#ifndef COURSEWORK\_ARRAY\_H

#define COURSEWORK\_ARRAY\_H

#include <iostream>

#include <algorithm>

template<typename T>

class Array {

T \*data;

size\_t \_size;

size\_t \_capacity;

public:

void resize() {

if (\_size == \_capacity) {

std::size\_t newCapacity = std::max(2.0, \_capacity \* 1.5);

reserve(newCapacity);

}

}

void resize(int newCapacity);

bool empty() {

return \_size == 0;

}

unsigned int size() const {

return \_size;

}

void clear() {

if (\_size > 0) {

delete[] data;

\_size = 0;

}

}

void reserve(int reserve\_size) {

if (\_size > reserve\_size) {

return;

} else {

T \*newData = new T[reserve\_size];

if (\_size > 0) {

for (int i = 0; i < \_size; i++) {

newData[i] = data[i];

}

}

data = newData;

}

\_capacity = reserve\_size;

}

void assign(const T &elem) {

resize();

T \*newData = new T[\_size + 1];

if (\_size > 0) {

for (size\_t i = 0; i < \_size; i++)

newData[i] = data[i];

delete[] data;

}

newData[\_size] = elem;

data = newData;

\_size++;

}

void remove\_last() {

(reinterpret\_cast<T \*>(data)[\_size-- - 1]).~T();

}

void swap(Array &other) {

using std::swap;

swap(\_capacity, other.\_capacity);

swap(\_size, other.\_size);

swap(data, other.data);

}

void remove(int pos) {

for (size\_t i = pos + 1; i < \_size; i++) {

data[i - 1] = data[i];

}

remove\_last();

}

~Array();

Array();

Array(const Array &other) : \_size(other.\_size), data(new T[other.\_size]), \_capacity(other.\_capacity) {

for (int i = 0; i < \_size; i++) {

data[i] = other.data[i];

}

}

T &at(int i) {

if (i < 0 || i > \_size - 1) throw std::logic\_error("Bounds error");

return data[i];

}

T &operator[](int i) {

return at(i);

}

Array<T> &operator=(Array<T> &rhs) {

rhs.swap(\*this);

return \*this;

}

explicit Array(const int reserve\_size) : \_size(0), data(new T[reserve\_size]), \_capacity(reserve\_size) {}

json jsonSerialize() {

json arr = json::array();

for (int i = 0; i < size(); ++i) {

auto t = at(i);

arr.push\_back(t->jsonSerialize());

}

return arr;

}

void jsonDeserialize(json &j);

};

template<typename T>

Array<T>::~Array() {

delete[] data;

}

template<typename T>

Array<T>::Array() : \_size(0), \_capacity(10000), data(new T[1]) {}

template<typename T>

void Array<T>::jsonDeserialize(json &j) {

for (auto& element : j) {

T t = T();

t->jsonDeserialize(element);

assign(t);

}

}

#endif //COURSEWORK\_ARRAY\_H

#include "Customer.h"

#include "ValidationException.h"

Customer::Customer(long id, float total, float discount)

: id(validateId(id)),

total(validateTotal(total)),

discount(validateDiscount(discount)) {}

long Customer::getId() const {

return id;

}

void Customer::setId(long id) {

Customer::id = validateId(id);

}

float Customer::getTotal() const {

return total;

}

void Customer::setTotal(float total) {

Customer::total = validateTotal(total);

}

float Customer::getDiscount() const {

return discount;

}

void Customer::setDiscount(float discount) {

Customer::discount = validateDiscount(discount);

}

std::ostream &operator<<(std::ostream &os, const Customer &customer) {

return os << "[Customer] id: " << customer.id << " total: " << customer.total << " discount: " << customer.discount;

}

long Customer::validateId(long id) {

if (id < 0) {

throw ValidationException("Wrong Customer id: should be greater than 0");

}

return id;

}

float Customer::validateTotal(float total) {

if (total < 0) {

throw ValidationException("Wrong Customer total: should be greater than 0");

}

return total;

}

float Customer::validateDiscount(float discount) {

if (discount < 0 || discount > 1) {

throw ValidationException("Wrong Customer discount: should be between 0 and 1");

}

return discount;

}

std::istream &operator>>(std::istream &is, Customer \*customer) {

do {

long tmp;

cout << "Enter customer id: " << endl;

is >> tmp;

try {

customer->setId(tmp);

} catch (ValidationException &e) {

cerr << e.what() << endl;

continue;

}

break;

} while (true);

do {

float tmp;

cout << "Enter customer total: " << endl;

is >> tmp;

try {

customer->setTotal(tmp);

} catch (ValidationException &e) {

cerr << e.what() << endl;

continue;

}

break;

} while (true);

while (true) {

float tmp;

cout << "Enter customer discount (0..1): " << endl;

is >> tmp;

try {

customer->setDiscount(tmp);

} catch (ValidationException &e) {

cerr << e.what() << endl;

continue;

}

break;

}

return is;

}

void Customer::write(ofstream &stream) {

stream.write((char \*) &id, sizeof(id));

stream.write((char \*) &total, sizeof(total));

stream.write((char \*) &discount, sizeof(discount));

}

json Customer::jsonSerialize() {

json obj;

obj["id"] = id;

obj["total"] = total;

obj["discount"] = discount;

return obj;

}

void Customer::read(ifstream &ifstream) {

ifstream.read((char \*) &id, sizeof(id));

ifstream.read((char \*) &total, sizeof(total));

ifstream.read((char \*) &discount, sizeof(discount));

}

template<class T>

Customer Customer::operator+(T plusTotal) {

total += plusTotal;

return \*this;

}

#ifndef COURSEWORK\_CUSTOMER\_H

#define COURSEWORK\_CUSTOMER\_H

#include <ostream>

#include "Writable.h"

#include "nlohmann/json.hpp"

#include "ValidationException.h"

#include "Readable.h"

using namespace std;

using namespace nlohmann;

class Customer: public Writable, public Readable {

long id = 0;

float total = 0;

float discount = 0;

protected:

static float validateTotal(float total);

static float validateDiscount(float discount);

static long validateId(long id);

public:

Customer(long id, float total, float discount);

Customer() = default;

long getId() const;

void setId(long id);

float getTotal() const;

void setTotal(float total);

float getDiscount() const;

void setDiscount(float discount);

friend std::ostream &operator<<(std::ostream &os, const Customer &customer);

friend std::istream &operator>>(std::istream &is, Customer \*customer);

void write(ofstream &stream);

json jsonSerialize();

void read(ifstream &ifstream);

template<class T>

Customer operator+(T plusTotal);

};

#endif //COURSEWORK\_CUSTOMER\_H

#include "DailyReport.h"

DailyReport::DailyReport(const Array<Order \*> &orders, const Date &date) : orders(orders), date(date) {}

const Array<Order \*> &DailyReport::getOrders() const {

return orders;

}

void DailyReport::setOrders(Array<Order \*> orders) {

DailyReport::orders = orders;

}

void DailyReport::clear() {

DailyReport::orders.clear();

}

const Date &DailyReport::getDate() const {

return date;

}

void DailyReport::setDate(const Date &date) {

DailyReport::date = date;

}

void DailyReport::addOrder(Order \*order) {

orders.assign(order);

}

Order \*DailyReport::operator[](int i) {

return orders[i];

}

ostream &operator<<(ostream &os, DailyReport \*report) {

os << "Order for " << report->date << endl;

auto \*table = new ATable::Table(ATable::DefaultAppearance(), "Orders");

auto productTables = new vector<ATable::Table \*>;

table->addColumn(new ATable::SimpleColumn("Order #", 20));

table->addColumn(new ATable::SimpleColumn("Customer Id", 20));

table->addColumn(new ATable::SimpleColumn("Total", 20));

table->addColumn(new ATable::SimpleColumn("With discount", 20));

for (int i = 0; i < report->orders.size(); ++i) {

Order \*order = (\*report)[i];

table->addCell(0, new ATable::IntegerCell(i + 1));

table->addCell(1, new ATable::IntegerCell(order->getCustomer()->getId()));

table->addCell(2, new ATable::FloatCell(order->getTotal()));

table->addCell(3, new ATable::FloatCell(order->getTotalPrice()));

auto productsTable = new ATable::Table(ATable::DefaultAppearance(), &"Order Products "[i]);

productsTable->addColumn(new ATable::SimpleColumn("Product #", 20));

productsTable->addColumn(new ATable::SimpleColumn("Name", 20));

productsTable->addColumn(new ATable::SimpleColumn("Price", 20));

productsTable->addColumn(new ATable::SimpleColumn("Has discount", 20));

for (int j = 0; j < order->getProducts().size(); ++j) {

Product \*product = (\*order)[j];

productsTable->addCell(0, new ATable::IntegerCell(j + 1));

productsTable->addCell(1, new ATable::StringCell(product->getName()));

productsTable->addCell(2, new ATable::FloatCell(product->getPrice()));

productsTable->addCell(3, new ATable::IntegerCell(product->isHasDiscount()));

}

productTables->push\_back(productsTable);

}

table->print(os);

for (int i = 0; i < productTables->size(); i++) {

os << endl << "Order #" << (i + 1) << " products: " << endl;

productTables[0][i]->print(os);

}

delete table;

delete productTables;

return os;

}

json DailyReport::jsonSerialize() {

json j;

j["date"] = date.jsonSerialize();

j["orders"] = orders.jsonSerialize();

return j;

}

void DailyReport::jsonDeserialize(ifstream &stream) {

json j;

stream >> j;

date = Date(j["date"][0], j["date"][1], j["date"][2]);

orders.jsonDeserialize(j["products"]);

}

void DailyReport::searchByText(const string &str) {

for (int i = 0; i < orders.size(); ++i) {

Array<Product \*> products = orders[i]->getProducts();

for (int j = 0; j < products.size(); ++j) {

if (products[j]->getName().find(str) == std::string::npos) {

cout << products[j] << endl;

}

}

}

}

void DailyReport::searchByNum(const int num) {

for (int i = 0; i < orders.size(); ++i) {

Array<Product \*> products = orders[i]->getProducts();

for (int j = 0; j < products.size(); ++j) {

if (products[j]->getPrice() == num) {

cout << \*products[j] << endl;

}

}

}

}

#ifndef COURSEWORK\_DAILYREPORT\_H

#define COURSEWORK\_DAILYREPORT\_H

#include <ostream>

#include "Order.h"

#include "Date.h"

#include "table/ATable.h"

#include "nlohmann/json.hpp"

using namespace nlohmann;

class DailyReport {

Array<Order \*> orders;

Date date;

public:

DailyReport(const Array<Order \*> &orders, const Date &date);

DailyReport() = default;

~DailyReport() = default;

const Array<Order \*> &getOrders() const;

Order \*operator[](int i);

void setOrders(Array<Order \*> orders);

const Date &getDate() const;

void setDate(const Date &date);

void addOrder(Order \*order);

friend ostream &operator<<(ostream &os, DailyReport \*report);

json jsonSerialize();

void clear();

void jsonDeserialize(ifstream &stream);

void searchByText(const string &str);

void searchByNum(const int num);

};

#endif //COURSEWORK\_DAILYREPORT\_H

#include "Date.h"

#include "ValidationException.h"

Date::Date(int year, int month, int day)

: year(year),

month(validateMonth(month)),

day(validateDay(day)) {}

int Date::getYear() const {

return year;

}

void Date::setYear(int year) {

Date::year = validateYear(year);

}

int Date::getMonth() const {

return month;

}

void Date::setMonth(int month) {

Date::month = validateDay(month);

}

int Date::getDay() const {

return day;

}

void Date::setDay(int day) {

Date::day = validateDay(day);

}

int Date::validateYear(int year) {

if (year < 1980 || year > 2100) {

throw ValidationException("Wrong Date year: are you time traveler?");

}

return year;

}

int Date::validateMonth(int month) {

if (month < 1 || month > 12) {

throw ValidationException("Wrong Date month: should be between 1 and 12");

}

return month;

}

int Date::validateDay(int day) {

if (day < 1 || day > 31) {

throw ValidationException("Wrong Date day: should be between 1 and 31");

}

return day;

}

ostream &operator<<(ostream &os, const Date &date) {

return os << date.year << "." << date.month << "." << date.day;

}

void Date::write(ofstream &stream) {

stream.write((char \*) &year, sizeof(year));

stream.write((char \*) &month, sizeof(month));

stream.write((char \*) &day, sizeof(day));

}

json Date::jsonSerialize() {

return json::array({year, month, day});

}

void Date::read(ifstream &ifstream) {

ifstream.read((char \*) &year, sizeof(year));

ifstream.read((char \*) &month, sizeof(month));

ifstream.read((char \*) &day, sizeof(day));

}

#ifndef COURSEWORK\_DATE\_H

#define COURSEWORK\_DATE\_H

#include "ValidationException.h"

#include "Writable.h"

#include "nlohmann/json.hpp"

#include "Readable.h"

using namespace std;

using namespace nlohmann;

class Date: public Writable, public Readable {

private:

int year = 0;

int month = 0;

int day = 0;

protected:

static int validateMonth(int month);

static int validateDay(int day);

static int validateYear(int year);

public:

Date(int year, int month, int day);

Date() = default;

int getYear() const;

void setYear(int year);

int getMonth() const;

void setMonth(int month);

int getDay() const;

void setDay(int day);

friend ostream &operator<<(ostream &os, const Date &date);

void write(ofstream &stream) override;

json jsonSerialize();

void read(ifstream &ifstream) override;

};

#endif //COURSEWORK\_DATE\_H

#include <iostream>#include "Date.h"

#include "ValidationException.h"

#include "Order.h"

#include "DailyReport.h"

using namespace std;

const int arrSize = 2;

string inputFilename() {

string filename;

while (true) {

cin.clear();

cin.sync();

cout << "Enter filename:" << endl;

getline(cin, filename);

if (filename.empty()) {

cerr << "Wrong filename" << endl;

continue;

}

return filename;

}

}

template<class T>

T averageProductsCount(Array<Order \*> orders) {

T sum = 0;

for (int i = 0; i < orders.size(); ++i) {

sum += orders[i]->getProducts().size();

}

return sum / orders.size();

}

template<typename T>

T averageTotal(Array<Order \*> orders) {

T sum = 0;

for (int i = 0; i < orders.size(); ++i) {

sum += orders[i]->getTotalPrice();

}

return sum / orders.size();

}

void handleCmd(const char cmd, DailyReport \*dailyReport) {

cout << "\n";

switch (cmd) {

case '1':

for (int i = 0; i < 1; ++i) {

cout << "Filling Order #" << i + 1 << " of " << arrSize << endl;

auto order = new Order();

auto customer = new Customer();

cin >> customer;

cout << endl;

order->setCustomer(customer);

for (int j = 0; j < arrSize; ++j) {

cout << "Filling Product #" << j + 1 << " of " << arrSize << endl;

auto product = new Product();

cin >> product;

cout << endl;

order->addProduct(product);

}

order->setDate(dailyReport->getDate());

dailyReport->addOrder(order);

}

break;

case '2':

cout << dailyReport << endl;

break;

case '3':

char choice;

do {

string filename;

cout << "Select file type:" << endl;

cout << "|1| Binary\n";

cout << "|2| Text\n";

cout << "|0| Back\n";

cout << "Enter option number:" << endl;

cin >> choice;

if (choice == '1' || choice == '2') {

filename = inputFilename();

ofstream file(filename, ios::binary);

if (!file.is\_open()) {

cerr << "Cannot open file" << endl;

continue;

