN_UNUSED_RESULT
24833     JSON_HEDLEY_DEPRECATED_FOR(3.8.0, from_bson(ptr, ptr + len))
24834     static basic_json from_bson(detail::span_input_adapter&& i,
24835                                 const bool strict = true,
24836                                 const bool allow_exceptions = true)
24837     {
24838         basic_json result;
24839         auto ia = i.get();
24840         detail::json_sax_dom_parser<basic_json, decltype(ia)> sdp(result, allow_exceptions);
24841         // NOLINTNEXTLINE(hicpp-move-const-arg,performance-move-const-arg)
24842         const bool res = binary_reader<decltype(ia)>(std::move(ia),
      input_format_t::bson).sax_parse(input_format_t::bson, &sdp, strict); // cppcheck-suppress[accessMoved]
24843         return res ? result : basic_json(value_t::discarded);
24844     }
24846
24848     // JSON Pointer support //
24850
24853
24856     reference operator[](const json_pointer& ptr)
24857     {
24858         return ptr.get_unchecked(this);
24859     }
24860
24861     template<typename BasicJsonType, detail::enable_if_t<detail::is_basic_json<BasicJsonType>::value,
      int> = 0>
24862     JSON_HEDLEY_DEPRECATED_FOR(3.11.0, basic_json::json_pointer or
      nlohmann::json_pointer<basic_json::string_t>) // NOLINT(readability/alt_tokens)
24863     reference operator[](const ::nlohmann::json_pointer<BasicJsonType>& ptr)
24864     {
24865         return ptr.get_unchecked(this);
24866     }
24867
24870     const_reference operator[](const json_pointer& ptr) const
24871     {
24872         return ptr.get_unchecked(this);
24873     }
24874
```

```
24875     template<typename BasicJsonType, detail::enable_if_t<detail::is_basic_json<BasicJsonType>::value,
      int> = 0>
24876     JSON_HEDLEY_DEPRECATED_FOR(3.11.0, basic_json::json_pointer or
      nlohmann::json_pointer<basic_json::string_t>) // NOLINT(readability/alt_tokens)
24877     const_reference operator[](const ::nlohmann::json_pointer<BasicJsonType>& ptr) const
24878     {
24879         return ptr.get_unchecked(this);
24880     }
24881
24884     reference at(const json_pointer& ptr)
24885     {
24886         return ptr.get_checked(this);
24887     }
24888
24889     template<typename BasicJsonType, detail::enable_if_t<detail::is_basic_json<BasicJsonType>::value,
      int> = 0>
24890     JSON_HEDLEY_DEPRECATED_FOR(3.11.0, basic_json::json_pointer or
      nlohmann::json_pointer<basic_json::string_t>) // NOLINT(readability/alt_tokens)
24891     reference at(const ::nlohmann::json_pointer<BasicJsonType>& ptr)
24892     {
24893         return ptr.get_checked(this);
24894     }
24895
24898     const_reference at(const json_pointer& ptr) const
24899     {
24900         return ptr.get_checked(this);
24901     }
24902
24903     template<typename BasicJsonType, detail::enable_if_t<detail::is_basic_json<BasicJsonType>::value,
      int> = 0>
24904     JSON_HEDLEY_DEPRECATED_FOR(3.11.0, basic_json::json_pointer or
      nlohmann::json_pointer<basic_json::string_t>) // NOLINT(readability/alt_tokens)
24905     const_reference at(const ::nlohmann::json_pointer<BasicJsonType>& ptr) const
24906     {
24907         return ptr.get_checked(this);
24908     }
24909
24912     basic_json flatten() const
24913     {
24914         basic_json result(value_t::object);
24915         json_pointer::flatten("", *this, result);
24916         return result;
24917     }
24918
24921     basic_json unflatten() const
24922     {
24923         return json_pointer::unflatten(*this);
24924     }
24925
24927
24929     // JSON Patch functions //
24931
24934
24937     void patch_inplace(const basic_json& json_patch)
24938     {
24939         basic_json& result = *this;
24940         // the valid JSON Patch operations
24941         enum class patch_operations {add, remove, replace, move, copy, test, invalid};
24942
24943         const auto get_op = [](const string_t& op)
24944         {
24945             if (op == "add")
24946             {
24947                 return patch_operations::add;
24948             }
24949             if (op == "remove")
24950             {
24951                 return patch_operations::remove;
24952             }
24953             if (op == "replace")
24954             {
24955                 return patch_operations::replace;
24956             }
24957             if (op == "move")
24958             {
24959                 return patch_operations::move;
24960             }
24961             if (op == "copy")
24962             {
24963                 return patch_operations::copy;
24964             }
24965             if (op == "test")
24966             {
24967                 return patch_operations::test;
24968             }
24969
24970             return patch_operations::invalid;
```

```
24971         };
24972
24973         // wrapper for "add" operation; add value at ptr
24974         const auto operation_add = [&result](json_pointer & ptr, const basic_json & val)
24975         {
24976             // adding to the root of the target document means replacing it
24977             if (ptr.empty())
24978             {
24979                 result = val;
24980                 return;
24981             }
24982
24983             // make sure the top element of the pointer exists
24984             json_pointer const top_pointer = ptr.top();
24985             if (top_pointer != ptr)
24986             {
24987                 result.at(top_pointer);
24988             }
24989
24990             // get reference to the parent of the JSON pointer ptr
24991             const auto last_path = ptr.back();
24992             ptr.pop_back();
24993             // parent must exist when performing patch add per RFC6902 specs
24994             basic_json& parent = result.at(ptr);
24995
24996             switch (parent.m_data.m_type)
24997             {
24998                 case value_t::null:
24999                 case value_t::object:
25000                 {
25001                     // use operator[] to add value
25002                     parent[last_path] = val;
25003                     break;
25004                 }
25005
25006                 case value_t::array:
25007                 {
25008                     if (last_path == "-")
25009                     {
25010                         // special case: append to back
25011                         parent.push_back(val);
25012                     }
25013                     else
25014                     {
25015                         const auto idx = json_pointer::template array_index<basic_json_t>(last_path);
25016                         if (JSON_HEDLEY_UNLIKELY(idx > parent.size()))
25017                         {
25018                             // avoid undefined behavior
25019                             JSON_THROW(out_of_range::create(401, detail::concat("array index ",
      std::to_string(idx), " is out of range"), &parent));
25020                         }
25021
25022                         // default case: insert add offset
25023                         parent.insert(parent.begin() + static_cast<difference_type>(idx), val);
25024                     }
25025                     break;
25026                 }
25027
25028                 // if there exists a parent, it cannot be primitive
25029                 case value_t::string: // LCOV_EXCL_LINE
25030                 case value_t::boolean: // LCOV_EXCL_LINE
25031                 case value_t::number_integer: // LCOV_EXCL_LINE
25032                 case value_t::number_unsigned: // LCOV_EXCL_LINE
25033                 case value_t::number_float: // LCOV_EXCL_LINE
25034                 case value_t::binary: // LCOV_EXCL_LINE
25035                 case value_t::discarded: // LCOV_EXCL_LINE
25036                 default:            // LCOV_EXCL_LINE
25037                     JSON_ASSERT(false); // NOLINT(cert-dcl03-c,hicpp-static-assert,misc-static-assert)
      LCOV_EXCL_LINE
25038             }
25039         };
25040
25041         // wrapper for "remove" operation; remove value at ptr
25042         const auto operation_remove = [this, & result](json_pointer & ptr)
25043         {
25044             // get reference to the parent of the JSON pointer ptr
25045             const auto last_path = ptr.back();
25046             ptr.pop_back();
25047             basic_json& parent = result.at(ptr);
25048
25049             // remove child
25050             if (parent.is_object())
25051             {
25052                 // perform range check
25053                 auto it = parent.find(last_path);
25054                 if (JSON_HEDLEY_LIKELY(it != parent.end()))
25055                 {
```

```
25056                     parent.erase(it);
25057                 }
25058                 else
25059                 {
25060                     JSON_THROW(out_of_range::create(403, detail::concat("key '", last_path, "' not
      found"), this));
25061                 }
25062             }
25063             else if (parent.is_array())
25064             {
25065                 // note erase performs range check
25066                 parent.erase(json_pointer::template array_index<basic_json_t>(last_path));
25067             }
25068         };
25069
25070         // type check: top level value must be an array
25071         if (JSON_HEDLEY_UNLIKELY(!json_patch.is_array()))
25072         {
25073             JSON_THROW(parse_error::create(104, 0, "JSON patch must be an array of objects",
      &json_patch));
25074         }
25075
25076         // iterate and apply the operations
25077         for (const auto& val : json_patch)
25078         {
25079             // wrapper to get a value for an operation
25080             const auto get_value = [&val](const string_t& op,
25081                                           const string_t& member,
25082                                           bool string_type) -> basic_json &
25083             {
25084                 // find value
25085                 auto it = val.m_data.m_value.object->find(member);
25086
25087                 // context-sensitive error message
25088                 const auto error_msg = (op == "op") ? "operation" : detail::concat("operation '", op,
      '\''); // NOLINT(bugprone-unused-local-non-trivial-variable)
25089
25090                 // check if the desired value is present
25091                 if (JSON_HEDLEY_UNLIKELY(it == val.m_data.m_value.object->end()))
25092                 {
25093                     // NOLINTNEXTLINE(performance-inefficient-string-concatenation)
25094                     JSON_THROW(parse_error::create(105, 0, detail::concat(error_msg, " must have
      member '", member, "'"), &val));
25095                 }
25096
25097                 // check if the result is of type string
25098                 if (JSON_HEDLEY_UNLIKELY(string_type && !it->second.is_string()))
25099                 {
25100                     // NOLINTNEXTLINE(performance-inefficient-string-concatenation)
25101                     JSON_THROW(parse_error::create(105, 0, detail::concat(error_msg, " must have
      string member '", member, "'"), &val));
25102                 }
25103
25104                 // no error: return value
25105                 return it->second;
25106             };
25107
25108             // type check: every element of the array must be an object
25109             if (JSON_HEDLEY_UNLIKELY(!val.is_object()))
25110             {
25111                 JSON_THROW(parse_error::create(104, 0, "JSON patch must be an array of objects",
      &val));
25112             }
25113
25114             // collect mandatory members
25115             const auto op = get_value("op", "op", true).template get<string_t>();
25116             const auto path = get_value(op, "path", true).template get<string_t>();
25117             json_pointer ptr(path);
25118
25119             switch (get_op(op))
25120             {
25121                 case patch_operations::add:
25122                 {
25123                     operation_add(ptr, get_value("add", "value", false));
25124                     break;
25125                 }
25126
25127                 case patch_operations::remove:
25128                 {
25129                     operation_remove(ptr);
25130                     break;
25131                 }
25132
25133                 case patch_operations::replace:
25134                 {
25135                     // the "path" location must exist - use at()
25136                     result.at(ptr) = get_value("replace", "value", false);
```

```
25137                     break;
25138                 }
25139
25140                 case patch_operations::move:
25141                 {
25142                     const auto from_path = get_value("move", "from", true).template get<string_t>();
25143                     json_pointer from_ptr(from_path);
25144
25145                     // the "from" location must exist - use at()
25146                     basic_json const v = result.at(from_ptr);
25147
25148                     // The move operation is functionally identical to a
25149                     // "remove" operation on the "from" location, followed
25150                     // immediately by an "add" operation at the target
25151                     // location with the value that was just removed.
25152                     operation_remove(from_ptr);
25153                     operation_add(ptr, v);
25154                     break;
25155                 }
25156
25157                 case patch_operations::copy:
25158                 {
25159                     const auto from_path = get_value("copy", "from", true).template get<string_t>();
25160                     const json_pointer from_ptr(from_path);
25161
25162                     // the "from" location must exist - use at()
25163                     basic_json const v = result.at(from_ptr);
25164
25165                     // The copy is functionally identical to an "add"
25166                     // operation at the target location using the value
25167                     // specified in the "from" member.
25168                     operation_add(ptr, v);
25169                     break;
25170                 }
25171
25172                 case patch_operations::test:
25173                 {
25174                     bool success = false;
25175                     JSON_TRY
25176                     {
25177                         // check if "value" matches the one at "path"
25178                         // the "path" location must exist - use at()
25179                         success = (result.at(ptr) == get_value("test", "value", false));
25180                     }
25181                     JSON_INTERNAL_CATCH (out_of_range&)
25182                     {
25183                         // ignore out of range errors: success remains false
25184                     }
25185
25186                     // throw an exception if the test fails
25187                     if (JSON_HEDLEY_UNLIKELY(!success))
25188                     {
25189                         JSON_THROW(other_error::create(501, detail::concat("unsuccessful: ",
      val.dump()), &val));
25190                     }
25191
25192                     break;
25193                 }
25194
25195                 case patch_operations::invalid:
25196                 default:
25197                 {
25198                     // op must be "add", "remove", "replace", "move", "copy", or
25199                     // "test"
25200                     JSON_THROW(parse_error::create(105, 0, detail::concat("operation value '", op, "'
      is invalid"), &val));
25201                 }
25202             }
25203         }
25204     }
25205
25208     basic_json patch(const basic_json& json_patch) const
25209     {
25210         basic_json result = *this;
25211         result.patch_inplace(json_patch);
25212         return result;
25213     }
25214
25217     JSON_HEDLEY_WARN_UNUSED_RESULT
25218     static basic_json diff(const basic_json& source, const basic_json& target,
25219                            const string_t& path = "")
25220     {
25221         // the patch
25222         basic_json result(value_t::array);
25223
25224         // if the values are the same, return an empty patch
25225         if (source == target)
```

```
25226         {
25227             return result;
25228         }
25229 
25230         if (source.type() != target.type())
25231         {
25232             // different types: replace value
25233             result.push_back(
25234             {
25235                 {"op", "replace"}, {"path", path}, {"value", target}
25236             });
25237             return result;
25238         }
25239 
25240         switch (source.type())
25241         {
25242             case value_t::array:
25243             {
25244                 // first pass: traverse common elements
25245                 std::size_t i = 0;
25246                 while (i < source.size() && i < target.size())
25247                 {
25248                     // recursive call to compare array values at index i
25249                     auto temp_diff = diff(source[i], target[i], detail::concat<string_t>(path, '/',
      detail::to_string<string_t>(i)));
25250                     result.insert(result.end(), temp_diff.begin(), temp_diff.end());
25251                     ++i;
25252                 }
25253 
25254                 // We now reached the end of at least one array
25255                 // in a second pass, traverse the remaining elements
25256 
25257                 // remove my remaining elements
25258                 const auto end_index = static_cast<difference_type>(result.size());
25259                 while (i < source.size())
25260                 {
25261                     // add operations in reverse order to avoid invalid
25262                     // indices
25263                     result.insert(result.begin() + end_index, object(
25264                     {
25265                         {"op", "remove"},
25266                         {"path", detail::concat<string_t>(path, '/', detail::to_string<string_t>(i))}
25267                     }));
25268                     ++i;
25269                 }
25270 
25271                 // add other remaining elements
25272                 while (i < target.size())
25273                 {
25274                     result.push_back(
25275                     {
25276                         {"op", "add"},
25277                         {"path", detail::concat<string_t>(path, "/-")},
25278                         {"value", target[i]}
25279                     });
25280                     ++i;
25281                 }
25282 
25283                 break;
25284             }
25285 
25286             case value_t::object:
25287             {
25288                 // first pass: traverse this object's elements
25289                 for (auto it = source.cbegin(); it != source.cend(); ++it)
25290                 {
25291                     // escape the key name to be used in a JSON patch
25292                     const auto path_key = detail::concat<string_t>(path, '/',
      detail::escape(it.key()));
25293 
25294                     if (target.find(it.key()) != target.end())
25295                     {
25296                         // recursive call to compare object values at key it
25297                         auto temp_diff = diff(it.value(), target[it.key()], path_key);
25298                         result.insert(result.end(), temp_diff.begin(), temp_diff.end());
25299                     }
25300                     else
25301                     {
25302                         // found a key that is not in o -> remove it
25303                         result.push_back(object(
25304                         {
25305                             {"op", "remove"}, {"path", path_key}
25306                         }));
25307                     }
25308                 }
25309 
25310                 // second pass: traverse other object's elements
```

```
25311                 for (auto it = target.cbegin(); it != target.cend(); ++it)
25312                 {
25313                     if (source.find(it.key()) == source.end())
25314                     {
25315                         // found a key that is not in this -> add it
25316                         const auto path_key = detail::concat<string_t>(path, '/',
     detail::escape(it.key()));
25317                         result.push_back(
25318                         {
25319                             {"op", "add"}, {"path", path_key},
25320                             {"value", it.value()}
25321                         });
25322                     }
25323                 }
25324
25325                 break;
25326             }
25327
25328             case value_t::null:
25329             case value_t::string:
25330             case value_t::boolean:
25331             case value_t::number_integer:
25332             case value_t::number_unsigned:
25333             case value_t::number_float:
25334             case value_t::binary:
25335             case value_t::discarded:
25336             default:
25337             {
25338                 // both primitive types: replace value
25339                 result.push_back(
25340                 {
25341                     {"op", "replace"}, {"path", path}, {"value", target}
25342                 });
25343                 break;
25344             }
25345         }
25346
25347         return result;
25348     }
25350
25352     // JSON Merge Patch functions //
25354
25357
25360     void merge_patch(const basic_json& apply_patch)
25361     {
25362         if (apply_patch.is_object())
25363         {
25364             if (!is_object())
25365             {
25366                 *this = object();
25367             }
25368             for (auto it = apply_patch.begin(); it != apply_patch.end(); ++it)
25369             {
25370                 if (it.value().is_null())
25371                 {
25372                     erase(it.key());
25373                 }
25374                 else
25375                 {
25376                     operator[](it.key()).merge_patch(it.value());
25377                 }
25378             }
25379         }
25380         else
25381         {
25382             *this = apply_patch;
25383         }
25384     }
25385
25387 };
25388
25391 NLOHMANN_BASIC_JSON_TPL_DECLARATION
25392 std::string to_string(const NLOHMANN_BASIC_JSON_TPL& j)
25393 {
25394     return j.dump();
25395 }
25396
25397 inline namespace literals
25398 {
25399 inline namespace json_literals
25400 {
25401
25404 JSON_HEDLEY_NON_NULL(1)
25405 #if !defined(JSON_HEDLEY_GCC_VERSION) || JSON_HEDLEY_GCC_VERSION_CHECK(4,9,0)
25406     inline nlohmann::json operator""_json(const char* s, std::size_t n)
25407 #else
25408     // GCC 4.8 requires a space between "" and suffix
```

```
25409     inline nlohmann::json operator"" _json(const char* s, std::size_t n)
25410 #endif
25411 {
25412     return nlohmann::json::parse(s, s + n);
25413 }
25414
25415 #if defined(__cpp_char8_t)
25416 JSON_HEDLEY_NON_NULL(1)
25417 inline nlohmann::json operator""_json(const char8_t* s, std::size_t n)
25418 {
25419     return nlohmann::json::parse(reinterpret_cast<const char*>(s),
25420                                  reinterpret_cast<const char*>(s) + n);
25421 }
25422 #endif
25423
25426 JSON_HEDLEY_NON_NULL(1)
25427 #if !defined(JSON_HEDLEY_GCC_VERSION) || JSON_HEDLEY_GCC_VERSION_CHECK(4,9,0)
25428     inline nlohmann::json::json_pointer operator""_json_pointer(const char* s, std::size_t n)
25429 #else
25430     // GCC 4.8 requires a space between "" and suffix
25431     inline nlohmann::json::json_pointer operator"" _json_pointer(const char* s, std::size_t n)
25432 #endif
25433 {
25434     return nlohmann::json::json_pointer(std::string(s, n));
25435 }
25436
25437 #if defined(__cpp_char8_t)
25438 inline nlohmann::json::json_pointer operator""_json_pointer(const char8_t* s, std::size_t n)
25439 {
25440     return nlohmann::json::json_pointer(std::string(reinterpret_cast<const char*>(s), n));
25441 }
25442 #endif
25443
25444 }  // namespace json_literals
25445 }  // namespace literals
25446 NLOHMANN_JSON_NAMESPACE_END
25447
25449 // nonmember support //
25451
25452 namespace std // NOLINT(cert-dcl58-cpp)
25453 {
25454
25457 NLOHMANN_BASIC_JSON_TPL_DECLARATION
25458 struct hash<nlohmann::NLOHMANN_BASIC_JSON_TPL> // NOLINT(cert-dcl58-cpp)
25459 {
25460     std::size_t operator()(const nlohmann::NLOHMANN_BASIC_JSON_TPL& j) const
25461     {
25462         return nlohmann::detail::hash(j);
25463     }
25464 };
25465
25466 // specialization for std::less<value_t>
25467 template<>
25468 struct less< ::nlohmann::detail::value_t> // do not remove the space after '<', see
      https://github.com/nlohmann/json/pull/679
25469 {
25474     bool operator()(::nlohmann::detail::value_t lhs,
25475                     ::nlohmann::detail::value_t rhs) const noexcept
25476     {
25477 #if JSON_HAS_THREE_WAY_COMPARISON
25478         return std::is_lt(lhs <=> rhs); // *NOPAD*
25479 #else
25480         return ::nlohmann::detail::operator<(lhs, rhs);
25481 #endif
25482     }
25483 };
25484
25485 // C++20 prohibit function specialization in the std namespace.
25486 #ifndef JSON_HAS_CPP_20
25487
25490 NLOHMANN_BASIC_JSON_TPL_DECLARATION
25491 inline void swap(nlohmann::NLOHMANN_BASIC_JSON_TPL& j1, nlohmann::NLOHMANN_BASIC_JSON_TPL& j2)
      noexcept(  // NOLINT(readability-inconsistent-declaration-parameter-name, cert-dcl58-cpp)
25492     is_nothrow_move_constructible<nlohmann::NLOHMANN_BASIC_JSON_TPL>::value&&
      // NOLINT(misc-redundant-expression,cppcoreguidelines-noexcept-swap,performance-noexcept-swap)
25493     is_nothrow_move_assignable<nlohmann::NLOHMANN_BASIC_JSON_TPL>::value)
25494 {
25495     j1.swap(j2);
25496 }
25497
25498 #endif
25499
25500 }  // namespace std
25501
25502 #if JSON_USE_GLOBAL_UDLS
25503     #if !defined(JSON_HEDLEY_GCC_VERSION) || JSON_HEDLEY_GCC_VERSION_CHECK(4,9,0)
25504         using nlohmann::literals::json_literals::operator""_json; //
```