}

if (choice == '1') {

unsigned int size = dailyReport->getOrders().size();

file.write((char \*) &size, sizeof(unsigned int));

for (int i = 0; i < size; i++) {

(\*dailyReport)[i]->write(file);

}

} else if (choice == '2') {

file << dailyReport->jsonSerialize().dump(4);

}

file.close();

break;

}

} while (choice != '0');

break;

case '4': {

string filename = inputFilename();

ifstream file(filename, ios::binary);

if (!file.is\_open()) {

cerr << "Cannot open file" << endl;

break;

}

do {

cout << "Select file type:" << endl;

cout << "|1| Binary\n";

cout << "|2| Text\n";

cout << "|0| Back\n";

cout << "Enter option number:" << endl;

cin >> choice;

dailyReport->clear();

if (choice == '1') {

int size;

file.read((char \*) &size, sizeof(int));

for (size\_t i = 0; i < size; i++) {

auto order = new Order();

order->read(file);

dailyReport->addOrder(order);

delete order;

}

cout << "Success!" << endl;

break;

} else if (choice == '2') {

dailyReport->jsonDeserialize(file);

cout << "Success!" << endl;

break;

}

} while (choice != '0');

file.close();

break;

}

case '5': {

do {

cout << "Select search type:" << endl;

cout << "|1| By text\n";

cout << "|2| By number\n";

cout << "|0| Back\n";

cout << "Enter option number:" << endl;

cin >> choice;

cin.clear();

cin.sync();

if (choice == '1') {

cout << "Enter text to search:" << endl;

string str;

getline(cin, str);

cout << "Result:" << endl;

dailyReport->searchByText(str);

choice = '0';

} else if (choice == '2') {

cout << "Enter number to search:" << endl;

int look;

cin >> look;

cout << "Result:" << endl;

dailyReport->searchByNum(look);

choice = '0';

}

} while (choice != '0');

break;

}

case '6': {

cout << "Products count: " << averageProductsCount<float>(dailyReport->getOrders()) << endl;

cout << "Average total: " << averageTotal<float>(dailyReport->getOrders()) << endl;

break;

}

default:

cerr << "Invalid option number" << endl;

break;

}

cout << "\n";

}

int main() {

auto dailyReport = new DailyReport();

dailyReport->setDate(Date(2019, 12, 22));

char option;

do {

cout << "Menu:\n\n";

cout << "|1| Create Objects\n";

cout << "|2| Print data\n";

cout << "|3| Write to file\n";

cout << "|4| Read from file\n";

cout << "|5| Search\n";

cout << "|6| Show average\n";

cout << "|0| Exit\n";

cout << "Enter option number:" << endl;

cin >> option;

try {

handleCmd(option, dailyReport);

} catch (ValidationException &e) {

cerr << e.what() << endl;

continue;

} catch (exception &e) {

cerr << "Unexpected error, interrupting: " << e.what() << endl;

break;

}

} while (option != '0');

}

#include "Order.h"

#include "ValidationException.h"

Order::Order(Customer \*customer, Array<Product \*> products, const Date &date, float totalPrice, float total)

: customer(customer),

products(products),

date(date),

totalWithDiscount(validateTotalWithDiscount(totalPrice)),

total(validateTotal(total)) {

recalculate();

}

Customer \*Order::getCustomer() const {

return customer;

}

void Order::setCustomer(Customer \*customer) {

Order::customer = customer;

}

const Array<Product \*> &Order::getProducts() const {

return products;

}

void Order::setProducts(Array<Product \*> products) {

Order::products = products;

recalculate();

}

const Date &Order::getDate() const {

return date;

}

void Order::setDate(const Date &date) {

Order::date = date;

}

float Order::getTotalPrice() const {

return totalWithDiscount;

}

void Order::setTotalPrice(float totalPrice) {

Order::totalWithDiscount = validateTotalWithDiscount(totalPrice);

}

float Order::getTotal() const {

return total;

}

void Order::setTotal(float total) {

Order::total = validateTotal(total);

}

float Order::validateTotal(float total) {

if (total < 0) {

throw ValidationException("Wrong Order total: should be greater than 0");

}

return total;

}

float Order::validateTotalWithDiscount(float totalWithDiscount) {

if (totalWithDiscount < 0 || total > totalWithDiscount) {

throw ValidationException("Wrong Order totalWithDiscount: should be greater than 0 and total price");

}

return totalWithDiscount;

}

void Order::addProduct(Product \*product) {

products.assign(product);

recalculate();

}

Product \*Order::operator[](int i) {

return products[i];

}

void Order::recalculate() {

for (int i = 0; i < products.size(); ++i) {

Order::total += products[i]->getPrice();

Order::totalWithDiscount += products[i]->getPrice();

if (products[i]->isHasDiscount()) {

Order::totalWithDiscount -= products[i]->getPrice() \* customer->getDiscount();

}

}

}

void Order::write(ofstream &stream) {

stream.write((char \*) &total, sizeof(total));

stream.write((char \*) &totalWithDiscount, sizeof(totalWithDiscount));

date.write(stream);

customer->write(stream);

unsigned int size = products.size();

stream.write((char \*) &size, sizeof(size));

for (int i = 0; i < size; ++i) {

products[i]->write(stream);

}

}

json Order::jsonSerialize() {

json obj;

obj["products"] = products.jsonSerialize();

obj["customer"] = customer->jsonSerialize();

obj["total"] = total;

obj["totalWithDiscount"] = totalWithDiscount;

return obj;

}

void Order::read(ifstream &stream) {

stream.read((char \*) &total, sizeof(float));

stream.read((char \*) &totalWithDiscount, sizeof(float));

date.read(stream);

customer = new Customer();

customer->read(stream);

int size;

stream.read((char \*) &size, sizeof(int));

for (size\_t i = 0; i < size; i++) {

auto product = new Product();

product->read(stream);

addProduct(product);

delete product;

}

}

template<class T>

Order Order::operator+(T plusTotal) {

totalWithDiscount += plusTotal;

return \*this;

}

void Order::jsonDeserialize(json j) {

date = Date(j["date"][0], j["date"][1], j["date"][2]);

total = j["total"];

totalWithDiscount = j["totalWithDiscount"];

}

#ifndef COURSEWORK\_ORDER\_H

#define COURSEWORK\_ORDER\_H

#include "Customer.h"

#include "Product.h"

#include "Date.h"

#include "Array.h"

class Order : public Writable, public Readable {

Customer \*customer = nullptr;

Array<Product \*> products;

Date date;

float total = 0;

float totalWithDiscount = 0;

protected:

static float validateTotal(float total);

float validateTotalWithDiscount(float totalWithDiscount);

public:

Order(Customer \*customer, Array<Product \*> products, const Date &date, float totalPrice, float total);

Order() = default;

~Order() = default;

Customer \*getCustomer() const;

void setCustomer(Customer \*customer);

const Array<Product \*> &getProducts() const;

Product \*operator[](int i);

void addProduct(Product \*product);

void setProducts(Array<Product \*> products);

const Date &getDate() const;

void setDate(const Date &date);

float getTotalPrice() const;

void setTotalPrice(float totalPrice);

float getTotal() const;

void setTotal(float total);

void recalculate();

void write(ofstream &stream) override;

json jsonSerialize();

void read(ifstream &stream) override;

template<class T>

Order operator+(T plusTotal);

void jsonDeserialize(json j);

};

#endif //COURSEWORK\_ORDER\_H

#include "Product.h"

#include "table/ATable.h"

Product::Product(const string &name, float price, bool hasDiscount)

: name(validateName(name)),

price(validatePrice(price)),

hasDiscount(validateHasDiscount(hasDiscount)) {}

const string &Product::getName() const {

return name;

}

void Product::setName(const string &name) {

Product::name = validateName(name);

}

float Product::getPrice() const {

return price;

}

void Product::setPrice(float price) {

Product::price = validatePrice(price);

}

bool Product::isHasDiscount() const {

return hasDiscount;

}

void Product::setHasDiscount(bool hasDiscount) {

Product::hasDiscount = validateHasDiscount(hasDiscount);

}

ostream &operator<<(ostream &os, const Product &product) {

ATable::Table\* table = new ATable::Table(ATable::DefaultAppearance(), "Test table"); // creating a new table with default borders

table->addColumn(new ATable::SimpleColumn("Name", 25));

table->addColumn(new ATable::SimpleColumn("Price", 15));

table->addColumn(new ATable::SimpleColumn("Has Discount", 20));

table->addCell(0, new ATable::StringCell(product.getName()));

table->addCell(1, new ATable::FloatCell(product.getPrice()));

table->addCell(2, new ATable::FloatCell(product.isHasDiscount()));

table->print(os);

return os;

}

istream &operator>>(istream &is, Product \*product) {

do {

string tmp;

cout << "Enter product name: " << endl;

is.clear();

is.sync();

getline(is, tmp);

try {

product->setName(tmp);

} catch (ValidationException &e) {

cerr << e.what() << endl;

continue;

}

break;

} while (true);

do {

float tmp;

cout << "Enter product price: " << endl;

is >> tmp;

try {

product->setPrice(tmp);

} catch (ValidationException &e) {

cerr << e.what() << endl;

continue;

}

break;

} while (true);

do {

bool tmp;

cout << "Enter has discount (0 = no): " << endl;

is >> tmp;

try {

product->setHasDiscount(tmp);

} catch (ValidationException &e) {

cerr << e.what() << endl;

continue;

}

break;

} while (true);

return is;

}

string Product::validateName(const string &name) {

if (name.length() < 3 || name.length() > 25) {

throw ValidationException("Product name should be longer than 3 and shorter than 25 symbols");

}

return name;

}

float Product::validatePrice(float price) {

if (price < 0) {

throw ValidationException("Product price should be greater than 0");

}

return price;

}

bool Product::validateHasDiscount(bool hasDiscount) {

return hasDiscount;

}

bool Product::operator==(const Product &rhs) const {

return name == rhs.name &&

price == rhs.price &&

hasDiscount == rhs.hasDiscount;

}

bool Product::operator!=(const Product &rhs) const {

return !(rhs == \*this);

}

void Product::write(ofstream &stream) {

int nameLength = name.length();

stream.write((char \*) &nameLength, sizeof(nameLength));

stream.write((char \*) &name[0], sizeof(char) \* nameLength);

stream.write((char \*) &price, sizeof(price));

stream.write((char \*) &hasDiscount, sizeof(hasDiscount));

}

json Product::jsonSerialize() {

json obj;

obj["name"] = name;

obj["price"] = price;

obj["hasDiscount"] = hasDiscount;

return obj;

}

void Product::read(ifstream &stream) {

int nameLength;

stream.read((char \*) &nameLength, sizeof(nameLength));

name.resize(nameLength);

stream.read((char \*) &name[0], nameLength \* sizeof(char));

stream.read((char \*) &price, sizeof(price));

stream.read((char \*) &hasDiscount, sizeof(hasDiscount));

}

template<class T>

Product Product::operator+(T plusPrice) {

price += plusPrice;

return \*this;

}

#ifndef COURSEWORK\_PRODUCT\_H

#define COURSEWORK\_PRODUCT\_H

#include <string>

#include <ostream>

#include "nlohmann/json.hpp"

#include "ValidationException.h"

#include "Readable.h"

#include "Writable.h"

using namespace std;

using namespace nlohmann;

class Product : public Readable, public Writable {

string name;

float price = 0;

bool hasDiscount = false;

protected:

static string validateName(const string &name);

static float validatePrice(float price);

static bool validateHasDiscount(bool hasDiscount);

public:

Product(const string &name, float price, bool hasDiscount);

Product() = default;

const string &getName() const;

void setName(const string &name);

float getPrice() const;

void setPrice(float price);

bool isHasDiscount() const;

void setHasDiscount(bool hasDiscount);

friend ostream &operator<<(ostream &os, const Product &product);

friend istream &operator>>(istream &is, Product \*product);

bool operator==(const Product &rhs) const;

bool operator!=(const Product &rhs) const;

void write(ofstream &stream) override;

json jsonSerialize();

void read(ifstream &stream) override;

template<class T>

Product operator+(T plusPrice);

};

#endif //COURSEWORK\_PRODUCT\_H

#ifndef COURSEWORK\_READABLE\_H

#define COURSEWORK\_READABLE\_H

#include "ValidationException.h"

class Readable {

public:

virtual void read(ifstream &stream) = 0;

};

#endif //COURSEWORK\_READABLE\_H

#ifndef COURSEWORK\_VALIDATIONEXCEPTION\_H

#define COURSEWORK\_VALIDATIONEXCEPTION\_H

#include <stdexcept>

#include <iostream>

#include <fstream>

#include <ostream>

using namespace std;

class ValidationException : public invalid\_argument {

public:

explicit ValidationException(const char \*what) : invalid\_argument(what) {}

};

#endif //COURSEWORK\_VALIDATIONEXCEPTION\_H

#ifndef COURSEWORK\_WRITABLE\_H

#define COURSEWORK\_WRITABLE\_H

#include "ValidationException.h"

class Writable {

public:

virtual void write(ofstream &stream) = 0;

};

#endif //COURSEWORK\_WRITABLE\_H

/\* \_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_

\_\_| | \_\_| | | | JSON for Modern C++

| | |\_\_ | | | | | | version 3.7.3

|\_\_\_\_\_|\_\_\_\_\_|\_\_\_\_\_|\_|\_\_\_| https://github.com/nlohmann/json

Licensed under the MIT License <http://opensource.org/licenses/MIT>.

SPDX-License-Identifier: MIT

Copyright (c) 2013-2019 Niels Lohmann <http://nlohmann.me>.

Permission is hereby granted, free of charge, to any person obtaining a copy

of this software and associated documentation files (the "Software"), to deal

in the Software without restriction, including without limitation the rights

to use, copy, modify, merge, publish, distribute, sublicense, and/or sell

copies of the Software, and to permit persons to whom the Software is

furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all

copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR

IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,

FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE

AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER

LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,

OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE

SOFTWARE.