```
    NOLINT(misc-unused-using-decls,google-global-names-in-headers)
25505         using nlohmann::literals::json_literals::operator""_json_pointer;
    //NOLINT(misc-unused-using-decls,google-global-names-in-headers)
25506     #else
25507         // GCC 4.8 requires a space between "" and suffix
25508         using nlohmann::literals::json_literals::operator"" _json; //
    NOLINT(misc-unused-using-decls,google-global-names-in-headers)
25509         using nlohmann::literals::json_literals::operator"" _json_pointer;
    //NOLINT(misc-unused-using-decls,google-global-names-in-headers)
25510     #endif
25511 #endif
25512
25513 // #include <nlohmann/detail/macro_unscope.hpp>
25514 //      __ _____ _____ _____
25515 //   __|  |   __|     |   | |  JSON for Modern C++
25516 // |  |  |__   |  |  | | | |  version 3.12.0
25517 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
25518 //
25519 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
25520 // SPDX-License-Identifier: MIT
25521
25522
25523
25524 // restore clang diagnostic settings
25525 #if defined(__clang__)
25526     #pragma clang diagnostic pop
25527 #endif
25528
25529 // clean up
25530 #undef JSON_ASSERT
25531 #undef JSON_INTERNAL_CATCH
25532 #undef JSON_THROW
25533 #undef JSON_PRIVATE_UNLESS_TESTED
25534 #undef NLOHMANN_BASIC_JSON_TPL_DECLARATION
25535 #undef NLOHMANN_BASIC_JSON_TPL
25536 #undef JSON_EXPLICIT
25537 #undef NLOHMANN_CAN_CALL_STD_FUNC_IMPL
25538 #undef JSON_INLINE_VARIABLE
25539 #undef JSON_NO_UNIQUE_ADDRESS
25540 #undef JSON_DISABLE_ENUM_SERIALIZATION
25541 #undef JSON_USE_GLOBAL_UDLS
25542
25543 #ifndef JSON_TEST_KEEP_MACROS
25544     #undef JSON_CATCH
25545     #undef JSON_TRY
25546     #undef JSON_HAS_CPP_11
25547     #undef JSON_HAS_CPP_14
25548     #undef JSON_HAS_CPP_17
25549     #undef JSON_HAS_CPP_20
25550     #undef JSON_HAS_CPP_23
25551     #undef JSON_HAS_CPP_26
25552     #undef JSON_HAS_FILESYSTEM
25553     #undef JSON_HAS_EXPERIMENTAL_FILESYSTEM
25554     #undef JSON_HAS_THREE_WAY_COMPARISON
25555     #undef JSON_HAS_RANGES
25556     #undef JSON_HAS_STATIC_RTTI
25557     #undef JSON_USE_LEGACY_DISCARDED_VALUE_COMPARISON
25558 #endif
25559
25560 // #include <nlohmann/thirdparty/hedley/hedley_undef.hpp>
25561 //      __ _____ _____ _____
25562 //   __|  |   __|     |   | |  JSON for Modern C++
25563 // |  |  |__   |  |  | | | |  version 3.12.0
25564 // |_____|_____|_____|_|___|  https://github.com/nlohmann/json
25565 //
25566 // SPDX-FileCopyrightText: 2013-2026 Niels Lohmann <https://nlohmann.me>
25567 // SPDX-License-Identifier: MIT
25568
25569
25570
25571 #undef JSON_HEDLEY_ALWAYS_INLINE
25572 #undef JSON_HEDLEY_ARM_VERSION
25573 #undef JSON_HEDLEY_ARM_VERSION_CHECK
25574 #undef JSON_HEDLEY_ARRAY_PARAM
25575 #undef JSON_HEDLEY_ASSUME
25576 #undef JSON_HEDLEY_BEGIN_C_DECLS
25577 #undef JSON_HEDLEY_CLANG_HAS_ATTRIBUTE
25578 #undef JSON_HEDLEY_CLANG_HAS_BUILTIN
25579 #undef JSON_HEDLEY_CLANG_HAS_CPP_ATTRIBUTE
25580 #undef JSON_HEDLEY_CLANG_HAS_DECLSPEC_DECLSPEC_ATTRIBUTE
25581 #undef JSON_HEDLEY_CLANG_HAS_EXTENSION
25582 #undef JSON_HEDLEY_CLANG_HAS_FEATURE
25583 #undef JSON_HEDLEY_CLANG_HAS_WARNING
25584 #undef JSON_HEDLEY_COMPCERT_VERSION
25585 #undef JSON_HEDLEY_COMPCERT_VERSION_CHECK
25586 #undef JSON_HEDLEY_CONCAT
25587 #undef JSON_HEDLEY_CONCAT3
```

```
25588 #undef JSON_HEDLEY_CONCAT3_EX
25589 #undef JSON_HEDLEY_CONCAT_EX
25590 #undef JSON_HEDLEY_CONST
25591 #undef JSON_HEDLEY_CONSTEXPR
25592 #undef JSON_HEDLEY_CONST_CAST
25593 #undef JSON_HEDLEY_CPP_CAST
25594 #undef JSON_HEDLEY_CRAY_VERSION
25595 #undef JSON_HEDLEY_CRAY_VERSION_CHECK
25596 #undef JSON_HEDLEY_C_DECL
25597 #undef JSON_HEDLEY_DEPRECATED
25598 #undef JSON_HEDLEY_DEPRECATED_FOR
25599 #undef JSON_HEDLEY_DIAGNOSTIC_DISABLE_CAST_QUAL
25600 #undef JSON_HEDLEY_DIAGNOSTIC_DISABLE_CPP98_COMPAT_WRAP_
25601 #undef JSON_HEDLEY_DIAGNOSTIC_DISABLE_DEPRECATED
25602 #undef JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_CPP_ATTRIBUTES
25603 #undef JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNKNOWN_PRAGMAS
25604 #undef JSON_HEDLEY_DIAGNOSTIC_DISABLE_UNUSED_FUNCTION
25605 #undef JSON_HEDLEY_DIAGNOSTIC_POP
25606 #undef JSON_HEDLEY_DIAGNOSTIC_PUSH
25607 #undef JSON_HEDLEY_DMC_VERSION
25608 #undef JSON_HEDLEY_DMC_VERSION_CHECK
25609 #undef JSON_HEDLEY_EMPTY_BASES
25610 #undef JSON_HEDLEY_EMSCRIPTEN_VERSION
25611 #undef JSON_HEDLEY_EMSCRIPTEN_VERSION_CHECK
25612 #undef JSON_HEDLEY_END_C_DECLS
25613 #undef JSON_HEDLEY_FLAGS
25614 #undef JSON_HEDLEY_FLAGS_CAST
25615 #undef JSON_HEDLEY_GCC_HAS_ATTRIBUTE
25616 #undef JSON_HEDLEY_GCC_HAS_BUILTIN
25617 #undef JSON_HEDLEY_GCC_HAS_CPP_ATTRIBUTE
25618 #undef JSON_HEDLEY_GCC_HAS_DECLSPEC_ATTRIBUTE
25619 #undef JSON_HEDLEY_GCC_HAS_EXTENSION
25620 #undef JSON_HEDLEY_GCC_HAS_FEATURE
25621 #undef JSON_HEDLEY_GCC_HAS_WARNING
25622 #undef JSON_HEDLEY_GCC_NOT_CLANG_VERSION_CHECK
25623 #undef JSON_HEDLEY_GCC_VERSION
25624 #undef JSON_HEDLEY_GCC_VERSION_CHECK
25625 #undef JSON_HEDLEY_GNUC_HAS_ATTRIBUTE
25626 #undef JSON_HEDLEY_GNUC_HAS_BUILTIN
25627 #undef JSON_HEDLEY_GNUC_HAS_CPP_ATTRIBUTE
25628 #undef JSON_HEDLEY_GNUC_HAS_DECLSPEC_ATTRIBUTE
25629 #undef JSON_HEDLEY_GNUC_HAS_EXTENSION
25630 #undef JSON_HEDLEY_GNUC_HAS_FEATURE
25631 #undef JSON_HEDLEY_GNUC_HAS_WARNING
25632 #undef JSON_HEDLEY_GNUC_VERSION
25633 #undef JSON_HEDLEY_GNUC_VERSION_CHECK
25634 #undef JSON_HEDLEY_HAS_ATTRIBUTE
25635 #undef JSON_HEDLEY_HAS_BUILTIN
25636 #undef JSON_HEDLEY_HAS_CPP_ATTRIBUTE
25637 #undef JSON_HEDLEY_HAS_CPP_ATTRIBUTE_NS
25638 #undef JSON_HEDLEY_HAS_DECLSPEC_ATTRIBUTE
25639 #undef JSON_HEDLEY_HAS_EXTENSION
25640 #undef JSON_HEDLEY_HAS_FEATURE
25641 #undef JSON_HEDLEY_HAS_WARNING
25642 #undef JSON_HEDLEY_IAR_VERSION
25643 #undef JSON_HEDLEY_IAR_VERSION_CHECK
25644 #undef JSON_HEDLEY_IBM_VERSION
25645 #undef JSON_HEDLEY_IBM_VERSION_CHECK
25646 #undef JSON_HEDLEY_IMPORT
25647 #undef JSON_HEDLEY_INLINE
25648 #undef JSON_HEDLEY_INTEL_CL_VERSION
25649 #undef JSON_HEDLEY_INTEL_CL_VERSION_CHECK
25650 #undef JSON_HEDLEY_INTEL_VERSION
25651 #undef JSON_HEDLEY_INTEL_VERSION_CHECK
25652 #undef JSON_HEDLEY_IS_CONSTANT
25653 #undef JSON_HEDLEY_IS_CONSTEXPR_
25654 #undef JSON_HEDLEY_LIKELY
25655 #undef JSON_HEDLEY_MALLOC
25656 #undef JSON_HEDLEY_MCST_LCC_VERSION
25657 #undef JSON_HEDLEY_MCST_LCC_VERSION_CHECK
25658 #undef JSON_HEDLEY_MESSAGE
25659 #undef JSON_HEDLEY_MSVC_VERSION
25660 #undef JSON_HEDLEY_MSVC_VERSION_CHECK
25661 #undef JSON_HEDLEY_NEVER_INLINE
25662 #undef JSON_HEDLEY_NON_NULL
25663 #undef JSON_HEDLEY_NO_ESCAPE
25664 #undef JSON_HEDLEY_NO_RETURN
25665 #undef JSON_HEDLEY_NO_THROW
25666 #undef JSON_HEDLEY_NULL
25667 #undef JSON_HEDLEY_PELLES_VERSION
25668 #undef JSON_HEDLEY_PELLES_VERSION_CHECK
25669 #undef JSON_HEDLEY_PGI_VERSION
25670 #undef JSON_HEDLEY_PGI_VERSION_CHECK
25671 #undef JSON_HEDLEY_PREDICT
25672 #undef JSON_HEDLEY_PRINTF_FORMAT
25673 #undef JSON_HEDLEY_PRIVATE
25674 #undef JSON_HEDLEY_PUBLIC
```

```
25675 #undef JSON_HEDLEY_PURE
25676 #undef JSON_HEDLEY_REINTERPRET_CAST
25677 #undef JSON_HEDLEY_REQUIRE
25678 #undef JSON_HEDLEY_REQUIRE_CONSTEXPR
25679 #undef JSON_HEDLEY_REQUIRE_MSG
25680 #undef JSON_HEDLEY_RESTRICT
25681 #undef JSON_HEDLEY_RETURNS_NON_NULL
25682 #undef JSON_HEDLEY_SENTINEL
25683 #undef JSON_HEDLEY_STATIC_ASSERT
25684 #undef JSON_HEDLEY_STATIC_CAST
25685 #undef JSON_HEDLEY_STRINGIFY
25686 #undef JSON_HEDLEY_STRINGIFY_EX
25687 #undef JSON_HEDLEY_SUNPRO_VERSION
25688 #undef JSON_HEDLEY_SUNPRO_VERSION_CHECK
25689 #undef JSON_HEDLEY_TINYC_VERSION
25690 #undef JSON_HEDLEY_TINYC_VERSION_CHECK
25691 #undef JSON_HEDLEY_TI_ARMCL_VERSION
25692 #undef JSON_HEDLEY_TI_ARMCL_VERSION_CHECK
25693 #undef JSON_HEDLEY_TI_CL2000_VERSION
25694 #undef JSON_HEDLEY_TI_CL2000_VERSION_CHECK
25695 #undef JSON_HEDLEY_TI_CL430_VERSION
25696 #undef JSON_HEDLEY_TI_CL430_VERSION_CHECK
25697 #undef JSON_HEDLEY_TI_CL6X_VERSION
25698 #undef JSON_HEDLEY_TI_CL6X_VERSION_CHECK
25699 #undef JSON_HEDLEY_TI_CL7X_VERSION
25700 #undef JSON_HEDLEY_TI_CL7X_VERSION_CHECK
25701 #undef JSON_HEDLEY_TI_CLPRU_VERSION
25702 #undef JSON_HEDLEY_TI_CLPRU_VERSION_CHECK
25703 #undef JSON_HEDLEY_TI_VERSION
25704 #undef JSON_HEDLEY_TI_VERSION_CHECK
25705 #undef JSON_HEDLEY_UNAVAILABLE
25706 #undef JSON_HEDLEY_UNLIKELY
25707 #undef JSON_HEDLEY_UNPREDICTABLE
25708 #undef JSON_HEDLEY_UNREACHABLE
25709 #undef JSON_HEDLEY_UNREACHABLE_RETURN
25710 #undef JSON_HEDLEY_VERSION
25711 #undef JSON_HEDLEY_VERSION_DECODE_MAJOR
25712 #undef JSON_HEDLEY_VERSION_DECODE_MINOR
25713 #undef JSON_HEDLEY_VERSION_DECODE_REVISION
25714 #undef JSON_HEDLEY_VERSION_ENCODE
25715 #undef JSON_HEDLEY_WARNING
25716 #undef JSON_HEDLEY_WARN_UNUSED_RESULT
25717 #undef JSON_HEDLEY_WARN_UNUSED_RESULT_MSG
25718 #undef JSON_HEDLEY_FALL_THROUGH
25719
25720
25721
25722 #endif  // INCLUDE_NLOHMANN_JSON_HPP_
```

## 10.8 mt.h