\*/

#ifndef INCLUDE\_NLOHMANN\_JSON\_HPP\_

#define INCLUDE\_NLOHMANN\_JSON\_HPP\_

#define NLOHMANN\_JSON\_VERSION\_MAJOR 3

#define NLOHMANN\_JSON\_VERSION\_MINOR 7

#define NLOHMANN\_JSON\_VERSION\_PATCH 3

#include <algorithm> // all\_of, find, for\_each

#include <cassert> // assert

#include <ciso646> // and, not, or

#include <cstddef> // nullptr\_t, ptrdiff\_t, size\_t

#include <functional> // hash, less

#include <initializer\_list> // initializer\_list

#include <iosfwd> // istream, ostream

#include <iterator> // random\_access\_iterator\_tag

#include <memory> // unique\_ptr

#include <numeric> // accumulate

#include <string> // string, stoi, to\_string

#include <utility> // declval, forward, move, pair, swap

#include <vector> // vector

// #include <nlohmann/adl\_serializer.hpp>

#include <utility>

// #include <nlohmann/detail/conversions/from\_json.hpp>

#include <algorithm> // transform

#include <array> // array

#include <ciso646> // and, not

#include <forward\_list> // forward\_list

#include <iterator> // inserter, front\_inserter, end

#include <map> // map

#include <string> // string

#include <tuple> // tuple, make\_tuple

#include <type\_traits> // is\_arithmetic, is\_same, is\_enum, underlying\_type, is\_convertible

#include <unordered\_map> // unordered\_map

#include <utility> // pair, declval

#include <valarray> // valarray

// #include <nlohmann/detail/exceptions.hpp>

#include <exception> // exception

#include <stdexcept> // runtime\_error

#include <string> // to\_string

// #include <nlohmann/detail/input/position\_t.hpp>

#include <cstddef> // size\_t

namespace nlohmann

{

namespace detail

{

/// struct to capture the start position of the current token

struct position\_t

{

/// the total number of characters read

std::size\_t chars\_read\_total = 0;

/// the number of characters read in the current line

std::size\_t chars\_read\_current\_line = 0;

/// the number of lines read

std::size\_t lines\_read = 0;

/// conversion to size\_t to preserve SAX interface

constexpr operator size\_t() const

{

return chars\_read\_total;

}

};

} // namespace detail

} // namespace nlohmann

// #include <nlohmann/detail/macro\_scope.hpp>

#include <utility> // pair

// #include <nlohmann/thirdparty/hedley/hedley.hpp>

/\* Hedley - https://nemequ.github.io/hedley

\* Created by Evan Nemerson <evan@nemerson.com>

\*

\* To the extent possible under law, the author(s) have dedicated all

\* copyright and related and neighboring rights to this software to

\* the public domain worldwide. This software is distributed without

\* any warranty.

\*

\* For details, see <http://creativecommons.org/publicdomain/zero/1.0/>.

\* SPDX-License-Identifier: CC0-1.0

\*/

#if !defined(JSON\_HEDLEY\_VERSION) || (JSON\_HEDLEY\_VERSION < 11)

#if defined(JSON\_HEDLEY\_VERSION)

#undef JSON\_HEDLEY\_VERSION

#endif

#define JSON\_HEDLEY\_VERSION 11

#if defined(JSON\_HEDLEY\_STRINGIFY\_EX)

#undef JSON\_HEDLEY\_STRINGIFY\_EX

#endif

#define JSON\_HEDLEY\_STRINGIFY\_EX(x) #x

#if defined(JSON\_HEDLEY\_STRINGIFY)

#undef JSON\_HEDLEY\_STRINGIFY

#endif

#define JSON\_HEDLEY\_STRINGIFY(x) JSON\_HEDLEY\_STRINGIFY\_EX(x)

#if defined(JSON\_HEDLEY\_CONCAT\_EX)

#undef JSON\_HEDLEY\_CONCAT\_EX

#endif

#define JSON\_HEDLEY\_CONCAT\_EX(a,b) a##b

#if defined(JSON\_HEDLEY\_CONCAT)

#undef JSON\_HEDLEY\_CONCAT

#endif

#define JSON\_HEDLEY\_CONCAT(a,b) JSON\_HEDLEY\_CONCAT\_EX(a,b)

#if defined(JSON\_HEDLEY\_VERSION\_ENCODE)

#undef JSON\_HEDLEY\_VERSION\_ENCODE

#endif

#define JSON\_HEDLEY\_VERSION\_ENCODE(major,minor,revision) (((major) \* 1000000) + ((minor) \* 1000) + (revision))

#if defined(JSON\_HEDLEY\_VERSION\_DECODE\_MAJOR)

#undef JSON\_HEDLEY\_VERSION\_DECODE\_MAJOR

#endif

#define JSON\_HEDLEY\_VERSION\_DECODE\_MAJOR(version) ((version) / 1000000)

#if defined(JSON\_HEDLEY\_VERSION\_DECODE\_MINOR)

#undef JSON\_HEDLEY\_VERSION\_DECODE\_MINOR

#endif

#define JSON\_HEDLEY\_VERSION\_DECODE\_MINOR(version) (((version) % 1000000) / 1000)

#if defined(JSON\_HEDLEY\_VERSION\_DECODE\_REVISION)

#undef JSON\_HEDLEY\_VERSION\_DECODE\_REVISION

#endif

#define JSON\_HEDLEY\_VERSION\_DECODE\_REVISION(version) ((version) % 1000)

#if defined(JSON\_HEDLEY\_GNUC\_VERSION)

#undef JSON\_HEDLEY\_GNUC\_VERSION

#endif

#if defined(\_\_GNUC\_\_) && defined(\_\_GNUC\_PATCHLEVEL\_\_)

#define JSON\_HEDLEY\_GNUC\_VERSION JSON\_HEDLEY\_VERSION\_ENCODE(\_\_GNUC\_\_, \_\_GNUC\_MINOR\_\_, \_\_GNUC\_PATCHLEVEL\_\_)

#elif defined(\_\_GNUC\_\_)

#define JSON\_HEDLEY\_GNUC\_VERSION JSON\_HEDLEY\_VERSION\_ENCODE(\_\_GNUC\_\_, \_\_GNUC\_MINOR\_\_, 0)

#endif

#if defined(JSON\_HEDLEY\_GNUC\_VERSION\_CHECK)

#undef JSON\_HEDLEY\_GNUC\_VERSION\_CHECK

#endif

#if defined(JSON\_HEDLEY\_GNUC\_VERSION)

#define JSON\_HEDLEY\_GNUC\_VERSION\_CHECK(major,minor,patch) (JSON\_HEDLEY\_GNUC\_VERSION >= JSON\_HEDLEY\_VERSION\_ENCODE(major, minor, patch))

#else

#define JSON\_HEDLEY\_GNUC\_VERSION\_CHECK(major,minor,patch) (0)

#endif

#if defined(JSON\_HEDLEY\_MSVC\_VERSION)

#undef JSON\_HEDLEY\_MSVC\_VERSION

#endif

#if defined(\_MSC\_FULL\_VER) && (\_MSC\_FULL\_VER >= 140000000)

#define JSON\_HEDLEY\_MSVC\_VERSION JSON\_HEDLEY\_VERSION\_ENCODE(\_MSC\_FULL\_VER / 10000000, (\_MSC\_FULL\_VER % 10000000) / 100000, (\_MSC\_FULL\_VER % 100000) / 100)

#elif defined(\_MSC\_FULL\_VER)

#define JSON\_HEDLEY\_MSVC\_VERSION JSON\_HEDLEY\_VERSION\_ENCODE(\_MSC\_FULL\_VER / 1000000, (\_MSC\_FULL\_VER % 1000000) / 10000, (\_MSC\_FULL\_VER % 10000) / 10)

#elif defined(\_MSC\_VER)

#define JSON\_HEDLEY\_MSVC\_VERSION JSON\_HEDLEY\_VERSION\_ENCODE(\_MSC\_VER / 100, \_MSC\_VER % 100, 0)

#endif

#if defined(JSON\_HEDLEY\_MSVC\_VERSION\_CHECK)

#undef JSON\_HEDLEY\_MSVC\_VERSION\_CHECK

#endif

#if !defined(\_MSC\_VER)

#define JSON\_HEDLEY\_MSVC\_VERSION\_CHECK(major,minor,patch) (0)

#elif defined(\_MSC\_VER) && (\_MSC\_VER >= 1400)

#define JSON\_HEDLEY\_MSVC\_VERSION\_CHECK(major,minor,patch) (\_MSC\_FULL\_VER >= ((major \* 10000000) + (minor \* 100000) + (patch)))

#elif defined(\_MSC\_VER) && (\_MSC\_VER >= 1200)

#define JSON\_HEDLEY\_MSVC\_VERSION\_CHECK(major,minor,patch) (\_MSC\_FULL\_VER >= ((major \* 1000000) + (minor \* 10000) + (patch)))

#else

#define JSON\_HEDLEY\_MSVC\_VERSION\_CHECK(major,minor,patch) (\_MSC\_VER >= ((major \* 100) + (minor)))

#endif

#if defined(JSON\_HEDLEY\_INTEL\_VERSION)

#undef JSON\_HEDLEY\_INTEL\_VERSION

#endif

#if defined(\_\_INTEL\_COMPILER) && defined(\_\_INTEL\_COMPILER\_UPDATE)

#define JSON\_HEDLEY\_INTEL\_VERSION JSON\_HEDLEY\_VERSION\_ENCODE(\_\_INTEL\_COMPILER / 100, \_\_INTEL\_COMPILER % 100, \_\_INTEL\_COMPILER\_UPDATE)

#elif defined(\_\_INTEL\_COMPILER)

#define JSON\_HEDLEY\_INTEL\_VERSION JSON\_HEDLEY\_VERSION\_ENCODE(\_\_INTEL\_COMPILER / 100, \_\_INTEL\_COMPILER % 100, 0)

#endif

#if defined(JSON\_HEDLEY\_INTEL\_VERSION\_CHECK)

#undef JSON\_HEDLEY\_INTEL\_VERSION\_CHECK

#endif

#if defined(JSON\_HEDLEY\_INTEL\_VERSION)

#define JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(major,minor,patch) (JSON\_HEDLEY\_INTEL\_VERSION >= JSON\_HEDLEY\_VERSION\_ENCODE(major, minor, patch))

#else

#define JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(major,minor,patch) (0)

#endif

#if defined(JSON\_HEDLEY\_PGI\_VERSION)

#undef JSON\_HEDLEY\_PGI\_VERSION

#endif

#if defined(\_\_PGI) && defined(\_\_PGIC\_\_) && defined(\_\_PGIC\_MINOR\_\_) && defined(\_\_PGIC\_PATCHLEVEL\_\_)

#define JSON\_HEDLEY\_PGI\_VERSION JSON\_HEDLEY\_VERSION\_ENCODE(\_\_PGIC\_\_, \_\_PGIC\_MINOR\_\_, \_\_PGIC\_PATCHLEVEL\_\_)

#endif

#if defined(JSON\_HEDLEY\_PGI\_VERSION\_CHECK)

#undef JSON\_HEDLEY\_PGI\_VERSION\_CHECK

#endif

#if defined(JSON\_HEDLEY\_PGI\_VERSION)

#define JSON\_HEDLEY\_PGI\_VERSION\_CHECK(major,minor,patch) (JSON\_HEDLEY\_PGI\_VERSION >= JSON\_HEDLEY\_VERSION\_ENCODE(major, minor, patch))

#else

#define JSON\_HEDLEY\_PGI\_VERSION\_CHECK(major,minor,patch) (0)

#endif

#if defined(JSON\_HEDLEY\_SUNPRO\_VERSION)

#undef JSON\_HEDLEY\_SUNPRO\_VERSION

#endif

#if defined(\_\_SUNPRO\_C) && (\_\_SUNPRO\_C > 0x1000)

#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)

#elif defined(\_\_SUNPRO\_C)

#define JSON\_HEDLEY\_SUNPRO\_VERSION JSON\_HEDLEY\_VERSION\_ENCODE((\_\_SUNPRO\_C >> 8) & 0xf, (\_\_SUNPRO\_C >> 4) & 0xf, (\_\_SUNPRO\_C) & 0xf)

#elif defined(\_\_SUNPRO\_CC) && (\_\_SUNPRO\_CC > 0x1000)

#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)

#elif defined(\_\_SUNPRO\_CC)

#define JSON\_HEDLEY\_SUNPRO\_VERSION JSON\_HEDLEY\_VERSION\_ENCODE((\_\_SUNPRO\_CC >> 8) & 0xf, (\_\_SUNPRO\_CC >> 4) & 0xf, (\_\_SUNPRO\_CC) & 0xf)

#endif

#if defined(JSON\_HEDLEY\_SUNPRO\_VERSION\_CHECK)

#undef JSON\_HEDLEY\_SUNPRO\_VERSION\_CHECK

#endif

#if defined(JSON\_HEDLEY\_SUNPRO\_VERSION)

#define JSON\_HEDLEY\_SUNPRO\_VERSION\_CHECK(major,minor,patch) (JSON\_HEDLEY\_SUNPRO\_VERSION >= JSON\_HEDLEY\_VERSION\_ENCODE(major, minor, patch))

#else

#define JSON\_HEDLEY\_SUNPRO\_VERSION\_CHECK(major,minor,patch) (0)

#endif

#if defined(JSON\_HEDLEY\_EMSCRIPTEN\_VERSION)

#undef JSON\_HEDLEY\_EMSCRIPTEN\_VERSION

#endif

#if defined(\_\_EMSCRIPTEN\_\_)

#define JSON\_HEDLEY\_EMSCRIPTEN\_VERSION JSON\_HEDLEY\_VERSION\_ENCODE(\_\_EMSCRIPTEN\_major\_\_, \_\_EMSCRIPTEN\_minor\_\_, \_\_EMSCRIPTEN\_tiny\_\_)

#endif

#if defined(JSON\_HEDLEY\_EMSCRIPTEN\_VERSION\_CHECK)

#undef JSON\_HEDLEY\_EMSCRIPTEN\_VERSION\_CHECK

#endif

#if defined(JSON\_HEDLEY\_EMSCRIPTEN\_VERSION)

#define JSON\_HEDLEY\_EMSCRIPTEN\_VERSION\_CHECK(major,minor,patch) (JSON\_HEDLEY\_EMSCRIPTEN\_VERSION >= JSON\_HEDLEY\_VERSION\_ENCODE(major, minor, patch))

#else

#define JSON\_HEDLEY\_EMSCRIPTEN\_VERSION\_CHECK(major,minor,patch) (0)

#endif

#if defined(JSON\_HEDLEY\_ARM\_VERSION)

#undef JSON\_HEDLEY\_ARM\_VERSION

#endif

#if defined(\_\_CC\_ARM) && defined(\_\_ARMCOMPILER\_VERSION)

#define JSON\_HEDLEY\_ARM\_VERSION JSON\_HEDLEY\_VERSION\_ENCODE(\_\_ARMCOMPILER\_VERSION / 1000000, (\_\_ARMCOMPILER\_VERSION % 1000000) / 10000, (\_\_ARMCOMPILER\_VERSION % 10000) / 100)

#elif defined(\_\_CC\_ARM) && defined(\_\_ARMCC\_VERSION)

#define JSON\_HEDLEY\_ARM\_VERSION JSON\_HEDLEY\_VERSION\_ENCODE(\_\_ARMCC\_VERSION / 1000000, (\_\_ARMCC\_VERSION % 1000000) / 10000, (\_\_ARMCC\_VERSION % 10000) / 100)

#endif

#if defined(JSON\_HEDLEY\_ARM\_VERSION\_CHECK)

#undef JSON\_HEDLEY\_ARM\_VERSION\_CHECK

#endif

#if defined(JSON\_HEDLEY\_ARM\_VERSION)

#define JSON\_HEDLEY\_ARM\_VERSION\_CHECK(major,minor,patch) (JSON\_HEDLEY\_ARM\_VERSION >= JSON\_HEDLEY\_VERSION\_ENCODE(major, minor, patch))

#else

#define JSON\_HEDLEY\_ARM\_VERSION\_CHECK(major,minor,patch) (0)

#endif

#if defined(JSON\_HEDLEY\_IBM\_VERSION)

#undef JSON\_HEDLEY\_IBM\_VERSION

#endif

#if defined(\_\_ibmxl\_\_)