```
00001
00007
00008 #ifndef METRICS_MT_H
00009 #define METRICS_MT_H
00010
00021 class MersenneTwister
00022 {
00023 public:
00024     MersenneTwister(void);
00025     ~MersenneTwister(void);
00026
00027     double random(void) { return genrand_real1(); }
00028     void print(void);
00029
00030     void init_genrand(unsigned long s);
00031     void init_by_array(unsigned long* init_key, int key_length);
00032
00033     unsigned long genrand_int32(void);
00034     long genrand_int31(void);
00035     double genrand_real1(void);
00036     double genrand_real2(void);
00037     double genrand_real3(void);
00038     double genrand_res53(void);
00039
00040 private:
00041     static const int N                   = 624;
00042     static const int M                   = 397;
00043     // constant vector a
00044     static const unsigned long MATRIX_A   = 0x9908b0dfUL;
00045     // most significant w-r bits
00046     static const unsigned long UPPER_MASK = 0x80000000UL;
00047     // least significant r bits
```

```
00048     static const unsigned long LOWER_MASK = 0x7fffffffUL;
00049
00050     unsigned long* mt_;                  // the state vector
00051     int mti_;                            // mti == N+1 means mt not initialized
00052
00053     unsigned long* init_key_;            // Storage for the seed vector
00054     int key_length_;                     // Seed vector length
00055     unsigned long s_;                    // Seed integer
00056     bool seeded_by_array_;               // Seeded by an array
00057     bool seeded_by_int_;                 // Seeded by an integer
00058 };
00059
00060 #endif // METRICS_MT_H
```

## 10.9 include/Optimizer/Blind.h File Reference

Header file for the Blind (Random Walk) optimization algorithm.

```
#include "Optimizer/Optimizer.h"
```

### Classes

- class Blind

  *Implements a blind (random walk) optimization algorithm.*

### 10.9.1 Detailed Description

Header file for the Blind (Random Walk) optimization algorithm.

**Author**

Alex Buckley

Definition in file Blind.h.

## 10.10 Blind.h

Go to the documentation of this file.

```
00001
00007
00008
00009 #ifndef BLIND_H
00010 #define BLIND_H
00011
00012 #include "Optimizer/Optimizer.h"
00013
00014
00027 class Blind : public Optimizer {
00028 public:
00036
00037     Blind(SolutionBuilder& solutionBuilder, Problem& problem, int maxIterations)
00038         : Optimizer(solutionBuilder, problem, maxIterations)
00039     {
00040         bestFitnesses.resize(maxIterations);
00041         bestSolution.resize(solutionBuilder.getDimensions());
00042         solutions.resize(maxIterations);
00043     }
00044
00050     double optimize() override;
00051 };
00052
00053 #endif
```

## 10.11 include/Optimizer/LocalSearch.h File Reference

Header file for the Local Search optimization algorithm.

```
#include "Optimizer/Optimizer.h"
```

### Classes

- class LocalSearch

  *Implements a local search optimization algorithm.*

### 10.11.1 Detailed Description

Header file for the Local Search optimization algorithm.

**Author**

Alex Buckley

Definition in file LocalSearch.h.

## 10.12 LocalSearch.h

Go to the documentation of this file.

```
00001
00007
00008
00009 #ifndef LOCAL_SEARCH_H
00010 #define LOCAL_SEARCH_H
00011
00012 #include "Optimizer/Optimizer.h"
00013
00026 class LocalSearch : public Optimizer {
00027 private:
00029     const double delta;
00030
00032     const int numNeighbors;
00033
00040     void localSearch();
00041
00042 public:
00052     LocalSearch(SolutionBuilder& solutionBuilder, Problem& problem, int maxIterations, double delta,
      int numNeighbors)
00053         : Optimizer(solutionBuilder, problem, maxIterations),
00054           delta(delta),
00055           numNeighbors(numNeighbors)
00056     {}
00057
00063     double optimize() override;
00064 };
00065
00066 #endif
00067
```

## 10.13 include/Optimizer/Optimizer.h File Reference

```
#include <vector>
#include "Problem/Problem.h"
#include "SolutionBuilder.h"
```

### Classes

- class Optimizer

  *Abstract base class for all optimization algorithms.*

### 10.13.1 Detailed Description

**Author**

Alex Buckley

Definition in file Optimizer.h.

## 10.14 Optimizer.h

Go to the documentation of this file.

```
00001
00008
00009
00010 #ifndef OPTIMIZER_H
00011 #define OPTIMIZER_H
00012
00013 #include <vector>
00014
00015 #include "Problem/Problem.h"
00016 #include "SolutionBuilder.h"
00017
00018
00028 class Optimizer {
00029 public:
00037     Optimizer(SolutionBuilder& solutionBuilder, Problem& problem, int maxIterations)
00038         : problem(problem),
00039           solutionBuilder(solutionBuilder),
00040           maxIterations(maxIterations)
00041     {}
00042
00044     virtual ~Optimizer() = default;
00045
00051     virtual double optimize() = 0;
00052
00053     // Getters
00056
00058     double getBestFitness() { return bestFitnesses.back(); }
00059
00061     std::vector<double>& getBestSolution() { return bestSolution; }
00062
00064     std::vector<double>& getBestFitnesses() { return bestFitnesses; }
00065
00067     std::vector<std::vector<double>>& getSolutions() { return solutions; }
00068
00070     int getMaxIterations() { return maxIterations; }
00071
00073     Problem& getProblem() { return problem; }
00074
00076     SolutionBuilder& getSolutionBuilder() { return solutionBuilder; }
00077
00079
00080
00081 protected:
00083     Problem& problem;
00084
00086     SolutionBuilder& solutionBuilder;
00087
00089     int maxIterations;
00090
00092     std::vector<double> bestSolution;
00093
00095     std::vector<double> bestFitnesses;
00096
00098     std::vector<std::vector<double>> solutions;
00099 };
00100
00101 #endif
```

## 10.15 include/Optimizer/OptimizerFactory.h File Reference

Factory utility for instantiating different optimizer types.

```
#include "Optimizer/Optimizer.h"
#include "Optimizer/Blind.h"
#include "Optimizer/LocalSearch.h"
#include <memory>
#include <string>
```

### Classes

- class OptimizerFactory

  *Factory class for creating optimizer instances.*

### 10.15.1 Detailed Description

Factory utility for instantiating different optimizer types.

**Author**

Alex Buckley

Definition in file OptimizerFactory.h.

## 10.16 OptimizerFactory.h

Go to the documentation of this file.

```
00001
00007
00008
00009 #ifndef OPTIMIZER_FACTORY_H
00010 #define OPTIMIZER_FACTORY_H
00011
00012 #include "Optimizer/Optimizer.h"
00013 #include "Optimizer/Blind.h"
00014 #include "Optimizer/LocalSearch.h"
00015
00016 #include <memory>
00017 #include <string>
00018
00019
00028 class OptimizerFactory {
00029 public:
00040     static std::unique_ptr<Optimizer> initOptimizer(Problem& problem, ExperimentConfig& config,
      SolutionBuilder& builder) {
00041         if(config.optimizer == "blind")
00042             return std::make_unique<Blind>(builder, problem, config.maxIterations);
00043         else if(config.optimizer == "local")
00044             return std::make_unique<LocalSearch>(builder, problem, 1, config.neighborDelta,
      config.numNeighbors);
00045         else if(config.optimizer == "repeated local")
00046             return std::make_unique<LocalSearch>(builder, problem, config.maxIterations,
      config.neighborDelta, config.numNeighbors);
00047
00048         return nullptr;
00049     }
00050
00051 };
00052
00053 #endif
```

## 10.17 Population.h

```
00001 #ifndef POPULATION_H
00002 #define POPULATION_H
00003
00004 #include "Problem/Problem.h"
00005 #include <vector>
00006
00007 class Population {
00008 private:
00009     const int n; // Population size (number of solutions)
00010     const int m; // Dimension of each solution vector
00011     std::vector<std::vector<double» solutions; // Size n x m matrix of solution vectors
00012
00013 public:
00014     // Constructors
00015     Population(int populationSize, int dimension);
00016
00017     // Methods
00018     int initialize(double lower, double upper, int seed);
00019     std::vector<double> evaluate(const Problem& problem);
00020     int generateNeighbors(
00021         const std::vector<double>& center,
00022         double delta,
00023         double lower,
00024         double upper,
00025         int seed
00026     );
00027
00028     // Accessors
00029     const std::vector<std::vector<double»& getSolutions() const;
00030 };
00031
00032 #endif
```

## 10.18 include/Problem/AckleyOne.h File Reference

Implementation of the Ackley 1 function.

```
#include "Problem/Problem.h"
#include <vector>
#include <cmath>
```

### Classes

- class AckleyOne

  *Implements the Ackley 1 benchmark function.*

### 10.18.1 Detailed Description

Implementation of the Ackley 1 function.

**Author**

Alex Buckley

Definition in file AckleyOne.h.

## 10.19 AckleyOne.h

Go to the documentation of this file.

```
00001
00007
00008
00009 #ifndef ACKLEY_ONE_H
00010 #define ACKLEY_ONE_H
00011
00012 #include "Problem/Problem.h"
00013 #include <vector>
00014 #include <cmath>
00015
00016
00021 class AckleyOne : public Problem {
00022 private:
00023     static constexpr double LOWER = -32.0;
00024     static constexpr double UPPER = 32.0;
00025     static constexpr std::string_view NAME = "AckleyOne";
00026
00027 public:
00028     AckleyOne() : Problem(LOWER, UPPER, NAME) {}
00029
00030     double evaluate(const std::vector<double>& x) const override {
00031         double sum = 0.0;
00032
00033         for(size_t i = 0; i < x.size() - 1; i++) {
00034             double term1 = std::exp(-0.2) * std::sqrt(x[i] * x[i] + x[i+1] * x[i+1]);
00035             double term2 = 3 * (std::cos(2 * x[i]) + std::sin(2 * x[i+1]));
00036             sum += term1 + term2;
00037         }
00038
00039         return sum;
00040     }
00041 };
00042
00043 #endif
```

## 10.20 include/Problem/AckleyTwo.h File Reference

Implementation of the Ackley 2 function.

```
#include "Problem/Problem.h"
#include <vector>
#include <cmath>
#include <numbers>
```

### Classes

- class AckleyTwo

  *Implements the Ackley 2 benchmark function.*

### 10.20.1 Detailed Description

Implementation of the Ackley 2 function.

**Author**

Alex Buckley

Definition in file AckleyTwo.h.

## 10.21 AckleyTwo.h

Go to the documentation of this file.

```
00001
00007
00008
00009 #ifndef ACKLEY_TWO_H
00010 #define ACKLEY_TWO_H
00011
00012 #include "Problem/Problem.h"
00013 #include <vector>
00014 #include <cmath>
00015 #include <numbers>
00016
00017
00022 class AckleyTwo : public Problem {
00023 private:
00024     static constexpr double LOWER = -32.0;
00025     static constexpr double UPPER = 32.0;
00026     static constexpr std::string_view NAME = "AckleyTwo";
00027
00028 public:
00029     AckleyTwo() : Problem(LOWER, UPPER, NAME) {}
00030
00031     double evaluate(const std::vector<double>& x) const override {
00032         double sum = 0.0;
00033
00034         for(size_t i = 0; i < x.size() - 1; i++) {
00035             double sqrtTerm = std::sqrt((x[i] * x[i] + x[i+1] * x[i+1]) / 2);
00036             double term3 = -20 / (std::exp(0.2 * sqrtTerm));
00037             double term4 = -1 * std::exp(0.5 * std::cos(2 * std::numbers::pi * x[i]) + std::cos(2 *
      std::numbers::pi * x[i+1]));
00038
00039             sum += 20 + std::exp(1.0) + term3 + term4;
00040         }
00041
00042         return sum;
00043     }
00044 };
00045
00046 #endif
```

## 10.22 DeJongOne.h

```
00001
00007
00008 #ifndef DEJONG_H
00009 #define DEJONG_H
00010
00011 #include "Problem/Problem.h"
00012 #include <vector>
00013
00014
00019 class DeJongOne : public Problem {
00020 private:
00021     static constexpr double LOWER = -100.0;
00022     static constexpr double UPPER = 100.0;
00023     static constexpr std::string_view NAME = "DeJong_1";
00024
00025 public:
00026     // Constructor method
00027     DeJongOne() : Problem(LOWER, UPPER, NAME) {}
00028
00029     // Evaluate function
00030     double evaluate(const std::vector<double>& x) const override {
00031         double sum = 0.0;
00032
00033         for(double xi : x)
00034             sum += xi * xi;
00035
00036         return sum;
00037     }
00038 };
00039
00040 #endif
```

## 10.23 include/Problem/EggHolder.h File Reference

Implementation of the Egg Holder function.

```
#include "Problem/Problem.h"
#include <vector>
#include <cmath>
```

### Classes

- class EggHolder

  *Implements the Egg Holder benchmark function.*

### 10.23.1 Detailed Description

Implementation of the Egg Holder function.

**Author**

Alex Buckley

Definition in file EggHolder.h.

## 10.24 EggHolder.h

Go to the documentation of this file.

```
00001
00007
00008
00009 #ifndef EGG_HOLDER_H
00010 #define EGG_HOLDER_H
00011
00012 #include "Problem/Problem.h"
00013 #include <vector>
00014 #include <cmath>
00015
00016
00021 class EggHolder : public Problem {
00022 private:
00023     static constexpr double LOWER = -500.0;
00024     static constexpr double UPPER = 500.0;
00025     static constexpr std::string_view NAME = "EggHolder";
00026
00027     inline double sinSqrtAbs(double n) const {
00028         return std::sin(std::sqrt(std::abs(n)));
00029     }
00030
00031 public:
00032     EggHolder() : Problem(LOWER, UPPER, NAME) {}
00033
00034     double evaluate(const std::vector<double>& x) const override {
00035         double sum = 0.0;
00036
00037         for(size_t i = 0; i < x.size() - 1; i++) {
00038             double term1 = -x[i] * sinSqrtAbs(x[i] - x[i+1] - 47);
00039             double term2 = -(x[i+1] + 47) * sinSqrtAbs(x[i+1] + 47 + x[i] / 2);
00040
00041             sum += term1 + term2;
00042         }
00043
00044         return sum;
00045     }
00046 };
00047
00048 #endif
```

## 10.25 include/Problem/Griewangk.h File Reference

Implementation of the Griewangk function.

```
#include "Problem/Problem.h"
#include <vector>
#include <cmath>
```

### Classes

- class Griewangk

  *Implements the Griewangk benchmark function.*

### 10.25.1 Detailed Description

Implementation of the Griewangk function.

**Author**

Alex Buckley

Definition in file Griewangk.h.

## 10.26 Griewangk.h

Go to the documentation of this file.

```
00001
00007
00008 #ifndef GRIEWANGK_H
00009 #define GRIEWANGK_H
00010
00011 #include "Problem/Problem.h"
00012 #include <vector>
00013 #include <cmath>
00014
00015
00020 class Griewangk : public Problem {
00021 private:
00022     static constexpr double LOWER = -500.0;
00023     static constexpr double UPPER = 500.0;
00024     static constexpr std::string_view NAME = "Griewangk";
00025
00026 public:
00027     Griewangk() : Problem(LOWER, UPPER, NAME) {}
00028
00029     double evaluate(const std::vector<double>& x) const override {
00030         double sum = 0.0;
00031         double prod = 1.0;
00032
00033         for(size_t i = 0; i < x.size(); i++) {
00034             sum += x[i] * x[i];
00035             prod *= std::cos(x[i] / std::sqrt(i + 1));
00036         }
00037
00038         return 1.0 + sum / 4000.0 - prod;
00039     }
00040 };
00041
00042 #endif
```

# 10.27 include/Problem/Problem.h File Reference

```
#include <vector>
```

### Classes

- class Problem

  *Abstract base class for all optimization benchmark problems.*

## 10.27.1 Detailed Description

**Author**

Alex Buckley

Definition in file Problem.h.

# 10.28 Problem.h

Go to the documentation of this file.

```
00001
00008
00009 #ifndef PROBLEM_H
00010 #define PROBLEM_H
00011
00012 #include <vector>
00013
00021 class Problem {
00022 protected:
00023     const double lowerBound;
00024     const double upperBound;
00025     const std::string name;
00026
00027 public:
00034     Problem(double lb, double ub, const std::string_view n)
00035         : lowerBound(lb), upperBound(ub), name(n) {}
00036
00040     virtual ~Problem() = default;
00041
00049     virtual double evaluate(const std::vector<double>& x) const = 0;
00050
00053
00055     double getLowerBound() const { return lowerBound; }
00056
00058     double getUpperBound() const { return upperBound; }
00059
00061     const std::string getName() const { return name; }
00062
00064 };
00065
00066 #endif // PROBLEM_H
```

# 10.29 include/Problem/Rastrigin.h File Reference

Implementation of the Ratrigin function.

```
#include "Problem/Problem.h"
#include <vector>
#include <cmath>
#include <numbers>
```

### Classes

- class Rastrigin

  *Implements the Rastrigin benchmark function.*

### 10.29.1 Detailed Description

Implementation of the Ratrigin function.

**Author**

Alex Buckley

Definition in file Rastrigin.h.

## 10.30 Rastrigin.h

Go to the documentation of this file.

```
00001
00007
00008
00009 #ifndef RASTRIGIN_H
00010 #define RASTRIGIN_H
00011
00012 #include "Problem/Problem.h"
00013 #include <vector>
00014 #include <cmath>
00015 #include <numbers>
00016
00017
00022 class Rastrigin : public Problem {
00023 private:
00024     static constexpr double LOWER = -30.0;
00025     static constexpr double UPPER = 30.0;
00026     static constexpr std::string_view NAME = "Rastrigin";
00027
00028 public:
00029     Rastrigin() : Problem(LOWER, UPPER, NAME) {}
00030
00031     double evaluate(const std::vector<double>& x) const override {
00032         double sum = 0.0;
00033
00034         for(double xi : x) // Calculate summation
00035             sum += xi * xi - 10 * std::cos(2 * std::numbers::pi * xi);
00036
00037         return 10 * x.size() + sum;
00038     }
00039 };
00040
00041 #endif
```

## 10.31 include/Problem/Rosenbrock.h File Reference

Implementation of the Rosenbrock function.

```
#include "Problem/Problem.h"
#include <vector>
```

### Classes

- class Rosenbrock

  *Implements the Rosenbrock benchmark function.*

### 10.31.1 Detailed Description

Implementation of the Rosenbrock function.

**Author**

Alex Buckley

Definition in file Rosenbrock.h.

## 10.32 Rosenbrock.h

Go to the documentation of this file.

```
00001
00007
00008
00009 #ifndef ROSENBROCK_H
00010 #define ROSENBROCK_H
00011
00012 #include "Problem/Problem.h"
00013 #include <vector>
00014
00015
00020 class Rosenbrock : public Problem {
00021 private:
00022     static constexpr double LOWER = -100.0;
00023     static constexpr double UPPER = 100.0;
00024     static constexpr std::string_view NAME = "Rosenbrock";
00025
00026 public:
00027     Rosenbrock() : Problem(LOWER, UPPER, NAME) {}
00028
00029     double evaluate(const std::vector<double>& x) const override {
00030         double sum = 0.0;
00031
00032         for(size_t i = 0; i < x.size() - 1; i++) {
00033             // Calculate inner terms
00034             const double term1 = x[i] * x[i] - x[i+1];
00035             const double term2 = 1 - x[i];
00036
00037             // Calculate value for summation
00038             sum += 100 * term1 * term1 + term2 * term2;
00039         }
00040
00041         return sum;
00042     }
00043 };
00044
00045 #endif
```

## 10.33 include/Problem/Schwefel.h File Reference

Implementation of the Schwefel function.

```
#include "Problem/Problem.h"
#include <vector>
```

### Classes

- class Schwefel

  *Implements the Schwefel benchmark function.*

### 10.33.1 Detailed Description

Implementation of the Schwefel function.

**Author**

Alex Buckley

Definition in file Schwefel.h.

## 10.34 Schwefel.h

Go to the documentation of this file.

```
00001
00007
00008
00009 #ifndef SCHWEFEL_H
00010 #define SCHWEFEL_H
00011
00012 #include "Problem/Problem.h"
00013 #include <vector>
00014
00015
00020 class Schwefel : public Problem {
00021 private:
00022     static constexpr double LOWER = -512.0;
00023     static constexpr double UPPER = 512.0;
00024     static constexpr std::string_view NAME = "Schwefel";
00025
00026 public:
00027     Schwefel() : Problem(LOWER, UPPER, NAME) {}
00028
00029     double evaluate(const std::vector<double>& x) const override {
00030         double sum = 0.0;
00031
00032         for(double xi : x)
00033             sum += xi * std::sin(std::sqrt(std::abs(xi)));
00034
00035         return 418.9829 * x.size() - sum;
00036     }
00037 };
00038
00039 #endif
```

## 10.35 include/Problem/SineEnvelope.h File Reference

Implementation of the Sine Envelope function.

```
#include "Problem/Problem.h"
#include <vector>
#include <cmath>
```

### Classes

- class SineEnvelope

  *Implements the Sine Envelope benchmark function.*

### 10.35.1 Detailed Description

Implementation of the Sine Envelope function.

**Author**

Alex Buckley

Definition in file SineEnvelope.h.

## 10.36 SineEnvelope.h

Go to the documentation of this file.

```
00001
00007
00008
00009 #ifndef SINE_ENVELOPE_H
00010 #define SINE_ENVELOPE_H
00011
00012 #include "Problem/Problem.h"
00013 #include <vector>
00014 #include <cmath>
00015
00016
00021 class SineEnvelope : public Problem {
00022 private:
00023     static constexpr double LOWER = -30.0;
00024     static constexpr double UPPER = 30.0;
00025     static constexpr std::string_view NAME = "SineEnvelope";
00026
00027 public:
00028     SineEnvelope() : Problem(LOWER, UPPER, NAME) {}
00029
00030     double evaluate(const std::vector<double>& x) const override {
00031         double sum = 0.0;
00032
00033         for(size_t i = 0; i < x.size() - 1; i++) {
00034             double sqrSum = x[i] * x[i] + x[i+1] * x[i+1];
00035             double numerator = std::sin(sqrSum - 0.5) * std::sin(sqrSum - 0.5);
00036             double denom = (1 + 0.001 * sqrSum) * (1 + 0.001 * sqrSum);
00037             sum += numerator / denom + 0.5;
00038         }
00039
00040         return sum * -1;
00041     }
00042 };
00043
00044 #endif
```

## 10.37 include/Problem/StretchedV.h File Reference

Implementation of the Stretched V function.

```
#include "Problem/Problem.h"
#include <vector>
#include <cmath>
```

### Classes

- class StretchedV

  *Implements the StretchedV benchmark function.*

### 10.37.1 Detailed Description

Implementation of the Stretched V function.

**Author**

Alex Buckley

Definition in file StretchedV.h.

## 10.38 StretchedV.h

Go to the documentation of this file.

```
00001
00007
00008
00009 #ifndef STRETCHED_V_H
00010 #define STRETCHED_V_H
00011
00012 #include "Problem/Problem.h"
00013 #include <vector>
00014 #include <cmath>
00015
00016
00021 class StretchedV : public Problem {
00022 private:
00023     static constexpr double LOWER = -30.0;
00024     static constexpr double UPPER = 30.0;
00025     static constexpr std::string_view NAME = "StretchedV";
00026
00027 public:
00028     StretchedV() : Problem(LOWER, UPPER, NAME) {}
00029
00030     double evaluate(const std::vector<double>& x) const override {
00031         double sum = 0.0;
00032
00033         for(size_t i = 0; i < x.size() - 1; i++) {
00034             double sqrSum = x[i] * x[i] + x[i+1] * x[i+1];
00035             double factor1 = std::sqrt(std::sqrt(sqrSum));
00036             double factor2 = std::sin(50 * std::pow(sqrSum, 0.1));
00037
00038             sum += factor1 * factor2 * factor2 + 1;
00039         }
00040
00041         return sum;
00042     }
00043 };
00044
00045 #endif
```

## 10.39 include/ProblemFactory.h File Reference

Factory for instantiating benchmark problems by ID.