#define JSON\_HEDLEY\_IBM\_VERSION JSON\_HEDLEY\_VERSION\_ENCODE(\_\_ibmxl\_version\_\_, \_\_ibmxl\_release\_\_, \_\_ibmxl\_modification\_\_)

#elif defined(\_\_xlC\_\_) && defined(\_\_xlC\_ver\_\_)

#define JSON\_HEDLEY\_IBM\_VERSION JSON\_HEDLEY\_VERSION\_ENCODE(\_\_xlC\_\_ >> 8, \_\_xlC\_\_ & 0xff, (\_\_xlC\_ver\_\_ >> 8) & 0xff)

#elif defined(\_\_xlC\_\_)

#define JSON\_HEDLEY\_IBM\_VERSION JSON\_HEDLEY\_VERSION\_ENCODE(\_\_xlC\_\_ >> 8, \_\_xlC\_\_ & 0xff, 0)

#endif

#if defined(JSON\_HEDLEY\_IBM\_VERSION\_CHECK)

#undef JSON\_HEDLEY\_IBM\_VERSION\_CHECK

#endif

#if defined(JSON\_HEDLEY\_IBM\_VERSION)

#define JSON\_HEDLEY\_IBM\_VERSION\_CHECK(major,minor,patch) (JSON\_HEDLEY\_IBM\_VERSION >= JSON\_HEDLEY\_VERSION\_ENCODE(major, minor, patch))

#else

#define JSON\_HEDLEY\_IBM\_VERSION\_CHECK(major,minor,patch) (0)

#endif

#if defined(JSON\_HEDLEY\_TI\_VERSION)

#undef JSON\_HEDLEY\_TI\_VERSION

#endif

#if defined(\_\_TI\_COMPILER\_VERSION\_\_)

#define JSON\_HEDLEY\_TI\_VERSION JSON\_HEDLEY\_VERSION\_ENCODE(\_\_TI\_COMPILER\_VERSION\_\_ / 1000000, (\_\_TI\_COMPILER\_VERSION\_\_ % 1000000) / 1000, (\_\_TI\_COMPILER\_VERSION\_\_ % 1000))

#endif

#if defined(JSON\_HEDLEY\_TI\_VERSION\_CHECK)

#undef JSON\_HEDLEY\_TI\_VERSION\_CHECK

#endif

#if defined(JSON\_HEDLEY\_TI\_VERSION)

#define JSON\_HEDLEY\_TI\_VERSION\_CHECK(major,minor,patch) (JSON\_HEDLEY\_TI\_VERSION >= JSON\_HEDLEY\_VERSION\_ENCODE(major, minor, patch))

#else

#define JSON\_HEDLEY\_TI\_VERSION\_CHECK(major,minor,patch) (0)

#endif

#if defined(JSON\_HEDLEY\_CRAY\_VERSION)

#undef JSON\_HEDLEY\_CRAY\_VERSION

#endif

#if defined(\_CRAYC)

#if defined(\_RELEASE\_PATCHLEVEL)

#define JSON\_HEDLEY\_CRAY\_VERSION JSON\_HEDLEY\_VERSION\_ENCODE(\_RELEASE\_MAJOR, \_RELEASE\_MINOR, \_RELEASE\_PATCHLEVEL)

#else

#define JSON\_HEDLEY\_CRAY\_VERSION JSON\_HEDLEY\_VERSION\_ENCODE(\_RELEASE\_MAJOR, \_RELEASE\_MINOR, 0)

#endif

#endif

#if defined(JSON\_HEDLEY\_CRAY\_VERSION\_CHECK)

#undef JSON\_HEDLEY\_CRAY\_VERSION\_CHECK

#endif

#if defined(JSON\_HEDLEY\_CRAY\_VERSION)

#define JSON\_HEDLEY\_CRAY\_VERSION\_CHECK(major,minor,patch) (JSON\_HEDLEY\_CRAY\_VERSION >= JSON\_HEDLEY\_VERSION\_ENCODE(major, minor, patch))

#else

#define JSON\_HEDLEY\_CRAY\_VERSION\_CHECK(major,minor,patch) (0)

#endif

#if defined(JSON\_HEDLEY\_IAR\_VERSION)

#undef JSON\_HEDLEY\_IAR\_VERSION

#endif

#if defined(\_\_IAR\_SYSTEMS\_ICC\_\_)

#if \_\_VER\_\_ > 1000

#define JSON\_HEDLEY\_IAR\_VERSION JSON\_HEDLEY\_VERSION\_ENCODE((\_\_VER\_\_ / 1000000), ((\_\_VER\_\_ / 1000) % 1000), (\_\_VER\_\_ % 1000))

#else

#define JSON\_HEDLEY\_IAR\_VERSION JSON\_HEDLEY\_VERSION\_ENCODE(VER / 100, \_\_VER\_\_ % 100, 0)

#endif

#endif

#if defined(JSON\_HEDLEY\_IAR\_VERSION\_CHECK)

#undef JSON\_HEDLEY\_IAR\_VERSION\_CHECK

#endif

#if defined(JSON\_HEDLEY\_IAR\_VERSION)

#define JSON\_HEDLEY\_IAR\_VERSION\_CHECK(major,minor,patch) (JSON\_HEDLEY\_IAR\_VERSION >= JSON\_HEDLEY\_VERSION\_ENCODE(major, minor, patch))

#else

#define JSON\_HEDLEY\_IAR\_VERSION\_CHECK(major,minor,patch) (0)

#endif

#if defined(JSON\_HEDLEY\_TINYC\_VERSION)

#undef JSON\_HEDLEY\_TINYC\_VERSION

#endif

#if defined(\_\_TINYC\_\_)

#define JSON\_HEDLEY\_TINYC\_VERSION JSON\_HEDLEY\_VERSION\_ENCODE(\_\_TINYC\_\_ / 1000, (\_\_TINYC\_\_ / 100) % 10, \_\_TINYC\_\_ % 100)

#endif

#if defined(JSON\_HEDLEY\_TINYC\_VERSION\_CHECK)

#undef JSON\_HEDLEY\_TINYC\_VERSION\_CHECK

#endif

#if defined(JSON\_HEDLEY\_TINYC\_VERSION)

#define JSON\_HEDLEY\_TINYC\_VERSION\_CHECK(major,minor,patch) (JSON\_HEDLEY\_TINYC\_VERSION >= JSON\_HEDLEY\_VERSION\_ENCODE(major, minor, patch))

#else

#define JSON\_HEDLEY\_TINYC\_VERSION\_CHECK(major,minor,patch) (0)

#endif

#if defined(JSON\_HEDLEY\_DMC\_VERSION)

#undef JSON\_HEDLEY\_DMC\_VERSION

#endif

#if defined(\_\_DMC\_\_)

#define JSON\_HEDLEY\_DMC\_VERSION JSON\_HEDLEY\_VERSION\_ENCODE(\_\_DMC\_\_ >> 8, (\_\_DMC\_\_ >> 4) & 0xf, \_\_DMC\_\_ & 0xf)

#endif

#if defined(JSON\_HEDLEY\_DMC\_VERSION\_CHECK)

#undef JSON\_HEDLEY\_DMC\_VERSION\_CHECK

#endif

#if defined(JSON\_HEDLEY\_DMC\_VERSION)

#define JSON\_HEDLEY\_DMC\_VERSION\_CHECK(major,minor,patch) (JSON\_HEDLEY\_DMC\_VERSION >= JSON\_HEDLEY\_VERSION\_ENCODE(major, minor, patch))

#else

#define JSON\_HEDLEY\_DMC\_VERSION\_CHECK(major,minor,patch) (0)

#endif

#if defined(JSON\_HEDLEY\_COMPCERT\_VERSION)

#undef JSON\_HEDLEY\_COMPCERT\_VERSION

#endif

#if defined(\_\_COMPCERT\_VERSION\_\_)

#define JSON\_HEDLEY\_COMPCERT\_VERSION JSON\_HEDLEY\_VERSION\_ENCODE(\_\_COMPCERT\_VERSION\_\_ / 10000, (\_\_COMPCERT\_VERSION\_\_ / 100) % 100, \_\_COMPCERT\_VERSION\_\_ % 100)

#endif

#if defined(JSON\_HEDLEY\_COMPCERT\_VERSION\_CHECK)

#undef JSON\_HEDLEY\_COMPCERT\_VERSION\_CHECK

#endif

#if defined(JSON\_HEDLEY\_COMPCERT\_VERSION)

#define JSON\_HEDLEY\_COMPCERT\_VERSION\_CHECK(major,minor,patch) (JSON\_HEDLEY\_COMPCERT\_VERSION >= JSON\_HEDLEY\_VERSION\_ENCODE(major, minor, patch))

#else

#define JSON\_HEDLEY\_COMPCERT\_VERSION\_CHECK(major,minor,patch) (0)

#endif

#if defined(JSON\_HEDLEY\_PELLES\_VERSION)

#undef JSON\_HEDLEY\_PELLES\_VERSION

#endif

#if defined(\_\_POCC\_\_)

#define JSON\_HEDLEY\_PELLES\_VERSION JSON\_HEDLEY\_VERSION\_ENCODE(\_\_POCC\_\_ / 100, \_\_POCC\_\_ % 100, 0)

#endif

#if defined(JSON\_HEDLEY\_PELLES\_VERSION\_CHECK)

#undef JSON\_HEDLEY\_PELLES\_VERSION\_CHECK

#endif

#if defined(JSON\_HEDLEY\_PELLES\_VERSION)

#define JSON\_HEDLEY\_PELLES\_VERSION\_CHECK(major,minor,patch) (JSON\_HEDLEY\_PELLES\_VERSION >= JSON\_HEDLEY\_VERSION\_ENCODE(major, minor, patch))

#else

#define JSON\_HEDLEY\_PELLES\_VERSION\_CHECK(major,minor,patch) (0)

#endif

#if defined(JSON\_HEDLEY\_GCC\_VERSION)

#undef JSON\_HEDLEY\_GCC\_VERSION

#endif

#if \

defined(JSON\_HEDLEY\_GNUC\_VERSION) && \

!defined(\_\_clang\_\_) && \

!defined(JSON\_HEDLEY\_INTEL\_VERSION) && \

!defined(JSON\_HEDLEY\_PGI\_VERSION) && \

!defined(JSON\_HEDLEY\_ARM\_VERSION) && \

!defined(JSON\_HEDLEY\_TI\_VERSION) && \

!defined(\_\_COMPCERT\_\_)

#define JSON\_HEDLEY\_GCC\_VERSION JSON\_HEDLEY\_GNUC\_VERSION

#endif

#if defined(JSON\_HEDLEY\_GCC\_VERSION\_CHECK)

#undef JSON\_HEDLEY\_GCC\_VERSION\_CHECK

#endif

#if defined(JSON\_HEDLEY\_GCC\_VERSION)

#define JSON\_HEDLEY\_GCC\_VERSION\_CHECK(major,minor,patch) (JSON\_HEDLEY\_GCC\_VERSION >= JSON\_HEDLEY\_VERSION\_ENCODE(major, minor, patch))

#else

#define JSON\_HEDLEY\_GCC\_VERSION\_CHECK(major,minor,patch) (0)

#endif

#if defined(JSON\_HEDLEY\_HAS\_ATTRIBUTE)

#undef JSON\_HEDLEY\_HAS\_ATTRIBUTE

#endif

#if defined(\_\_has\_attribute)

#define JSON\_HEDLEY\_HAS\_ATTRIBUTE(attribute) \_\_has\_attribute(attribute)

#else

#define JSON\_HEDLEY\_HAS\_ATTRIBUTE(attribute) (0)

#endif

#if defined(JSON\_HEDLEY\_GNUC\_HAS\_ATTRIBUTE)

#undef JSON\_HEDLEY\_GNUC\_HAS\_ATTRIBUTE

#endif

#if defined(\_\_has\_attribute)

#define JSON\_HEDLEY\_GNUC\_HAS\_ATTRIBUTE(attribute,major,minor,patch) \_\_has\_attribute(attribute)

#else

#define JSON\_HEDLEY\_GNUC\_HAS\_ATTRIBUTE(attribute,major,minor,patch) JSON\_HEDLEY\_GNUC\_VERSION\_CHECK(major,minor,patch)

#endif

#if defined(JSON\_HEDLEY\_GCC\_HAS\_ATTRIBUTE)

#undef JSON\_HEDLEY\_GCC\_HAS\_ATTRIBUTE

#endif

#if defined(\_\_has\_attribute)

#define JSON\_HEDLEY\_GCC\_HAS\_ATTRIBUTE(attribute,major,minor,patch) \_\_has\_attribute(attribute)

#else

#define JSON\_HEDLEY\_GCC\_HAS\_ATTRIBUTE(attribute,major,minor,patch) JSON\_HEDLEY\_GCC\_VERSION\_CHECK(major,minor,patch)

#endif

#if defined(JSON\_HEDLEY\_HAS\_CPP\_ATTRIBUTE)

#undef JSON\_HEDLEY\_HAS\_CPP\_ATTRIBUTE

#endif

#if \

defined(\_\_has\_cpp\_attribute) && \

defined(\_\_cplusplus) && \

(!defined(JSON\_HEDLEY\_SUNPRO\_VERSION) || JSON\_HEDLEY\_SUNPRO\_VERSION\_CHECK(5,15,0))

#define JSON\_HEDLEY\_HAS\_CPP\_ATTRIBUTE(attribute) \_\_has\_cpp\_attribute(attribute)

#else

#define JSON\_HEDLEY\_HAS\_CPP\_ATTRIBUTE(attribute) (0)

#endif

#if defined(JSON\_HEDLEY\_HAS\_CPP\_ATTRIBUTE\_NS)

#undef JSON\_HEDLEY\_HAS\_CPP\_ATTRIBUTE\_NS

#endif

#if !defined(\_\_cplusplus) || !defined(\_\_has\_cpp\_attribute)

#define JSON\_HEDLEY\_HAS\_CPP\_ATTRIBUTE\_NS(ns,attribute) (0)

#elif \

!defined(JSON\_HEDLEY\_PGI\_VERSION) && \

(!defined(JSON\_HEDLEY\_SUNPRO\_VERSION) || JSON\_HEDLEY\_SUNPRO\_VERSION\_CHECK(5,15,0)) && \

(!defined(JSON\_HEDLEY\_MSVC\_VERSION) || JSON\_HEDLEY\_MSVC\_VERSION\_CHECK(19,20,0))

#define JSON\_HEDLEY\_HAS\_CPP\_ATTRIBUTE\_NS(ns,attribute) JSON\_HEDLEY\_HAS\_CPP\_ATTRIBUTE(ns::attribute)

#else

#define JSON\_HEDLEY\_HAS\_CPP\_ATTRIBUTE\_NS(ns,attribute) (0)

#endif

#if defined(JSON\_HEDLEY\_GNUC\_HAS\_CPP\_ATTRIBUTE)

#undef JSON\_HEDLEY\_GNUC\_HAS\_CPP\_ATTRIBUTE

#endif

#if defined(\_\_has\_cpp\_attribute) && defined(\_\_cplusplus)

#define JSON\_HEDLEY\_GNUC\_HAS\_CPP\_ATTRIBUTE(attribute,major,minor,patch) \_\_has\_cpp\_attribute(attribute)

#else

#define JSON\_HEDLEY\_GNUC\_HAS\_CPP\_ATTRIBUTE(attribute,major,minor,patch) JSON\_HEDLEY\_GNUC\_VERSION\_CHECK(major,minor,patch)

#endif

#if defined(JSON\_HEDLEY\_GCC\_HAS\_CPP\_ATTRIBUTE)

#undef JSON\_HEDLEY\_GCC\_HAS\_CPP\_ATTRIBUTE

#endif

#if defined(\_\_has\_cpp\_attribute) && defined(\_\_cplusplus)

#define JSON\_HEDLEY\_GCC\_HAS\_CPP\_ATTRIBUTE(attribute,major,minor,patch) \_\_has\_cpp\_attribute(attribute)

#else

#define JSON\_HEDLEY\_GCC\_HAS\_CPP\_ATTRIBUTE(attribute,major,minor,patch) JSON\_HEDLEY\_GCC\_VERSION\_CHECK(major,minor,patch)

#endif

#if defined(JSON\_HEDLEY\_HAS\_BUILTIN)

#undef JSON\_HEDLEY\_HAS\_BUILTIN

#endif

#if defined(\_\_has\_builtin)

#define JSON\_HEDLEY\_HAS\_BUILTIN(builtin) \_\_has\_builtin(builtin)

#else

#define JSON\_HEDLEY\_HAS\_BUILTIN(builtin) (0)

#endif

#if defined(JSON\_HEDLEY\_GNUC\_HAS\_BUILTIN)

#undef JSON\_HEDLEY\_GNUC\_HAS\_BUILTIN

#endif

#if defined(\_\_has\_builtin)

#define JSON\_HEDLEY\_GNUC\_HAS\_BUILTIN(builtin,major,minor,patch) \_\_has\_builtin(builtin)

#else

#define JSON\_HEDLEY\_GNUC\_HAS\_BUILTIN(builtin,major,minor,patch) JSON\_HEDLEY\_GNUC\_VERSION\_CHECK(major,minor,patch)

#endif

#if defined(JSON\_HEDLEY\_GCC\_HAS\_BUILTIN)

#undef JSON\_HEDLEY\_GCC\_HAS\_BUILTIN

#endif

#if defined(\_\_has\_builtin)

#define JSON\_HEDLEY\_GCC\_HAS\_BUILTIN(builtin,major,minor,patch) \_\_has\_builtin(builtin)

#else

#define JSON\_HEDLEY\_GCC\_HAS\_BUILTIN(builtin,major,minor,patch) JSON\_HEDLEY\_GCC\_VERSION\_CHECK(major,minor,patch)

#endif

#if defined(JSON\_HEDLEY\_HAS\_FEATURE)

#undef JSON\_HEDLEY\_HAS\_FEATURE

#endif

#if defined(\_\_has\_feature)

#define JSON\_HEDLEY\_HAS\_FEATURE(feature) \_\_has\_feature(feature)

#else

#define JSON\_HEDLEY\_HAS\_FEATURE(feature) (0)

#endif

#if defined(JSON\_HEDLEY\_GNUC\_HAS\_FEATURE)

#undef JSON\_HEDLEY\_GNUC\_HAS\_FEATURE

#endif

#if defined(\_\_has\_feature)

#define JSON\_HEDLEY\_GNUC\_HAS\_FEATURE(feature,major,minor,patch) \_\_has\_feature(feature)

#else

#define JSON\_HEDLEY\_GNUC\_HAS\_FEATURE(feature,major,minor,patch) JSON\_HEDLEY\_GNUC\_VERSION\_CHECK(major,minor,patch)

#endif

#if defined(JSON\_HEDLEY\_GCC\_HAS\_FEATURE)

#undef JSON\_HEDLEY\_GCC\_HAS\_FEATURE

#endif

#if defined(\_\_has\_feature)

#define JSON\_HEDLEY\_GCC\_HAS\_FEATURE(feature,major,minor,patch) \_\_has\_feature(feature)

#else

#define JSON\_HEDLEY\_GCC\_HAS\_FEATURE(feature,major,minor,patch) JSON\_HEDLEY\_GCC\_VERSION\_CHECK(major,minor,patch)

#endif

#if defined(JSON\_HEDLEY\_HAS\_EXTENSION)

#undef JSON\_HEDLEY\_HAS\_EXTENSION

#endif

#if defined(\_\_has\_extension)

#define JSON\_HEDLEY\_HAS\_EXTENSION(extension) \_\_has\_extension(extension)

#else

#define JSON\_HEDLEY\_HAS\_EXTENSION(extension) (0)

#endif

#if defined(JSON\_HEDLEY\_GNUC\_HAS\_EXTENSION)

#undef JSON\_HEDLEY\_GNUC\_HAS\_EXTENSION

#endif

#if defined(\_\_has\_extension)

#define JSON\_HEDLEY\_GNUC\_HAS\_EXTENSION(extension,major,minor,patch) \_\_has\_extension(extension)

#else

#define JSON\_HEDLEY\_GNUC\_HAS\_EXTENSION(extension,major,minor,patch) JSON\_HEDLEY\_GNUC\_VERSION\_CHECK(major,minor,patch)

#endif

#if defined(JSON\_HEDLEY\_GCC\_HAS\_EXTENSION)

#undef JSON\_HEDLEY\_GCC\_HAS\_EXTENSION

#endif

#if defined(\_\_has\_extension)

#define JSON\_HEDLEY\_GCC\_HAS\_EXTENSION(extension,major,minor,patch) \_\_has\_extension(extension)

#else

#define JSON\_HEDLEY\_GCC\_HAS\_EXTENSION(extension,major,minor,patch) JSON\_HEDLEY\_GCC\_VERSION\_CHECK(major,minor,patch)

#endif

#if defined(JSON\_HEDLEY\_HAS\_DECLSPEC\_ATTRIBUTE)

#undef JSON\_HEDLEY\_HAS\_DECLSPEC\_ATTRIBUTE

#endif

#if defined(\_\_has\_declspec\_attribute)

#define JSON\_HEDLEY\_HAS\_DECLSPEC\_ATTRIBUTE(attribute) \_\_has\_declspec\_attribute(attribute)

#else

#define JSON\_HEDLEY\_HAS\_DECLSPEC\_ATTRIBUTE(attribute) (0)

#endif

#if defined(JSON\_HEDLEY\_GNUC\_HAS\_DECLSPEC\_ATTRIBUTE)

#undef JSON\_HEDLEY\_GNUC\_HAS\_DECLSPEC\_ATTRIBUTE

#endif

#if defined(\_\_has\_declspec\_attribute)

#define JSON\_HEDLEY\_GNUC\_HAS\_DECLSPEC\_ATTRIBUTE(attribute,major,minor,patch) \_\_has\_declspec\_attribute(attribute)

#else

#define JSON\_HEDLEY\_GNUC\_HAS\_DECLSPEC\_ATTRIBUTE(attribute,major,minor,patch) JSON\_HEDLEY\_GNUC\_VERSION\_CHECK(major,minor,patch)

#endif

#if defined(JSON\_HEDLEY\_GCC\_HAS\_DECLSPEC\_ATTRIBUTE)

#undef JSON\_HEDLEY\_GCC\_HAS\_DECLSPEC\_ATTRIBUTE

#endif

#if defined(\_\_has\_declspec\_attribute)

#define JSON\_HEDLEY\_GCC\_HAS\_DECLSPEC\_ATTRIBUTE(attribute,major,minor,patch) \_\_has\_declspec\_attribute(attribute)

#else

#define JSON\_HEDLEY\_GCC\_HAS\_DECLSPEC\_ATTRIBUTE(attribute,major,minor,patch) JSON\_HEDLEY\_GCC\_VERSION\_CHECK(major,minor,patch)

#endif

#if defined(JSON\_HEDLEY\_HAS\_WARNING)

#undef JSON\_HEDLEY\_HAS\_WARNING

#endif

#if defined(\_\_has\_warning)

#define JSON\_HEDLEY\_HAS\_WARNING(warning) \_\_has\_warning(warning)

#else

#define JSON\_HEDLEY\_HAS\_WARNING(warning) (0)

#endif

#if defined(JSON\_HEDLEY\_GNUC\_HAS\_WARNING)

#undef JSON\_HEDLEY\_GNUC\_HAS\_WARNING

#endif

#if defined(\_\_has\_warning)

#define JSON\_HEDLEY\_GNUC\_HAS\_WARNING(warning,major,minor,patch) \_\_has\_warning(warning)

#else

#define JSON\_HEDLEY\_GNUC\_HAS\_WARNING(warning,major,minor,patch) JSON\_HEDLEY\_GNUC\_VERSION\_CHECK(major,minor,patch)

#endif

#if defined(JSON\_HEDLEY\_GCC\_HAS\_WARNING)

#undef JSON\_HEDLEY\_GCC\_HAS\_WARNING

#endif

#if defined(\_\_has\_warning)

#define JSON\_HEDLEY\_GCC\_HAS\_WARNING(warning,major,minor,patch) \_\_has\_warning(warning)

#else

#define JSON\_HEDLEY\_GCC\_HAS\_WARNING(warning,major,minor,patch) JSON\_HEDLEY\_GCC\_VERSION\_CHECK(major,minor,patch)

#endif

/\* JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_CPP98\_COMPAT\_WRAP\_ is for

HEDLEY INTERNAL USE ONLY. API subject to change without notice. \*/

#if defined(JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_CPP98\_COMPAT\_WRAP\_)

#undef JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_CPP98\_COMPAT\_WRAP\_

#endif

#if defined(\_\_cplusplus) && JSON\_HEDLEY\_HAS\_WARNING("-Wc++98-compat")

# define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_CPP98\_COMPAT\_WRAP\_(xpr) \

JSON\_HEDLEY\_DIAGNOSTIC\_PUSH \

\_Pragma("clang diagnostic ignored \"-Wc++98-compat\"") \

xpr \

JSON\_HEDLEY\_DIAGNOSTIC\_POP

#else

# define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_CPP98\_COMPAT\_WRAP\_(x) x

#endif

#if \

(defined(\_\_STDC\_VERSION\_\_) && (\_\_STDC\_VERSION\_\_ >= 199901L)) || \

defined(\_\_clang\_\_) || \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(3,0,0) || \

JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0) || \

JSON\_HEDLEY\_IAR\_VERSION\_CHECK(8,0,0) || \

JSON\_HEDLEY\_PGI\_VERSION\_CHECK(18,4,0) || \

JSON\_HEDLEY\_ARM\_VERSION\_CHECK(4,1,0) || \

JSON\_HEDLEY\_TI\_VERSION\_CHECK(6,0,0) || \

JSON\_HEDLEY\_CRAY\_VERSION\_CHECK(5,0,0) || \

JSON\_HEDLEY\_TINYC\_VERSION\_CHECK(0,9,17) || \

JSON\_HEDLEY\_SUNPRO\_VERSION\_CHECK(8,0,0) || \

(JSON\_HEDLEY\_IBM\_VERSION\_CHECK(10,1,0) && defined(\_\_C99\_PRAGMA\_OPERATOR))

#define JSON\_HEDLEY\_PRAGMA(value) \_Pragma(#value)

#elif JSON\_HEDLEY\_MSVC\_VERSION\_CHECK(15,0,0)

#define JSON\_HEDLEY\_PRAGMA(value) \_\_pragma(value)

#else

#define JSON\_HEDLEY\_PRAGMA(value)

#endif

#if defined(JSON\_HEDLEY\_DIAGNOSTIC\_PUSH)

#undef JSON\_HEDLEY\_DIAGNOSTIC\_PUSH

#endif

#if defined(JSON\_HEDLEY\_DIAGNOSTIC\_POP)

#undef JSON\_HEDLEY\_DIAGNOSTIC\_POP

#endif

#if defined(\_\_clang\_\_)

#define JSON\_HEDLEY\_DIAGNOSTIC\_PUSH \_Pragma("clang diagnostic push")

#define JSON\_HEDLEY\_DIAGNOSTIC\_POP \_Pragma("clang diagnostic pop")

#elif JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0)

#define JSON\_HEDLEY\_DIAGNOSTIC\_PUSH \_Pragma("warning(push)")

#define JSON\_HEDLEY\_DIAGNOSTIC\_POP \_Pragma("warning(pop)")

#elif JSON\_HEDLEY\_GCC\_VERSION\_CHECK(4,6,0)

#define JSON\_HEDLEY\_DIAGNOSTIC\_PUSH \_Pragma("GCC diagnostic push")

#define JSON\_HEDLEY\_DIAGNOSTIC\_POP \_Pragma("GCC diagnostic pop")

#elif JSON\_HEDLEY\_MSVC\_VERSION\_CHECK(15,0,0)

#define JSON\_HEDLEY\_DIAGNOSTIC\_PUSH \_\_pragma(warning(push))

#define JSON\_HEDLEY\_DIAGNOSTIC\_POP \_\_pragma(warning(pop))

#elif JSON\_HEDLEY\_ARM\_VERSION\_CHECK(5,6,0)

#define JSON\_HEDLEY\_DIAGNOSTIC\_PUSH \_Pragma("push")

#define JSON\_HEDLEY\_DIAGNOSTIC\_POP \_Pragma("pop")

#elif JSON\_HEDLEY\_TI\_VERSION\_CHECK(8,1,0)

#define JSON\_HEDLEY\_DIAGNOSTIC\_PUSH \_Pragma("diag\_push")

#define JSON\_HEDLEY\_DIAGNOSTIC\_POP \_Pragma("diag\_pop")

#elif JSON\_HEDLEY\_PELLES\_VERSION\_CHECK(2,90,0)

#define JSON\_HEDLEY\_DIAGNOSTIC\_PUSH \_Pragma("warning(push)")

#define JSON\_HEDLEY\_DIAGNOSTIC\_POP \_Pragma("warning(pop)")

#else

#define JSON\_HEDLEY\_DIAGNOSTIC\_PUSH

#define JSON\_HEDLEY\_DIAGNOSTIC\_POP

#endif

#if defined(JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_DEPRECATED)

#undef JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_DEPRECATED

#endif

#if JSON\_HEDLEY\_HAS\_WARNING("-Wdeprecated-declarations")

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_DEPRECATED \_Pragma("clang diagnostic ignored \"-Wdeprecated-declarations\"")

#elif JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0)

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_DEPRECATED \_Pragma("warning(disable:1478 1786)")

#elif JSON\_HEDLEY\_PGI\_VERSION\_CHECK(17,10,0)

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_DEPRECATED \_Pragma("diag\_suppress 1215,1444")

#elif JSON\_HEDLEY\_GCC\_VERSION\_CHECK(4,3,0)

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_DEPRECATED \_Pragma("GCC diagnostic ignored \"-Wdeprecated-declarations\"")

#elif JSON\_HEDLEY\_MSVC\_VERSION\_CHECK(15,0,0)

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_DEPRECATED \_\_pragma(warning(disable:4996))

#elif JSON\_HEDLEY\_TI\_VERSION\_CHECK(8,0,0)

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_DEPRECATED \_Pragma("diag\_suppress 1291,1718")

#elif JSON\_HEDLEY\_SUNPRO\_VERSION\_CHECK(5,13,0) && !defined(\_\_cplusplus)