```
#include "Problem/Problem.h"
#include <array>
#include <memory>
#include <stdexcept>
```

### Classes

- class ProblemFactory

  *Utility to create problem instances dynamically.*

### 10.39.1 Detailed Description

Factory for instantiating benchmark problems by ID.

**Author**

Alex Buckley

Definition in file ProblemFactory.h.

## 10.40 ProblemFactory.h

Go to the documentation of this file.

```
00001
00007
00008 #ifndef PROBLEM_FACTORY_H
00009 #define PROBLEM_FACTORY_H
00010
00011 #include "Problem/Problem.h"
00012 #include <array>
00013 #include <memory>
00014 #include <stdexcept>
00015
00020 class ProblemFactory {
00021 public:
00028     static std::unique_ptr<Problem> create(int id);
00029 };
00030
00031
00032 #endif
```

## 10.41 include/RunExperiments.h File Reference

```
#include <vector>
#include <string>
#include "Config.h"
#include "Optimizer/Optimizer.h"
```

### Classes

- class RunExperiments

  *High-level controller that orchestrates the benchmarking process.*

## 10.42 RunExperiments.h

Go to the documentation of this file.

```
00001
00008
00009
00010 #ifndef RUN_EXPERIMENTS_H
00011 #define RUN_EXPERIMENTS_H
00012
00013 #include <vector>
00014 #include <string>
00015
00016 #include "Config.h"
00017 #include "Optimizer/Optimizer.h"
00018
00019
00029 class RunExperiments {
00030 private:
00031 std::vector<ExperimentConfig> configs;
00032     std::string outputFile;
00033
00039     bool loadConfig(const std::string& inputFile);
00040
00046     std::vector<std::string> getNames(std::vector<ExperimentConfig> configs);
00047
00055     bool writeResults(
00056         std::vector<double> bestSolution,
00057         std::vector<double> bestFitnesses,
00058         std::vector<std::vector<double» solutions
00059     );
00060
00063     static constexpr std::string_view bestFitnessesFile = "best_fitnesses.csv";
00064     static constexpr std::string_view solutionsFile = "solutions.csv";
00065     static constexpr std::string_view timesFile = "times.csv";
00067
00068 public:
00074     RunExperiments(const std::string inputFile, const std::string& outputFile)
00075         : outputFile(outputFile)
00076     {
00077         loadConfig(inputFile);
00078     }
00079
00086     int runExperiments();
00087
00088 };
00089
00090
00091
00092 #endif
```

## 10.43 include/SolutionBuilder.h File Reference

Utility class for generating and manipulating candidate solutions.

```
#include <vector>
#include "Problem/Problem.h"
#include "External/mt.h"
```

### Classes

- class SolutionBuilder

  *Responsible for creating random solutions and neighborhood samples.*

### 10.43.1 Detailed Description

Utility class for generating and manipulating candidate solutions.

**Author**

Alex Buckley

Definition in file SolutionBuilder.h.

## 10.44 SolutionBuilder.h

Go to the documentation of this file.

```
00001
00007
00008
00009 #ifndef SOLUTION_BUILDER_H
00010 #define SOLUTION_BUILDER_H
00011
00012 #include <vector>
00013
00014 #include "Problem/Problem.h"
00015 #include "External/mt.h"
00016
00017
00026 class SolutionBuilder {
00027 private:
00028     const int dimensions;
00029     const int lower;
00030     const int upper;
00031     MersenneTwister mt;
00032
00038     double checkBounds(double value);
00039 public:
00047     SolutionBuilder(int dimensions, int lower, int upper, int seed)
00048         : dimensions(dimensions),
00049           lower(lower),
00050           upper(upper)
00051     {
00052         mt.init_genrand(seed);
00053     }
00054
00059     std::vector<double> getRand();
00060
00072     std::vector<std::vector<double» getNeighbors(
00073         const std::vector<double>& center,
00074         int numNeighbors,
00075         double maxDelta
00076     );
00077
00079     double getDimensions() { return dimensions; }
00080 };
00081
00082 #endif
```

## 10.45 BenchmarkRunner.cpp

```
00001 #include "BenchmarkRunner.h"
00002
00003 #include <filesystem>
00004 #include <fstream>
00005 #include <iostream>
00006 #include <string>
00007 #include <stdexcept>
00008
00009
00010 using json = nlohmann::json;
00011
00012
00013 json BenchmarkRunner::loadJSON(const std::string& path) {
```

```
00014     std::ifstream f(path);
00015
00016     if(!f)
00017         throw std::runtime_error("Failed to open config file: " + path);
00018     else
00019         std::cout « "Opened file: " « path « "\n";
00020
00021     return json::parse(f);
00022 }
00023
00024
00025 Experiment BenchmarkRunner::parseExperiment(const json& j) {
00026     // Parse data from JSON
00027     std::string name = j.at("experiment_name").get<std::string>();
00028     int problemType = j.at("problem_type").get<int>();
00029     int popSize = j.at("population_size").get<int>();
00030     int dims = j.at("dimensions").get<int>();
00031     int upper = j.at("upper_bound").get<int>();
00032     int lower = j.at("lower_bound").get<int>();
00033     unsigned int seed = j.at("seed").get<unsigned int>();
00034
00035     // Return Experiment object
00036     return Experiment(
00037         name,
00038         problemType,
00039         popSize,
00040         dims,
00041         seed,
00042         lower,
00043         upper
00044     );
00045 }
00046
00047
00048 std::vector<Experiment> BenchmarkRunner::parseExperiments(const json& cfg) {
00049     std::vector<Experiment> experiments;
00050
00051     const auto& arr = cfg.at("experiments");
00052
00053     if(!arr.is_array())
00054         throw std::runtime_error("'experiments' must be an array");
00055
00056     for(const auto& jexp : arr)
00057         experiments.push_back(parseExperiment(jexp));
00058
00059     return experiments;
00060 }
00061
00062
00063 void BenchmarkRunner::writeFitnessCSV(const std::vector<Experiment>& experiments, const std::string&
      filename) {
00064     if (experiments.empty()) {
00065         throw std::runtime_error("No experiments to write");
00066     }
00067
00068     std::ofstream file(filename);
00069     if (!file.is_open()) {
00070         throw std::runtime_error("Cannot open file");
00071     }
00072
00073     // Assume all experiments have same fitness vector size
00074     const size_t fitnessCount = experiments.front().getFitness().size();
00075
00076     // Write experiment names as column headers
00077     for (size_t j = 0; j < experiments.size(); ++j) {
00078         file « experiments[j].getName(); // Write name
00079         if(j + 1 < experiments.size())
00080             file « ","; // Write delimiter
00081     }
00082
00083     file « "\n"; // Next line
00084
00085     // Write rows (fitness values)
00086     for (size_t i = 0; i < fitnessCount; ++i) { // Iterate through fitness value
00087         // Write fitness__i for each experiment
00088         for (size_t j = 0; j < experiments.size(); ++j) {
00089             const auto& fitness = experiments[j].getFitness(); // Get fitness values
00090
00091             if(fitness.size() != fitnessCount) // Size error
00092                 throw std::runtime_error("Fitness size mismatch between experiments");
00093
00094             file « fitness[i]; // Write fitness__i for experiment__j
00095
00096             if(j + 1 < experiments.size())
00097                 file « ","; // Write delimiter
00098         }
00099
```

```
00100         file « "\n"; // Write newline
00101     }
00102
00103     file.close();
00104 }
00105
00106
00107 void BenchmarkRunner::writeTimeCSV(const std::vector<Experiment>& experiments, const std::string&
      filename) {
00108     if(experiments.empty()) // Missing experiments
00109         throw std::runtime_error("No experiments to write");
00110
00111     std::ofstream file(filename);
00112     if (!file.is_open())
00113         throw std::runtime_error("Cannot open file");
00114
00115     // Write header to CSV
00116     file « "experiment_name,wall_time\n";
00117
00118     // Write experiment name and wall time to CSV
00119     for(size_t i = 0; i < experiments.size(); i++)
00120         file « experiments[i].getName() « ", " « experiments[i].getWallTime() « "\n";
00121
00122 }
00123
00124
00125 void BenchmarkRunner::runBenchmarks(const std::string& inputFile, const std::string& benchmarkName) {
00126
00127
00128     const std::string fitnessName = "fitness.csv";
00129     const std::string timeName = "time.csv";
00130
00131     json j = loadJSON(inputFile);
00132     std::vector<Experiment> experiments = parseExperiments(j);
00133
00134     // Run each experiment
00135     for(size_t i = 0; i < experiments.size(); i++)
00136         experiments[i].runExperiment();
00137
00138     namespace fs = std::filesystem; // Alias filesystem namespace
00139     fs::path resultDir = fs::current_path() / "results" / benchmarkName; // Get path to results
      directory
00140
00141     writeFitnessCSV(experiments, resultDir / fitnessCSV);
00142     writeTimeCSV(experiments, resultDir / timeCSV);
00143
00144
00145 }
```

## 10.46 Experiment.cpp

```
00001 #include "Experiment.h"
00002
00003 #include <iostream>
00004 #include <fstream>
00005 #include <vector>
00006 #include <string>
00007
00008
00009 #include <chrono>
00010 #include <iomanip>
00011 #include <sstream>
00012
00013
00014 #include "ProblemFactory.h"
00015
00016
00017
00018
00019 void Experiment::runExperiment() {
00020     // Initialize clock
00021     using clock = std::chrono::steady_clock;
00022
00023     // Time evaluation
00024     const auto start = clock::now();
00025     fitness = population.evaluate(*problem);
00026     const auto end = clock::now();
00027
00028     // Set wall time field
00029     wallTime = std::chrono::duration<double>(end - start).count();
00030 }
00031
00032
00033
```

# 10.47 mt.cpp

```
00001
00007
00008 /*
00009    A C-program for MT19937, with initialization improved 2002/1/26.
00010    Coded by Takuji Nishimura and Makoto Matsumoto.
00011
00012    Before using, initialize the state by using init_genrand(seed)
00013    or init_by_array(init_key, key_length).
00014
00015    Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
00016    All rights reserved.
00017
00018    Redistribution and use in source and binary forms, with or without
00019    modification, are permitted provided that the following conditions
00020    are met:
00021
00022      1. Redistributions of source code must retain the above copyright
00023         notice, this list of conditions and the following disclaimer.
00024
00025      2. Redistributions in binary form must reproduce the above copyright
00026         notice, this list of conditions and the following disclaimer in the
00027         documentation and/or other materials provided with the distribution.
00028
00029      3. The names of its contributors may not be used to endorse or promote
00030         products derived from this software without specific prior written
00031         permission.
00032
00033    THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
00034    "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
00035    LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
00036    A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
00037    CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
00038    EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
00039    PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
00040    PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
00041    LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
00042    NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
00043    SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
00044
00045    Any feedback is very welcome.
00046    http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
00047    email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
00048 */
00049
00050 #include <iostream>
00051 #include <cassert>
00052
00053 #include "External/mt.h"
00054
00058 MersenneTwister::MersenneTwister(void):
00059     mt_(new unsigned long[N]), mti_(N+1),
00060     init_key_(NULL), key_length_(0), s_(0),
00061     seeded_by_array_(false), seeded_by_int_(false)
00062 {
00063     unsigned long init[4] = { 0x123, 0x234, 0x345, 0x456 };
00064     unsigned long length = 4;
00065     init_by_array(init, length);
00066 }
00067
00071 MersenneTwister::~MersenneTwister(void)
00072 {
00073     assert(mt_ != NULL);
00074     delete[] mt_;
00075     mt_ = NULL;
00076
00077     assert(init_key_ != NULL);
00078     delete[] init_key_;
00079     init_key_ = NULL;
00080 }
00081
00087 void MersenneTwister::init_genrand(unsigned long s)
00088 {
00089     mt_[0]= s & 0xffffffffUL;
00090     for (mti_=1; mti_<N; mti_++) {
00091         mt_[mti_] =
00092         (1812433253UL * (mt_[mti_-1] ^ (mt_[mti_-1] >> 30)) + mti_);
00093         /* See Knuth TAOCP Vol2. 3rd Ed. P.106 for multiplier. */
00094         /* In the previous versions, MSBs of the seed affect   */
00095         /* only MSBs of the array mt_[].                        */
00096         /* 2002/01/09 modified by Makoto Matsumoto             */
00097         mt_[mti_] &= 0xffffffffUL;
00098         /* for >32 bit machines */
00099     }
00100     // Store the seed
```