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_DEPRECATED \_Pragma("error\_messages(off,E\_DEPRECATED\_ATT,E\_DEPRECATED\_ATT\_MESS)")

#elif JSON\_HEDLEY\_SUNPRO\_VERSION\_CHECK(5,13,0) && defined(\_\_cplusplus)

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_DEPRECATED \_Pragma("error\_messages(off,symdeprecated,symdeprecated2)")

#elif JSON\_HEDLEY\_IAR\_VERSION\_CHECK(8,0,0)

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_DEPRECATED \_Pragma("diag\_suppress=Pe1444,Pe1215")

#elif JSON\_HEDLEY\_PELLES\_VERSION\_CHECK(2,90,0)

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_DEPRECATED \_Pragma("warn(disable:2241)")

#else

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_DEPRECATED

#endif

#if defined(JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_UNKNOWN\_PRAGMAS)

#undef JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_UNKNOWN\_PRAGMAS

#endif

#if JSON\_HEDLEY\_HAS\_WARNING("-Wunknown-pragmas")

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_UNKNOWN\_PRAGMAS \_Pragma("clang diagnostic ignored \"-Wunknown-pragmas\"")

#elif JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0)

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_UNKNOWN\_PRAGMAS \_Pragma("warning(disable:161)")

#elif JSON\_HEDLEY\_PGI\_VERSION\_CHECK(17,10,0)

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_UNKNOWN\_PRAGMAS \_Pragma("diag\_suppress 1675")

#elif JSON\_HEDLEY\_GCC\_VERSION\_CHECK(4,3,0)

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_UNKNOWN\_PRAGMAS \_Pragma("GCC diagnostic ignored \"-Wunknown-pragmas\"")

#elif JSON\_HEDLEY\_MSVC\_VERSION\_CHECK(15,0,0)

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_UNKNOWN\_PRAGMAS \_\_pragma(warning(disable:4068))

#elif JSON\_HEDLEY\_TI\_VERSION\_CHECK(8,0,0)

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_UNKNOWN\_PRAGMAS \_Pragma("diag\_suppress 163")

#elif JSON\_HEDLEY\_IAR\_VERSION\_CHECK(8,0,0)

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_UNKNOWN\_PRAGMAS \_Pragma("diag\_suppress=Pe161")

#else

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_UNKNOWN\_PRAGMAS

#endif

#if defined(JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_UNKNOWN\_CPP\_ATTRIBUTES)

#undef JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_UNKNOWN\_CPP\_ATTRIBUTES

#endif

#if JSON\_HEDLEY\_HAS\_WARNING("-Wunknown-attributes")

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_UNKNOWN\_CPP\_ATTRIBUTES \_Pragma("clang diagnostic ignored \"-Wunknown-attributes\"")

#elif JSON\_HEDLEY\_GCC\_VERSION\_CHECK(4,6,0)

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_UNKNOWN\_CPP\_ATTRIBUTES \_Pragma("GCC diagnostic ignored \"-Wdeprecated-declarations\"")

#elif JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(17,0,0)

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_UNKNOWN\_CPP\_ATTRIBUTES \_Pragma("warning(disable:1292)")

#elif JSON\_HEDLEY\_MSVC\_VERSION\_CHECK(19,0,0)

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_UNKNOWN\_CPP\_ATTRIBUTES \_\_pragma(warning(disable:5030))

#elif JSON\_HEDLEY\_PGI\_VERSION\_CHECK(17,10,0)

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_UNKNOWN\_CPP\_ATTRIBUTES \_Pragma("diag\_suppress 1097")

#elif JSON\_HEDLEY\_SUNPRO\_VERSION\_CHECK(5,14,0) && defined(\_\_cplusplus)

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_UNKNOWN\_CPP\_ATTRIBUTES \_Pragma("error\_messages(off,attrskipunsup)")

#elif JSON\_HEDLEY\_TI\_VERSION\_CHECK(8,0,0)

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_UNKNOWN\_CPP\_ATTRIBUTES \_Pragma("diag\_suppress 1173")

#else

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_UNKNOWN\_CPP\_ATTRIBUTES

#endif

#if defined(JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_CAST\_QUAL)

#undef JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_CAST\_QUAL

#endif

#if JSON\_HEDLEY\_HAS\_WARNING("-Wcast-qual")

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_CAST\_QUAL \_Pragma("clang diagnostic ignored \"-Wcast-qual\"")

#elif JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0)

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_CAST\_QUAL \_Pragma("warning(disable:2203 2331)")

#elif JSON\_HEDLEY\_GCC\_VERSION\_CHECK(3,0,0)

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_CAST\_QUAL \_Pragma("GCC diagnostic ignored \"-Wcast-qual\"")

#else

#define JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_CAST\_QUAL

#endif

#if defined(JSON\_HEDLEY\_DEPRECATED)

#undef JSON\_HEDLEY\_DEPRECATED

#endif

#if defined(JSON\_HEDLEY\_DEPRECATED\_FOR)

#undef JSON\_HEDLEY\_DEPRECATED\_FOR

#endif

#if defined(\_\_cplusplus) && (\_\_cplusplus >= 201402L)

#define JSON\_HEDLEY\_DEPRECATED(since) JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_CPP98\_COMPAT\_WRAP\_([[deprecated("Since " #since)]])

#define JSON\_HEDLEY\_DEPRECATED\_FOR(since, replacement) JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_CPP98\_COMPAT\_WRAP\_([[deprecated("Since " #since "; use " #replacement)]])

#elif \

JSON\_HEDLEY\_HAS\_EXTENSION(attribute\_deprecated\_with\_message) || \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(4,5,0) || \

JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0) || \

JSON\_HEDLEY\_ARM\_VERSION\_CHECK(5,6,0) || \

JSON\_HEDLEY\_SUNPRO\_VERSION\_CHECK(5,13,0) || \

JSON\_HEDLEY\_PGI\_VERSION\_CHECK(17,10,0) || \

JSON\_HEDLEY\_TI\_VERSION\_CHECK(8,3,0)

#define JSON\_HEDLEY\_DEPRECATED(since) \_\_attribute\_\_((\_\_deprecated\_\_("Since " #since)))

#define JSON\_HEDLEY\_DEPRECATED\_FOR(since, replacement) \_\_attribute\_\_((\_\_deprecated\_\_("Since " #since "; use " #replacement)))

#elif \

JSON\_HEDLEY\_HAS\_ATTRIBUTE(deprecated) || \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(3,1,0) || \

JSON\_HEDLEY\_ARM\_VERSION\_CHECK(4,1,0) || \

JSON\_HEDLEY\_TI\_VERSION\_CHECK(8,0,0) || \

(JSON\_HEDLEY\_TI\_VERSION\_CHECK(7,3,0) && defined(\_\_TI\_GNU\_ATTRIBUTE\_SUPPORT\_\_))

#define JSON\_HEDLEY\_DEPRECATED(since) \_\_attribute\_\_((\_\_deprecated\_\_))

#define JSON\_HEDLEY\_DEPRECATED\_FOR(since, replacement) \_\_attribute\_\_((\_\_deprecated\_\_))

#elif JSON\_HEDLEY\_MSVC\_VERSION\_CHECK(14,0,0)

#define JSON\_HEDLEY\_DEPRECATED(since) \_\_declspec(deprecated("Since " # since))

#define JSON\_HEDLEY\_DEPRECATED\_FOR(since, replacement) \_\_declspec(deprecated("Since " #since "; use " #replacement))

#elif \

JSON\_HEDLEY\_MSVC\_VERSION\_CHECK(13,10,0) || \

JSON\_HEDLEY\_PELLES\_VERSION\_CHECK(6,50,0)

#define JSON\_HEDLEY\_DEPRECATED(since) \_\_declspec(deprecated)

#define JSON\_HEDLEY\_DEPRECATED\_FOR(since, replacement) \_\_declspec(deprecated)

#elif JSON\_HEDLEY\_IAR\_VERSION\_CHECK(8,0,0)

#define JSON\_HEDLEY\_DEPRECATED(since) \_Pragma("deprecated")

#define JSON\_HEDLEY\_DEPRECATED\_FOR(since, replacement) \_Pragma("deprecated")

#else

#define JSON\_HEDLEY\_DEPRECATED(since)

#define JSON\_HEDLEY\_DEPRECATED\_FOR(since, replacement)

#endif

#if defined(JSON\_HEDLEY\_UNAVAILABLE)

#undef JSON\_HEDLEY\_UNAVAILABLE

#endif

#if \

JSON\_HEDLEY\_HAS\_ATTRIBUTE(warning) || \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(4,3,0) || \

JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0)

#define JSON\_HEDLEY\_UNAVAILABLE(available\_since) \_\_attribute\_\_((\_\_warning\_\_("Not available until " #available\_since)))

#else

#define JSON\_HEDLEY\_UNAVAILABLE(available\_since)

#endif

#if defined(JSON\_HEDLEY\_WARN\_UNUSED\_RESULT)

#undef JSON\_HEDLEY\_WARN\_UNUSED\_RESULT

#endif

#if defined(\_\_cplusplus) && (\_\_cplusplus >= 201703L)

#define JSON\_HEDLEY\_WARN\_UNUSED\_RESULT JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_CPP98\_COMPAT\_WRAP\_([[nodiscard]])

#elif \

JSON\_HEDLEY\_HAS\_ATTRIBUTE(warn\_unused\_result) || \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(3,4,0) || \

JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0) || \

JSON\_HEDLEY\_TI\_VERSION\_CHECK(8,0,0) || \

(JSON\_HEDLEY\_TI\_VERSION\_CHECK(7,3,0) && defined(\_\_TI\_GNU\_ATTRIBUTE\_SUPPORT\_\_)) || \

(JSON\_HEDLEY\_SUNPRO\_VERSION\_CHECK(5,15,0) && defined(\_\_cplusplus)) || \

JSON\_HEDLEY\_PGI\_VERSION\_CHECK(17,10,0)

#define JSON\_HEDLEY\_WARN\_UNUSED\_RESULT \_\_attribute\_\_((\_\_warn\_unused\_result\_\_))

#elif defined(\_Check\_return\_) /\* SAL \*/

#define JSON\_HEDLEY\_WARN\_UNUSED\_RESULT \_Check\_return\_

#else

#define JSON\_HEDLEY\_WARN\_UNUSED\_RESULT

#endif

#if defined(JSON\_HEDLEY\_SENTINEL)

#undef JSON\_HEDLEY\_SENTINEL

#endif

#if \

JSON\_HEDLEY\_HAS\_ATTRIBUTE(sentinel) || \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(4,0,0) || \

JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0) || \

JSON\_HEDLEY\_ARM\_VERSION\_CHECK(5,4,0)

#define JSON\_HEDLEY\_SENTINEL(position) \_\_attribute\_\_((\_\_sentinel\_\_(position)))

#else

#define JSON\_HEDLEY\_SENTINEL(position)

#endif

#if defined(JSON\_HEDLEY\_NO\_RETURN)

#undef JSON\_HEDLEY\_NO\_RETURN

#endif

#if JSON\_HEDLEY\_IAR\_VERSION\_CHECK(8,0,0)

#define JSON\_HEDLEY\_NO\_RETURN \_\_noreturn

#elif JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0)

#define JSON\_HEDLEY\_NO\_RETURN \_\_attribute\_\_((\_\_noreturn\_\_))

#elif defined(\_\_STDC\_VERSION\_\_) && \_\_STDC\_VERSION\_\_ >= 201112L

#define JSON\_HEDLEY\_NO\_RETURN \_Noreturn

#elif defined(\_\_cplusplus) && (\_\_cplusplus >= 201103L)

#define JSON\_HEDLEY\_NO\_RETURN JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_CPP98\_COMPAT\_WRAP\_([[noreturn]])

#elif \

JSON\_HEDLEY\_HAS\_ATTRIBUTE(noreturn) || \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(3,2,0) || \

JSON\_HEDLEY\_SUNPRO\_VERSION\_CHECK(5,11,0) || \

JSON\_HEDLEY\_ARM\_VERSION\_CHECK(4,1,0) || \

JSON\_HEDLEY\_IBM\_VERSION\_CHECK(10,1,0) || \

JSON\_HEDLEY\_TI\_VERSION\_CHECK(18,0,0) || \

(JSON\_HEDLEY\_TI\_VERSION\_CHECK(17,3,0) && defined(\_\_TI\_GNU\_ATTRIBUTE\_SUPPORT\_\_))

#define JSON\_HEDLEY\_NO\_RETURN \_\_attribute\_\_((\_\_noreturn\_\_))

#elif JSON\_HEDLEY\_SUNPRO\_VERSION\_CHECK(5,10,0)

#define JSON\_HEDLEY\_NO\_RETURN \_Pragma("does\_not\_return")

#elif JSON\_HEDLEY\_MSVC\_VERSION\_CHECK(13,10,0)

#define JSON\_HEDLEY\_NO\_RETURN \_\_declspec(noreturn)

#elif JSON\_HEDLEY\_TI\_VERSION\_CHECK(6,0,0) && defined(\_\_cplusplus)

#define JSON\_HEDLEY\_NO\_RETURN \_Pragma("FUNC\_NEVER\_RETURNS;")

#elif JSON\_HEDLEY\_COMPCERT\_VERSION\_CHECK(3,2,0)

#define JSON\_HEDLEY\_NO\_RETURN \_\_attribute((noreturn))

#elif JSON\_HEDLEY\_PELLES\_VERSION\_CHECK(9,0,0)

#define JSON\_HEDLEY\_NO\_RETURN \_\_declspec(noreturn)

#else

#define JSON\_HEDLEY\_NO\_RETURN

#endif

#if defined(JSON\_HEDLEY\_NO\_ESCAPE)

#undef JSON\_HEDLEY\_NO\_ESCAPE

#endif

#if JSON\_HEDLEY\_HAS\_ATTRIBUTE(noescape)

#define JSON\_HEDLEY\_NO\_ESCAPE \_\_attribute\_\_((\_\_noescape\_\_))

#else

#define JSON\_HEDLEY\_NO\_ESCAPE

#endif

#if defined(JSON\_HEDLEY\_UNREACHABLE)

#undef JSON\_HEDLEY\_UNREACHABLE

#endif

#if defined(JSON\_HEDLEY\_UNREACHABLE\_RETURN)

#undef JSON\_HEDLEY\_UNREACHABLE\_RETURN

#endif

#if \

(JSON\_HEDLEY\_HAS\_BUILTIN(\_\_builtin\_unreachable) && (!defined(JSON\_HEDLEY\_ARM\_VERSION))) || \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(4,5,0) || \

JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0) || \

JSON\_HEDLEY\_IBM\_VERSION\_CHECK(13,1,5)

#define JSON\_HEDLEY\_UNREACHABLE() \_\_builtin\_unreachable()

#elif JSON\_HEDLEY\_MSVC\_VERSION\_CHECK(13,10,0)

#define JSON\_HEDLEY\_UNREACHABLE() \_\_assume(0)

#elif JSON\_HEDLEY\_TI\_VERSION\_CHECK(6,0,0)

#if defined(\_\_cplusplus)

#define JSON\_HEDLEY\_UNREACHABLE() std::\_nassert(0)

#else

#define JSON\_HEDLEY\_UNREACHABLE() \_nassert(0)

#endif

#define JSON\_HEDLEY\_UNREACHABLE\_RETURN(value) return value

#elif defined(EXIT\_FAILURE)

#define JSON\_HEDLEY\_UNREACHABLE() abort()

#else

#define JSON\_HEDLEY\_UNREACHABLE()

#define JSON\_HEDLEY\_UNREACHABLE\_RETURN(value) return value

#endif

#if !defined(JSON\_HEDLEY\_UNREACHABLE\_RETURN)

#define JSON\_HEDLEY\_UNREACHABLE\_RETURN(value) JSON\_HEDLEY\_UNREACHABLE()

#endif

#if defined(JSON\_HEDLEY\_ASSUME)

#undef JSON\_HEDLEY\_ASSUME

#endif

#if \

JSON\_HEDLEY\_MSVC\_VERSION\_CHECK(13,10,0) || \

JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0)

#define JSON\_HEDLEY\_ASSUME(expr) \_\_assume(expr)

#elif JSON\_HEDLEY\_HAS\_BUILTIN(\_\_builtin\_assume)

#define JSON\_HEDLEY\_ASSUME(expr) \_\_builtin\_assume(expr)

#elif JSON\_HEDLEY\_TI\_VERSION\_CHECK(6,0,0)

#if defined(\_\_cplusplus)

#define JSON\_HEDLEY\_ASSUME(expr) std::\_nassert(expr)

#else

#define JSON\_HEDLEY\_ASSUME(expr) \_nassert(expr)

#endif

#elif \

(JSON\_HEDLEY\_HAS\_BUILTIN(\_\_builtin\_unreachable) && !defined(JSON\_HEDLEY\_ARM\_VERSION)) || \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(4,5,0) || \

JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0) || \

JSON\_HEDLEY\_IBM\_VERSION\_CHECK(13,1,5)

#define JSON\_HEDLEY\_ASSUME(expr) ((void) ((expr) ? 1 : (\_\_builtin\_unreachable(), 1)))

#else

#define JSON\_HEDLEY\_ASSUME(expr) ((void) (expr))

#endif

JSON\_HEDLEY\_DIAGNOSTIC\_PUSH

#if JSON\_HEDLEY\_HAS\_WARNING("-Wpedantic")

#pragma clang diagnostic ignored "-Wpedantic"

#endif

#if JSON\_HEDLEY\_HAS\_WARNING("-Wc++98-compat-pedantic") && defined(\_\_cplusplus)

#pragma clang diagnostic ignored "-Wc++98-compat-pedantic"

#endif

#if JSON\_HEDLEY\_GCC\_HAS\_WARNING("-Wvariadic-macros",4,0,0)

#if defined(\_\_clang\_\_)

#pragma clang diagnostic ignored "-Wvariadic-macros"

#elif defined(JSON\_HEDLEY\_GCC\_VERSION)

#pragma GCC diagnostic ignored "-Wvariadic-macros"

#endif

#endif

#if defined(JSON\_HEDLEY\_NON\_NULL)

#undef JSON\_HEDLEY\_NON\_NULL

#endif

#if \

JSON\_HEDLEY\_HAS\_ATTRIBUTE(nonnull) || \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(3,3,0) || \

JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0) || \

JSON\_HEDLEY\_ARM\_VERSION\_CHECK(4,1,0)

#define JSON\_HEDLEY\_NON\_NULL(...) \_\_attribute\_\_((\_\_nonnull\_\_(\_\_VA\_ARGS\_\_)))

#else

#define JSON\_HEDLEY\_NON\_NULL(...)

#endif

JSON\_HEDLEY\_DIAGNOSTIC\_POP

#if defined(JSON\_HEDLEY\_PRINTF\_FORMAT)

#undef JSON\_HEDLEY\_PRINTF\_FORMAT

#endif

#if defined(\_\_MINGW32\_\_) && JSON\_HEDLEY\_GCC\_HAS\_ATTRIBUTE(format,4,4,0) && !defined(\_\_USE\_MINGW\_ANSI\_STDIO)

#define JSON\_HEDLEY\_PRINTF\_FORMAT(string\_idx,first\_to\_check) \_\_attribute\_\_((\_\_format\_\_(ms\_printf, string\_idx, first\_to\_check)))

#elif defined(\_\_MINGW32\_\_) && JSON\_HEDLEY\_GCC\_HAS\_ATTRIBUTE(format,4,4,0) && defined(\_\_USE\_MINGW\_ANSI\_STDIO)

#define JSON\_HEDLEY\_PRINTF\_FORMAT(string\_idx,first\_to\_check) \_\_attribute\_\_((\_\_format\_\_(gnu\_printf, string\_idx, first\_to\_check)))

#elif \

JSON\_HEDLEY\_HAS\_ATTRIBUTE(format) || \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(3,1,0) || \

JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0) || \

JSON\_HEDLEY\_ARM\_VERSION\_CHECK(5,6,0) || \

JSON\_HEDLEY\_IBM\_VERSION\_CHECK(10,1,0) || \

JSON\_HEDLEY\_TI\_VERSION\_CHECK(8,0,0) || \

(JSON\_HEDLEY\_TI\_VERSION\_CHECK(7,3,0) && defined(\_\_TI\_GNU\_ATTRIBUTE\_SUPPORT\_\_))

#define JSON\_HEDLEY\_PRINTF\_FORMAT(string\_idx,first\_to\_check) \_\_attribute\_\_((\_\_format\_\_(\_\_printf\_\_, string\_idx, first\_to\_check)))

#elif JSON\_HEDLEY\_PELLES\_VERSION\_CHECK(6,0,0)

#define JSON\_HEDLEY\_PRINTF\_FORMAT(string\_idx,first\_to\_check) \_\_declspec(vaformat(printf,string\_idx,first\_to\_check))

#else

#define JSON\_HEDLEY\_PRINTF\_FORMAT(string\_idx,first\_to\_check)

#endif

#if defined(JSON\_HEDLEY\_CONSTEXPR)

#undef JSON\_HEDLEY\_CONSTEXPR

#endif

#if defined(\_\_cplusplus)

#if \_\_cplusplus >= 201103L

#define JSON\_HEDLEY\_CONSTEXPR JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_CPP98\_COMPAT\_WRAP\_(constexpr)

#endif

#endif

#if !defined(JSON\_HEDLEY\_CONSTEXPR)

#define JSON\_HEDLEY\_CONSTEXPR

#endif

#if defined(JSON\_HEDLEY\_PREDICT)

#undef JSON\_HEDLEY\_PREDICT

#endif

#if defined(JSON\_HEDLEY\_LIKELY)

#undef JSON\_HEDLEY\_LIKELY

#endif

#if defined(JSON\_HEDLEY\_UNLIKELY)

#undef JSON\_HEDLEY\_UNLIKELY

#endif

#if defined(JSON\_HEDLEY\_UNPREDICTABLE)

#undef JSON\_HEDLEY\_UNPREDICTABLE

#endif

#if JSON\_HEDLEY\_HAS\_BUILTIN(\_\_builtin\_unpredictable)

#define JSON\_HEDLEY\_UNPREDICTABLE(expr) \_\_builtin\_unpredictable(!!(expr))

#endif

#if \

JSON\_HEDLEY\_HAS\_BUILTIN(\_\_builtin\_expect\_with\_probability) || \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(9,0,0)

# define JSON\_HEDLEY\_PREDICT(expr, value, probability) \_\_builtin\_expect\_with\_probability(expr, value, probability)

# define JSON\_HEDLEY\_PREDICT\_TRUE(expr, probability) \_\_builtin\_expect\_with\_probability(!!(expr), 1, probability)

# define JSON\_HEDLEY\_PREDICT\_FALSE(expr, probability) \_\_builtin\_expect\_with\_probability(!!(expr), 0, probability)

# define JSON\_HEDLEY\_LIKELY(expr) \_\_builtin\_expect(!!(expr), 1)

# define JSON\_HEDLEY\_UNLIKELY(expr) \_\_builtin\_expect(!!(expr), 0)

#if !defined(JSON\_HEDLEY\_BUILTIN\_UNPREDICTABLE)

#define JSON\_HEDLEY\_BUILTIN\_UNPREDICTABLE(expr) \_\_builtin\_expect\_with\_probability(!!(expr), 1, 0.5)

#endif

#elif \

JSON\_HEDLEY\_HAS\_BUILTIN(\_\_builtin\_expect) || \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(3,0,0) || \

JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0) || \

(JSON\_HEDLEY\_SUNPRO\_VERSION\_CHECK(5,15,0) && defined(\_\_cplusplus)) || \

JSON\_HEDLEY\_ARM\_VERSION\_CHECK(4,1,0) || \

JSON\_HEDLEY\_IBM\_VERSION\_CHECK(10,1,0) || \

JSON\_HEDLEY\_TI\_VERSION\_CHECK(6,1,0) || \

JSON\_HEDLEY\_TINYC\_VERSION\_CHECK(0,9,27)

# define JSON\_HEDLEY\_PREDICT(expr, expected, probability) \

(((probability) >= 0.9) ? \_\_builtin\_expect(!!(expr), (expected)) : (((void) (expected)), !!(expr)))

# define JSON\_HEDLEY\_PREDICT\_TRUE(expr, probability) \

(\_\_extension\_\_ ({ \

JSON\_HEDLEY\_CONSTEXPR double hedley\_probability\_ = (probability); \

((hedley\_probability\_ >= 0.9) ? \_\_builtin\_expect(!!(expr), 1) : ((hedley\_probability\_ <= 0.1) ? \_\_builtin\_expect(!!(expr), 0) : !!(expr))); \

}))

# define JSON\_HEDLEY\_PREDICT\_FALSE(expr, probability) \

(\_\_extension\_\_ ({ \

JSON\_HEDLEY\_CONSTEXPR double hedley\_probability\_ = (probability); \

((hedley\_probability\_ >= 0.9) ? \_\_builtin\_expect(!!(expr), 0) : ((hedley\_probability\_ <= 0.1) ? \_\_builtin\_expect(!!(expr), 1) : !!(expr))); \

}))

# define JSON\_HEDLEY\_LIKELY(expr) \_\_builtin\_expect(!!(expr), 1)

# define JSON\_HEDLEY\_UNLIKELY(expr) \_\_builtin\_expect(!!(expr), 0)

#else

# define JSON\_HEDLEY\_PREDICT(expr, expected, probability) (((void) (expected)), !!(expr))

# define JSON\_HEDLEY\_PREDICT\_TRUE(expr, probability) (!!(expr))

# define JSON\_HEDLEY\_PREDICT\_FALSE(expr, probability) (!!(expr))

# define JSON\_HEDLEY\_LIKELY(expr) (!!(expr))

# define JSON\_HEDLEY\_UNLIKELY(expr) (!!(expr))

#endif

#if !defined(JSON\_HEDLEY\_UNPREDICTABLE)

#define JSON\_HEDLEY\_UNPREDICTABLE(expr) JSON\_HEDLEY\_PREDICT(expr, 1, 0.5)

#endif

#if defined(JSON\_HEDLEY\_MALLOC)

#undef JSON\_HEDLEY\_MALLOC

#endif

#if \

JSON\_HEDLEY\_HAS\_ATTRIBUTE(malloc) || \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(3,1,0) || \

JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0) || \

JSON\_HEDLEY\_SUNPRO\_VERSION\_CHECK(5,11,0) || \

JSON\_HEDLEY\_ARM\_VERSION\_CHECK(4,1,0) || \

JSON\_HEDLEY\_IBM\_VERSION\_CHECK(12,1,0) || \

JSON\_HEDLEY\_TI\_VERSION\_CHECK(8,0,0) || \

(JSON\_HEDLEY\_TI\_VERSION\_CHECK(7,3,0) && defined(\_\_TI\_GNU\_ATTRIBUTE\_SUPPORT\_\_))

#define JSON\_HEDLEY\_MALLOC \_\_attribute\_\_((\_\_malloc\_\_))

#elif JSON\_HEDLEY\_SUNPRO\_VERSION\_CHECK(5,10,0)

#define JSON\_HEDLEY\_MALLOC \_Pragma("returns\_new\_memory")

#elif JSON\_HEDLEY\_MSVC\_VERSION\_CHECK(14, 0, 0)

#define JSON\_HEDLEY\_MALLOC \_\_declspec(restrict)

#else

#define JSON\_HEDLEY\_MALLOC

#endif

#if defined(JSON\_HEDLEY\_PURE)

#undef JSON\_HEDLEY\_PURE

#endif

#if \

JSON\_HEDLEY\_HAS\_ATTRIBUTE(pure) || \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(2,96,0) || \

JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0) || \

JSON\_HEDLEY\_SUNPRO\_VERSION\_CHECK(5,11,0) || \

JSON\_HEDLEY\_ARM\_VERSION\_CHECK(4,1,0) || \

JSON\_HEDLEY\_IBM\_VERSION\_CHECK(10,1,0) || \

JSON\_HEDLEY\_TI\_VERSION\_CHECK(8,0,0) || \

(JSON\_HEDLEY\_TI\_VERSION\_CHECK(7,3,0) && defined(\_\_TI\_GNU\_ATTRIBUTE\_SUPPORT\_\_)) || \

JSON\_HEDLEY\_PGI\_VERSION\_CHECK(17,10,0)

#define JSON\_HEDLEY\_PURE \_\_attribute\_\_((\_\_pure\_\_))

#elif JSON\_HEDLEY\_SUNPRO\_VERSION\_CHECK(5,10,0)

#define JSON\_HEDLEY\_PURE \_Pragma("does\_not\_write\_global\_data")

#elif JSON\_HEDLEY\_TI\_VERSION\_CHECK(6,0,0) && defined(\_\_cplusplus)

#define JSON\_HEDLEY\_PURE \_Pragma("FUNC\_IS\_PURE;")

#else

#define JSON\_HEDLEY\_PURE

#endif

#if defined(JSON\_HEDLEY\_CONST)

#undef JSON\_HEDLEY\_CONST

#endif

#if \

JSON\_HEDLEY\_HAS\_ATTRIBUTE(const) || \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(2,5,0) || \

JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0) || \

JSON\_HEDLEY\_SUNPRO\_VERSION\_CHECK(5,11,0) || \

JSON\_HEDLEY\_ARM\_VERSION\_CHECK(4,1,0) || \

JSON\_HEDLEY\_IBM\_VERSION\_CHECK(10,1,0) || \

JSON\_HEDLEY\_TI\_VERSION\_CHECK(8,0,0) || \

(JSON\_HEDLEY\_TI\_VERSION\_CHECK(7,3,0) && defined(\_\_TI\_GNU\_ATTRIBUTE\_SUPPORT\_\_)) || \

JSON\_HEDLEY\_PGI\_VERSION\_CHECK(17,10,0)

#define JSON\_HEDLEY\_CONST \_\_attribute\_\_((\_\_const\_\_))

#elif \

JSON\_HEDLEY\_SUNPRO\_VERSION\_CHECK(5,10,0)

#define JSON\_HEDLEY\_CONST \_Pragma("no\_side\_effect")

#else

#define JSON\_HEDLEY\_CONST JSON\_HEDLEY\_PURE

#endif

#if defined(JSON\_HEDLEY\_RESTRICT)

#undef JSON\_HEDLEY\_RESTRICT

#endif

#if defined(\_\_STDC\_VERSION\_\_) && (\_\_STDC\_VERSION\_\_ >= 199901L) && !defined(\_\_cplusplus)