```
00101     s_ = s;
00102     seeded_by_array_ = false;
00103     seeded_by_int_ = true;
00104 }
00105
00112 void MersenneTwister::init_by_array(unsigned long* init_key, int key_length)
00113 {
00114     // Store the key array
00115     int i, j, k;
00116     init_genrand(19650218UL);
00117     i=1; j=0;
00118     k = (N>key_length ? N : key_length);
00119     for (; k; k--) {
00120         mt_[i] = (mt_[i] ^ ((mt_[i-1] ^ (mt_[i-1] » 30)) * 1664525UL))
00121           + init_key[j] + j; /* non linear */
00122         mt_[i] &= 0xffffffffUL; /* for WORDSIZE > 32 machines */
00123         i++; j++;
00124         if (i>=N) { mt_[0] = mt_[N-1]; i=1; }
00125         if (j>=key_length) j=0;
00126     }
00127     for (k=N-1; k; k--) {
00128         mt_[i] = (mt_[i] ^ ((mt_[i-1] ^ (mt_[i-1] » 30)) * 1566083941UL))
00129           - i; /* non linear */
00130         mt_[i] &= 0xffffffffUL; /* for WORDSIZE > 32 machines */
00131         i++;
00132         if (i>=N) { mt_[0] = mt_[N-1]; i=1; }
00133     }
00134
00135     mt_[0] = 0x80000000UL; /* MSB is 1; assuring non-zero initial array */
00136
00137     // Store the seed
00138     if (init_key_ != NULL) {
00139         delete[] init_key_;
00140     }
00141     init_key_ = new unsigned long[key_length];
00142     for (int k = 0; k < key_length; k++) {
00143         init_key_[k] = init_key[k];
00144     }
00145     key_length_ = key_length;
00146     seeded_by_int_ = false;
00147     seeded_by_array_ = true;
00148 }
00149
00155 unsigned long MersenneTwister::genrand_int32(void)
00156 {
00157     unsigned long y;
00158     static unsigned long mag01[2]={0x0UL, MATRIX_A};
00159     /* mag01[x] = x * MATRIX_A  for x=0,1 */
00160
00161     if (mti_ >= N) { /* generate N words at one time */
00162         int kk;
00163
00164         if (mti_ == N+1)   /* if init_genrand() has not been called, */
00165             init_genrand(5489UL); /* a default initial seed is used */
00166
00167         for (kk=0;kk<N-M;kk++) {
00168             y = (mt_[kk]&UPPER_MASK)|(mt_[kk+1]&LOWER_MASK);
00169             mt_[kk] = mt_[kk+M] ^ (y » 1) ^ mag01[y & 0x1UL];
00170         }
00171         for (;kk<N-1;kk++) {
00172             y = (mt_[kk]&UPPER_MASK)|(mt_[kk+1]&LOWER_MASK);
00173             mt_[kk] = mt_[kk+(M-N)] ^ (y » 1) ^ mag01[y & 0x1UL];
00174         }
00175         y = (mt_[N-1]&UPPER_MASK)|(mt_[0]&LOWER_MASK);
00176         mt_[N-1] = mt_[M-1] ^ (y » 1) ^ mag01[y & 0x1UL];
00177
00178         mti_ = 0;
00179     }
00180
00181     y = mt_[mti_++];
00182
00183     /* Tempering */
00184     y ^= (y » 11);
00185     y ^= (y « 7) & 0x9d2c5680UL;
00186     y ^= (y « 15) & 0xefc60000UL;
00187     y ^= (y » 18);
00188
00189     return y;
00190 }
00191
00197 long MersenneTwister::genrand_int31(void)
00198 {
00199     return (long)(genrand_int32()»1);
00200 }
00201
00207 double MersenneTwister::genrand_real1(void)
00208 {
```

```
00209     return genrand_int32()*(1.0/4294967295.0);
00210     /* divided by 2^32-1 */
00211 }
00212
00218 double MersenneTwister::genrand_real2(void)
00219 {
00220     return genrand_int32()*(1.0/4294967296.0);
00221     /* divided by 2^32 */
00222 }
00223
00229 double MersenneTwister::genrand_real3(void)
00230 {
00231     return (((double)genrand_int32()) + 0.5)*(1.0/4294967296.0);
00232     /* divided by 2^32 */
00233 }
00234
00240 double MersenneTwister::genrand_res53(void)
00241 {
00242     unsigned long a=genrand_int32()»5, b=genrand_int32()»6;
00243     return(a*67108864.0+b)*(1.0/9007199254740992.0);
00244 }
00245 /* These real versions are due to Isaku Wada, 2002/01/09 added */
00246
00251 void MersenneTwister::print(void)
00252 {
00253     std::cout « "MersenneTwister (M. Matsumoto and T. Nishimura), seed = ";
00254     if (seeded_by_int_) {
00255         std::cout « s_ « std::endl;
00256     } else {
00257         std::cout « "[ ";
00258         for (int k = 0; k < key_length_; k++) {
00259             std::cout « init_key_[k] « " ";
00260         }
00261         std::cout « "]" « std::endl;
00262     }
00263 }
```

## 10.48 src/main.cpp File Reference

Entry point for the Numerical Optimization Benchmarks CLI.

```
#include <iostream>
#include <string.h>
#include "RunExperiments.h"
#include "debug.h"
```

### Functions

- int main (int argc, char ∗argv[ ])

### 10.48.1 Detailed Description

Entry point for the Numerical Optimization Benchmarks CLI.

**Author**

Alex Buckley

- This file handles the command-line interface, initializes the debugging environment, and hands off execution to the RunExperiments controller.
-

### 10.48.2 Usage

```
./optimization_benchmarks <config_path> <output_path>
```

- **Parameters**

| | |
|---|---|
| *argc* | Argument count. |

| | |
|---|---|
| *argv* | Argument vector. Expects [1] config path and [2] output path. |

**Returns**

int Status code (0 for success, 1 for error).

Definition in file main.cpp.

### 10.48.3 Function Documentation

#### 10.48.3.1 main()

```
int main (
            int argc,
            char * argv[])
```

Definition at line 24 of file main.cpp.

## 10.49 main.cpp

Go to the documentation of this file.

```
00001
00016
00017
00018 #include <iostream>
00019 #include <string.h>
00020
00021 #include "RunExperiments.h"
00022 #include "debug.h"
00023
00024 int main(int argc, char* argv[]) {
00025     // Debug print
00026     debug::log("\nDebug Mode Enabled For:\t", argv[0]);
00027
00028     if(argc < 3) { // Read program arguments
00029         std::cerr «"Error, not enough arguments provided\n";
00030         return 1;
00031     }
00032
00033     RunExperiments runner(argv[1], argv[2]);
00034     runner.runExperiments();
00035
00036     return 0;
00037 }
```

## 10.50 Blind.cpp

```
00001 #include "Optimizer/Blind.h"
00002
00003 #include <limits>
00004
00005
00006 double Blind::optimize() {
00007     // Start timing
00008     using clock = std::chrono::high_resolution_clock;
00009     auto start = clock::now();
00010
00011     // Iterate population
00012     for(int i = 0; i < maxIterations; i++) {
00013         // Get neighbors
00014         solutions[i] = solutionBuilder.getRand();
00015
```

```
00016         // Set next fitness
00017         bestFitnesses[i] = problem.evaluate(solutions[i]);
00018
00019         // Update best fitness
00020         if(i > 0 && bestFitnesses[i-1] < bestFitnesses[i]) {
00021             bestFitnesses[i] = bestFitnesses[i - 1];
00022             solutions[i] = solutions[i - 1];
00023         }
00024     }
00025
00026     // Return execution time
00027     return std::chrono::duration<double>(clock::now() - start).count();
00028 }
```

## 10.51 LocalSearch.cpp

```
00001 #include "Optimizer/LocalSearch.h"
00002
00003 #include <limits>
00004
00005 void LocalSearch::localSearch() {
00006     // Get initial population pseudo-randomly
00007     std::vector<double> curSolution = solutionBuilder.getRand();
00008     double curFitness = problem.evaluate(curSolution);
00009     bool minimaFound = false;
00010
00011     // Loop until local minima found
00012     while(!minimaFound) {
00013         minimaFound = true;
00014
00015         // Get set of neighbors
00016         std::vector<std::vector<double» neighbors = solutionBuilder.getNeighbors(
00017             curSolution,
00018             numNeighbors,
00019             delta
00020         );
00021
00022         // Track local minima
00023         int bestNeighborIdx = -1;
00024         double bestNeighborFitness = std::numeric_limits<double>::max();
00025
00026         // Check all neighbors
00027         for(size_t i = 0; i < neighbors.size(); i++) {
00028             // Evaluate neighbor's fitness
00029             double neighborFitness = problem.evaluate(neighbors[i]);
00030
00031             // Better neighbor found, update stats
00032             if(neighborFitness < bestNeighborFitness) {
00033                 bestNeighborIdx = i;
00034                 bestNeighborFitness = neighborFitness;
00035             }
00036         }
00037
00038         // Compare best neighbor to center fitness
00039         if(bestNeighborFitness < curFitness) {
00040             minimaFound = false;
00041             curSolution = neighbors[bestNeighborIdx];
00042             curFitness = bestNeighborFitness;
00043         }
00044
00045         // Append best found fitness to results
00046         if(!bestFitnesses.size() || bestFitnesses.back() > curFitness)
00047             bestFitnesses.push_back(curFitness); // New best found
00048         else // No new best found
00049             bestFitnesses.push_back(bestFitnesses.back());
00050     }
00051 }
00052
00053
00054
00055 double LocalSearch::optimize() {
00056     // Start timing
00057     using clock = std::chrono::high_resolution_clock;
00058     auto start = clock::now();
00059
00060     for(int i = 0; i < maxIterations; i++)
00061         localSearch();
00062
00063     return std::chrono::duration<double>(clock::now() - start).count();
00064 }
```

## 10.52 Population.cpp

```
00001 #include "Population.h"
00002 #include <iostream>
00003 #include "External/mt.h"
00004
00005 #define ERROR 0
00006 #define SUCCESS 1
00007
00008 Population::Population(int populationSize, int dimension)
00009     : n(populationSize), m(dimension),
00010       solutions(n, std::vector<double>(m, 0.0))
00011 {}
00012
00013 int Population::initialize(double lower, double upper, int seed) {
00014     if(n < 1 || m < 1) // Validate population size
00015         return ERROR;
00016
00017     // Initialize seeded pseudo-random generator
00018     MersenneTwister mt;
00019     mt.init_genrand(seed);
00020
00021     // Generate random solutions
00022     for(int i = 0; i < n; i++) {
00023         for(int j = 0; j < m; j++) {
00024             solutions[i][j] = lower + (upper - lower) * mt.genrand_real1();
00025         }
00026     }
00027
00028     return SUCCESS;
00029 }
00030
00031 int Population::generateNeighbors(
00032     const std::vector<double>& center,
00033     double delta,
00034     double lower,
00035     double upper,
00036     int seed
00037 ) {
00038     // Clear and resize vector to store neighbors as population
00039     solutions.clear();
00040     solutions.resize(2 * m, std::vector<double>(m, 0.0));
00041
00042     int idx = 0;
00043
00044     for(int j = 0; j < m; j++) {
00045         // plus delta
00046         auto plus = center;
00047         plus[j] = std::min(upper, plus[j] + delta);
00048         solutions[idx++] = plus;
00049
00050         // minus delta
00051         auto minus = center;
00052         minus[j] = std::max(lower, minus[j] - delta);
00053         solutions[idx++] = minus;
00054     }
00055 }
00056
00057
00058
00059 std::vector<double> Population::evaluate(const Problem& problem) {
00060     std::vector<double> fitness(n);
00061
00062     for(int i = 0; i < n; i++) {
00063         fitness[i] = problem.evaluate(solutions[i]);
00064     }
00065
00066     return fitness;
00067 }
00068
00069 const std::vector<std::vector<double»& Population::getSolutions() const {
00070     return solutions;
00071 }
```

## 10.53 ProblemFactory.cpp

```
00001 #include "ProblemFactory.h"
00002 #include "Problem/Schwefel.h"
00003 #include "Problem/DeJongOne.h"
00004 #include "Problem/Rosenbrock.h"
00005 #include "Problem/Rastrigin.h"
00006 #include "Problem/Griewangk.h"
```

```
00007 #include "Problem/SineEnvelope.h"
00008 #include "Problem/StretchedV.h"
00009 #include "Problem/AckleyOne.h"
00010 #include "Problem/AckleyTwo.h"
00011 #include "Problem/EggHolder.h"
00012
00013 namespace {
00014
00015 // Type alias for a factory function returning unique_ptr<Problem>
00016 using Creator = std::unique_ptr<Problem>(*)();
00017
00018 // Factory functions for each problem
00019 std::unique_ptr<Problem> createSchwefel() {
00020     return std::make_unique<Schwefel>();
00021 }
00022
00023 std::unique_ptr<Problem> createDeJongOne() {
00024     return std::make_unique<DeJongOne>();
00025 }
00026
00027 std::unique_ptr<Problem> createRosenbrock() {
00028     return std::make_unique<Rosenbrock>();
00029 }
00030
00031 std::unique_ptr<Problem> createRastrigin() {
00032     return std::make_unique<Rastrigin>();
00033 }
00034
00035 std::unique_ptr<Problem> createGriewangk() {
00036     return std::make_unique<Griewangk>();
00037 }
00038
00039 std::unique_ptr<Problem> createSineEnvelope() {
00040     return std::make_unique<SineEnvelope>();
00041 }
00042
00043 std::unique_ptr<Problem> createStretchedV() {
00044     return std::make_unique<StretchedV>();
00045 }
00046
00047 std::unique_ptr<Problem> createAckleyOne() {
00048     return std::make_unique<AckleyOne>();
00049 }
00050
00051 std::unique_ptr<Problem> createAckleyTwo() {
00052     return std::make_unique<AckleyTwo>();
00053 }
00054
00055 std::unique_ptr<Problem> createEggHolder() {
00056     return std::make_unique<EggHolder>();
00057 }
00058
00059
00060
00061 // Array mapping integer IDs -> factory functions
00062 constexpr std::array<Creator, 10> creators = {
00063     &createSchwefel,
00064     &createDeJongOne,
00065     &createRosenbrock,
00066     &createRastrigin,
00067     &createGriewangk,
00068     &createSineEnvelope,
00069     &createStretchedV,
00070     &createAckleyOne,
00071     &createAckleyTwo,
00072     &createEggHolder
00073 };
00074
00075 } // anonymous namespace
00076
00077 std::unique_ptr<Problem> ProblemFactory::create(int id) {
00078     int index = id - 1; // Decrement index to array index
00079
00080     // Get Creator function
00081     if (index < 0 || static_cast<std::size_t>(index) >= creators.size())
00082         throw std::out_of_range("Invalid problem ID");
00083
00084     return creators[index](); // Call the factory function
00085 }
```

# 10.54 RunExperiments.cpp

```
00001 #include "RunExperiments.h"
```

```
00002
00003 #include <iostream>
00004 #include <fstream>
00005 #include <memory>
00006 #include <optional>
00007
00008 #include <External/json.hpp>
00009
00010 #include "ProblemFactory.h"
00011 #include "Problem/Problem.h"
00012 #include "SolutionBuilder.h"
00013 #include "Optimizer/OptimizerFactory.h"
00014 #include "debug.h"
00015
00016
00017 using json = nlohmann::json; // Alias JSON parsing library
00018
00019
00020
00021 bool RunExperiments::loadConfig(const std::string& inputFile) {
00022     debug::log("\nConfig Loading from:\t", inputFile);
00023
00024     std::ifstream file(inputFile);
00025
00026     if(!file) { // File failed to open
00027         std::cerr « "Cannot open input file: " « inputFile « "\n";
00028         return false;
00029     }
00030
00031     json j;
00032     file » j;
00033
00034     // Ensure experiments exist
00035     if(!j.contains("experiments") || !j["experiments"].is_array()) {
00036         std::cerr « "JSON does not contain 'experiments' array.\n";
00037         return false;
00038     }
00039
00040     std::vector<ExperimentConfig> experiments; // Stores experiment configs
00041
00042     for(const auto& item : j["experiments"]) {
00043         ExperimentConfig cfg;
00044
00045         // Basic fields
00046         cfg.experimentName = item.value("experiment_name", "");
00047         cfg.problemType = item.value("problem_type", 0);
00048         cfg.dimensions = item.value("dimensions", 0);
00049         cfg.lower = item.value("lower_bound", 0.0);
00050         cfg.upper = item.value("upper_bound", 0.0);
00051         cfg.seed = item.value("seed", 1);
00052
00053         // Optimizer fields
00054         if (item.contains("optimizer") && item["optimizer"].is_object()) {
00055             const auto& opt = item["optimizer"];
00056             cfg.optimizer = opt.value("type", "");
00057             cfg.maxIterations = opt.value("iterations", 1);
00058             cfg.neighborDelta = opt.value("delta", 0.0);
00059             cfg.numNeighbors = opt.value("num_neighbors", 0);
00060         } else { // No optimizer provided
00061             cfg.optimizer = "";
00062             cfg.maxIterations = 0;
00063             cfg.neighborDelta = 0.0;
00064             cfg.numNeighbors = 0;
00065         }
00066
00067         experiments.push_back(cfg);
00068
00069         debug::log(
00070             "\n\nExperiment Config Created for ",
00071             cfg.experimentName, "\t(", cfg.problemType, ")",
00072             "\n Range: [", cfg.lower, ", ", cfg.upper, "]",
00073             "\nDimensions: ", cfg.dimensions,
00074             "\nSeed: ", cfg.seed,
00075             "\nOptimizer: ", cfg.optimizer,
00076             "\nIterations: ", cfg.maxIterations,
00077             "\nNeighbors/Max Delta: ", cfg.numNeighbors,
00078             ", ", cfg.neighborDelta
00079         );
00080     }
00081
00082     // Store loaded configs as field
00083     this->configs = experiments;
00084
00085     return true;
00086 }
00087 /*
00088 bool RunExperiments::runExperiment(ExperimentConfig& config) {
```

```
00089     // Perform experiment setup
00090     std::unique_ptr<Problem> problem = ProblemFactory::create(config.problemType);
00091     SolutionBuilder builder(config.dimensions, config.upper, config.lower, config.seed);
00092     std::unique_ptr<Optimizer> optimizer = OptimizerFactory::initOptimizer(*problem, config, builder);
00093 
00094     // Perform experiment
00095     double execTime = optimizer->optimize();
00096 
00097     writeResults(
00098 
00099         optimizer->getBestSolution(),
00100         optimizer->getBestFitnesses(),
00101         optimizer->getSolutions()
00102     );
00103 }
00104 */
00105 
00106 bool writeCSV(
00107     const std::string& filename,
00108     const std::vector<std::vector<double>>& data,
00109     const std::optional<std::vector<std::string>>& rowLabels = std::nullopt,
00110     const std::optional<std::vector<std::string>>& colLabels = std::nullopt
00111 ) {
00112     std::ofstream file(filename); // Open file to write
00113 
00114     if(!file.is_open()) // Error opening file
00115         return false;
00116 
00117     // Write column labels
00118     if(colLabels) {
00119         if(rowLabels) // top-left empty cell if row and col labels
00120             file << ",";
00121 
00122         // Write all labels
00123         for(size_t j = 0; j < colLabels->size(); ++j) {
00124             if (j > 0) file << ",";
00125             file << (*colLabels)[j];
00126         }
00127 
00128         file << "\n";
00129     }
00130 
00131     // Write data rows
00132     for (size_t i = 0; i < data.size(); ++i) {
00133         if(rowLabels)
00134             file << (*rowLabels)[i] << ",";
00135 
00136         for(size_t j = 0; j < data[i].size(); ++j) {
00137             if (j > 0) file << ",";
00138             file << data[i][j];
00139         }
00140 
00141         file << "\n";
00142     }
00143 
00144     return true;
00145 }
00146 
00147 
00148 std::vector<std::string> RunExperiments::getNames(std::vector<ExperimentConfig> configs) {
00149     std::vector<std::string> names(configs.size());
00150 
00151     for(size_t i = 0; i < configs.size(); i++) {
00152         names[i] = configs[i].experimentName;
00153     }
00154 
00155     return names;
00156 }
00157 
00158 
00159 
00160 int RunExperiments::runExperiments() {
00161     int numExperiments = configs.size();
00162 
00163     // Create vectors to store all experiment results
00164     std::vector<double> runtimes(numExperiments);
00165     std::vector<std::vector<double>> fitnessResults(numExperiments);
00166 
00167     for(int i = 0; i < numExperiments; i++) {
00168         debug::log("\nRunning Experiment:\t", configs[i].experimentName);
00169         ExperimentConfig& config = configs[i];
00170 
00171         // Perform experiment setup
00172         std::unique_ptr<Problem> problem = ProblemFactory::create(config.problemType);
00173         SolutionBuilder builder(config.dimensions, config.upper, config.lower, config.seed);
00174         std::unique_ptr<Optimizer> optimizer = OptimizerFactory::initOptimizer(*problem, config,
      builder);
```

```
00175
00176         // Perform experiment
00177         runtimes[i] = optimizer->optimize();
00178         fitnessResults[i] = optimizer->getBestFitnesses();
00179
00180         // Display best found fitness and runtime for experiment
00181         std::cout « "\nFitness of " « optimizer->getBestFitness() « " found for experiment " «
      config.experimentName « " in " « runtimes[i] « " seconds.";
00182     }
00183
00184     std::vector<std::string> experimentNames = getNames(configs);
00185     std::string bestFitnessesPath = outputFile + "/" + std::string(bestFitnessesFile);
00186     std::string timesPath = outputFile + "/" + std::string(timesFile);
00187     std::vector<std::vector<double» timeWriteCSV(runtimes.size());
00188
00189     // Convert runtimes to csv-friendly format
00190     for(size_t i = 0; i < runtimes.size(); i++)
00191         timeWriteCSV[i] = { runtimes[i] };
00192
00193     // Write fitness data to csv
00194     writeCSV(
00195         bestFitnessesPath,
00196         fitnessResults,
00197         experimentNames
00198     );
00199
00200     // Write time data to csv
00201     writeCSV(
00202         timesPath,
00203         timeWriteCSV,
00204         experimentNames,
00205         std::vector<std::string>(1, "Execution Time")
00206     );
00207
00208     return numExperiments;
00209 }
```

## 10.55 SolutionBuilder.cpp

```
00001 #include "SolutionBuilder.h"
00002
00003
00004 std::vector<double> SolutionBuilder::getRand() {
00005     std::vector<double> solution(dimensions);
00006
00007     // Generate random solutions
00008     for(int i = 0; i < dimensions; i++) {{
00009             solution[i] = lower + (upper - lower) * mt.genrand_real1();
00010         }
00011     }
00012
00013     return solution;
00014 }
00015
00016 double SolutionBuilder::checkBounds(double value) {
00017     if(value > upper)
00018         value = upper;
00019     else if(value < lower)
00020         value = lower;
00021
00022     return value;
00023 }
00024
00025 std::vector<std::vector<double» SolutionBuilder::getNeighbors(
00026     const std::vector<double>& center,
00027     int numNeighbors,
00028     double maxDelta
00029 ) {
00030     std::vector<std::vector<double» solutions(numNeighbors, std::vector<double>(dimensions));
00031
00032     // Randomly generate neighbors within maxDelta for each dimension
00033     for(int i = 0; i < numNeighbors; i++) {
00034         for(int j = 0; j < dimensions; j++) {
00035             // Increment randomly within maxDelta range
00036             double delta = (2.0 * mt.genrand_real1() - 1) * maxDelta;
00037             solutions[i][j] = checkBounds(center[j] + delta);
00038         }
00039     }
00040
00041
00042     return solutions;
00043 }
```

# Index