#define JSON\_HEDLEY\_RESTRICT restrict

#elif \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(3,1,0) || \

JSON\_HEDLEY\_MSVC\_VERSION\_CHECK(14,0,0) || \

JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0) || \

JSON\_HEDLEY\_ARM\_VERSION\_CHECK(4,1,0) || \

JSON\_HEDLEY\_IBM\_VERSION\_CHECK(10,1,0) || \

JSON\_HEDLEY\_PGI\_VERSION\_CHECK(17,10,0) || \

JSON\_HEDLEY\_TI\_VERSION\_CHECK(8,0,0) || \

(JSON\_HEDLEY\_SUNPRO\_VERSION\_CHECK(5,14,0) && defined(\_\_cplusplus)) || \

JSON\_HEDLEY\_IAR\_VERSION\_CHECK(8,0,0) || \

defined(\_\_clang\_\_)

#define JSON\_HEDLEY\_RESTRICT \_\_restrict

#elif JSON\_HEDLEY\_SUNPRO\_VERSION\_CHECK(5,3,0) && !defined(\_\_cplusplus)

#define JSON\_HEDLEY\_RESTRICT \_Restrict

#else

#define JSON\_HEDLEY\_RESTRICT

#endif

#if defined(JSON\_HEDLEY\_INLINE)

#undef JSON\_HEDLEY\_INLINE

#endif

#if \

(defined(\_\_STDC\_VERSION\_\_) && (\_\_STDC\_VERSION\_\_ >= 199901L)) || \

(defined(\_\_cplusplus) && (\_\_cplusplus >= 199711L))

#define JSON\_HEDLEY\_INLINE inline

#elif \

defined(JSON\_HEDLEY\_GCC\_VERSION) || \

JSON\_HEDLEY\_ARM\_VERSION\_CHECK(6,2,0)

#define JSON\_HEDLEY\_INLINE \_\_inline\_\_

#elif \

JSON\_HEDLEY\_MSVC\_VERSION\_CHECK(12,0,0) || \

JSON\_HEDLEY\_ARM\_VERSION\_CHECK(4,1,0) || \

JSON\_HEDLEY\_TI\_VERSION\_CHECK(8,0,0)

#define JSON\_HEDLEY\_INLINE \_\_inline

#else

#define JSON\_HEDLEY\_INLINE

#endif

#if defined(JSON\_HEDLEY\_ALWAYS\_INLINE)

#undef JSON\_HEDLEY\_ALWAYS\_INLINE

#endif

#if \

JSON\_HEDLEY\_HAS\_ATTRIBUTE(always\_inline) || \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(4,0,0) || \

JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0) || \

JSON\_HEDLEY\_SUNPRO\_VERSION\_CHECK(5,11,0) || \

JSON\_HEDLEY\_ARM\_VERSION\_CHECK(4,1,0) || \

JSON\_HEDLEY\_IBM\_VERSION\_CHECK(10,1,0) || \

JSON\_HEDLEY\_TI\_VERSION\_CHECK(8,0,0) || \

(JSON\_HEDLEY\_TI\_VERSION\_CHECK(7,3,0) && defined(\_\_TI\_GNU\_ATTRIBUTE\_SUPPORT\_\_))

#define JSON\_HEDLEY\_ALWAYS\_INLINE \_\_attribute\_\_((\_\_always\_inline\_\_)) JSON\_HEDLEY\_INLINE

#elif JSON\_HEDLEY\_MSVC\_VERSION\_CHECK(12,0,0)

#define JSON\_HEDLEY\_ALWAYS\_INLINE \_\_forceinline

#elif JSON\_HEDLEY\_TI\_VERSION\_CHECK(7,0,0) && defined(\_\_cplusplus)

#define JSON\_HEDLEY\_ALWAYS\_INLINE \_Pragma("FUNC\_ALWAYS\_INLINE;")

#elif JSON\_HEDLEY\_IAR\_VERSION\_CHECK(8,0,0)

#define JSON\_HEDLEY\_ALWAYS\_INLINE \_Pragma("inline=forced")

#else

#define JSON\_HEDLEY\_ALWAYS\_INLINE JSON\_HEDLEY\_INLINE

#endif

#if defined(JSON\_HEDLEY\_NEVER\_INLINE)

#undef JSON\_HEDLEY\_NEVER\_INLINE

#endif

#if \

JSON\_HEDLEY\_HAS\_ATTRIBUTE(noinline) || \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(4,0,0) || \

JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0) || \

JSON\_HEDLEY\_SUNPRO\_VERSION\_CHECK(5,11,0) || \

JSON\_HEDLEY\_ARM\_VERSION\_CHECK(4,1,0) || \

JSON\_HEDLEY\_IBM\_VERSION\_CHECK(10,1,0) || \

JSON\_HEDLEY\_TI\_VERSION\_CHECK(8,0,0) || \

(JSON\_HEDLEY\_TI\_VERSION\_CHECK(7,3,0) && defined(\_\_TI\_GNU\_ATTRIBUTE\_SUPPORT\_\_))

#define JSON\_HEDLEY\_NEVER\_INLINE \_\_attribute\_\_((\_\_noinline\_\_))

#elif JSON\_HEDLEY\_MSVC\_VERSION\_CHECK(13,10,0)

#define JSON\_HEDLEY\_NEVER\_INLINE \_\_declspec(noinline)

#elif JSON\_HEDLEY\_PGI\_VERSION\_CHECK(10,2,0)

#define JSON\_HEDLEY\_NEVER\_INLINE \_Pragma("noinline")

#elif JSON\_HEDLEY\_TI\_VERSION\_CHECK(6,0,0) && defined(\_\_cplusplus)

#define JSON\_HEDLEY\_NEVER\_INLINE \_Pragma("FUNC\_CANNOT\_INLINE;")

#elif JSON\_HEDLEY\_IAR\_VERSION\_CHECK(8,0,0)

#define JSON\_HEDLEY\_NEVER\_INLINE \_Pragma("inline=never")

#elif JSON\_HEDLEY\_COMPCERT\_VERSION\_CHECK(3,2,0)

#define JSON\_HEDLEY\_NEVER\_INLINE \_\_attribute((noinline))

#elif JSON\_HEDLEY\_PELLES\_VERSION\_CHECK(9,0,0)

#define JSON\_HEDLEY\_NEVER\_INLINE \_\_declspec(noinline)

#else

#define JSON\_HEDLEY\_NEVER\_INLINE

#endif

#if defined(JSON\_HEDLEY\_PRIVATE)

#undef JSON\_HEDLEY\_PRIVATE

#endif

#if defined(JSON\_HEDLEY\_PUBLIC)

#undef JSON\_HEDLEY\_PUBLIC

#endif

#if defined(JSON\_HEDLEY\_IMPORT)

#undef JSON\_HEDLEY\_IMPORT

#endif

#if defined(\_WIN32) || defined(\_\_CYGWIN\_\_)

#define JSON\_HEDLEY\_PRIVATE

#define JSON\_HEDLEY\_PUBLIC \_\_declspec(dllexport)

#define JSON\_HEDLEY\_IMPORT \_\_declspec(dllimport)

#else

#if \

JSON\_HEDLEY\_HAS\_ATTRIBUTE(visibility) || \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(3,3,0) || \

JSON\_HEDLEY\_SUNPRO\_VERSION\_CHECK(5,11,0) || \

JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0) || \

JSON\_HEDLEY\_ARM\_VERSION\_CHECK(4,1,0) || \

JSON\_HEDLEY\_IBM\_VERSION\_CHECK(13,1,0) || \

JSON\_HEDLEY\_TI\_VERSION\_CHECK(8,0,0) || \

(JSON\_HEDLEY\_TI\_VERSION\_CHECK(7,3,0) && defined(\_\_TI\_EABI\_\_) && defined(\_\_TI\_GNU\_ATTRIBUTE\_SUPPORT\_\_))

#define JSON\_HEDLEY\_PRIVATE \_\_attribute\_\_((\_\_visibility\_\_("hidden")))

#define JSON\_HEDLEY\_PUBLIC \_\_attribute\_\_((\_\_visibility\_\_("default")))

#else

#define JSON\_HEDLEY\_PRIVATE

#define JSON\_HEDLEY\_PUBLIC

#endif

#define JSON\_HEDLEY\_IMPORT extern

#endif

#if defined(JSON\_HEDLEY\_NO\_THROW)

#undef JSON\_HEDLEY\_NO\_THROW

#endif

#if \

JSON\_HEDLEY\_HAS\_ATTRIBUTE(nothrow) || \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(3,3,0) || \

JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0)

#define JSON\_HEDLEY\_NO\_THROW \_\_attribute\_\_((\_\_nothrow\_\_))

#elif \

JSON\_HEDLEY\_MSVC\_VERSION\_CHECK(13,1,0) || \

JSON\_HEDLEY\_ARM\_VERSION\_CHECK(4,1,0)

#define JSON\_HEDLEY\_NO\_THROW \_\_declspec(nothrow)

#else

#define JSON\_HEDLEY\_NO\_THROW

#endif

#if defined(JSON\_HEDLEY\_FALL\_THROUGH)

#undef JSON\_HEDLEY\_FALL\_THROUGH

#endif

#if JSON\_HEDLEY\_GNUC\_HAS\_ATTRIBUTE(fallthrough,7,0,0) && !defined(JSON\_HEDLEY\_PGI\_VERSION)

#define JSON\_HEDLEY\_FALL\_THROUGH \_\_attribute\_\_((\_\_fallthrough\_\_))

#elif JSON\_HEDLEY\_HAS\_CPP\_ATTRIBUTE\_NS(clang,fallthrough)

#define JSON\_HEDLEY\_FALL\_THROUGH JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_CPP98\_COMPAT\_WRAP\_([[clang::fallthrough]])

#elif JSON\_HEDLEY\_HAS\_CPP\_ATTRIBUTE(fallthrough)

#define JSON\_HEDLEY\_FALL\_THROUGH JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_CPP98\_COMPAT\_WRAP\_([[fallthrough]])

#elif defined(\_\_fallthrough) /\* SAL \*/

#define JSON\_HEDLEY\_FALL\_THROUGH \_\_fallthrough

#else

#define JSON\_HEDLEY\_FALL\_THROUGH

#endif

#if defined(JSON\_HEDLEY\_RETURNS\_NON\_NULL)

#undef JSON\_HEDLEY\_RETURNS\_NON\_NULL

#endif

#if \

JSON\_HEDLEY\_HAS\_ATTRIBUTE(returns\_nonnull) || \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(4,9,0)

#define JSON\_HEDLEY\_RETURNS\_NON\_NULL \_\_attribute\_\_((\_\_returns\_nonnull\_\_))

#elif defined(\_Ret\_notnull\_) /\* SAL \*/

#define JSON\_HEDLEY\_RETURNS\_NON\_NULL \_Ret\_notnull\_

#else

#define JSON\_HEDLEY\_RETURNS\_NON\_NULL

#endif

#if defined(JSON\_HEDLEY\_ARRAY\_PARAM)

#undef JSON\_HEDLEY\_ARRAY\_PARAM

#endif

#if \

defined(\_\_STDC\_VERSION\_\_) && (\_\_STDC\_VERSION\_\_ >= 199901L) && \

!defined(\_\_STDC\_NO\_VLA\_\_) && \

!defined(\_\_cplusplus) && \

!defined(JSON\_HEDLEY\_PGI\_VERSION) && \

!defined(JSON\_HEDLEY\_TINYC\_VERSION)

#define JSON\_HEDLEY\_ARRAY\_PARAM(name) (name)

#else

#define JSON\_HEDLEY\_ARRAY\_PARAM(name)

#endif

#if defined(JSON\_HEDLEY\_IS\_CONSTANT)

#undef JSON\_HEDLEY\_IS\_CONSTANT

#endif

#if defined(JSON\_HEDLEY\_REQUIRE\_CONSTEXPR)

#undef JSON\_HEDLEY\_REQUIRE\_CONSTEXPR

#endif

/\* JSON\_HEDLEY\_IS\_CONSTEXPR\_ is for

HEDLEY INTERNAL USE ONLY. API subject to change without notice. \*/

#if defined(JSON\_HEDLEY\_IS\_CONSTEXPR\_)

#undef JSON\_HEDLEY\_IS\_CONSTEXPR\_

#endif

#if \

JSON\_HEDLEY\_HAS\_BUILTIN(\_\_builtin\_constant\_p) || \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(3,4,0) || \

JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0) || \

JSON\_HEDLEY\_TINYC\_VERSION\_CHECK(0,9,19) || \

JSON\_HEDLEY\_ARM\_VERSION\_CHECK(4,1,0) || \

JSON\_HEDLEY\_IBM\_VERSION\_CHECK(13,1,0) || \

JSON\_HEDLEY\_TI\_VERSION\_CHECK(6,1,0) || \

(JSON\_HEDLEY\_SUNPRO\_VERSION\_CHECK(5,10,0) && !defined(\_\_cplusplus)) || \

JSON\_HEDLEY\_CRAY\_VERSION\_CHECK(8,1,0)

#define JSON\_HEDLEY\_IS\_CONSTANT(expr) \_\_builtin\_constant\_p(expr)

#endif

#if !defined(\_\_cplusplus)

# if \

JSON\_HEDLEY\_HAS\_BUILTIN(\_\_builtin\_types\_compatible\_p) || \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(3,4,0) || \

JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0) || \

JSON\_HEDLEY\_IBM\_VERSION\_CHECK(13,1,0) || \

JSON\_HEDLEY\_CRAY\_VERSION\_CHECK(8,1,0) || \

JSON\_HEDLEY\_ARM\_VERSION\_CHECK(5,4,0) || \

JSON\_HEDLEY\_TINYC\_VERSION\_CHECK(0,9,24)

#if defined(\_\_INTPTR\_TYPE\_\_)

#define JSON\_HEDLEY\_IS\_CONSTEXPR\_(expr) \_\_builtin\_types\_compatible\_p(\_\_typeof\_\_((1 ? (void\*) ((\_\_INTPTR\_TYPE\_\_) ((expr) \* 0)) : (int\*) 0)), int\*)

#else

#include <stdint.h>

#define JSON\_HEDLEY\_IS\_CONSTEXPR\_(expr) \_\_builtin\_types\_compatible\_p(\_\_typeof\_\_((1 ? (void\*) ((intptr\_t) ((expr) \* 0)) : (int\*) 0)), int\*)

#endif

# elif \

(defined(\_\_STDC\_VERSION\_\_) && (\_\_STDC\_VERSION\_\_ >= 201112L) && !defined(JSON\_HEDLEY\_SUNPRO\_VERSION) && !defined(JSON\_HEDLEY\_PGI\_VERSION)) || \

JSON\_HEDLEY\_HAS\_EXTENSION(c\_generic\_selections) || \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(4,9,0) || \

JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(17,0,0) || \

JSON\_HEDLEY\_IBM\_VERSION\_CHECK(12,1,0) || \

JSON\_HEDLEY\_ARM\_VERSION\_CHECK(5,3,0)

#if defined(\_\_INTPTR\_TYPE\_\_)

#define JSON\_HEDLEY\_IS\_CONSTEXPR\_(expr) \_Generic((1 ? (void\*) ((\_\_INTPTR\_TYPE\_\_) ((expr) \* 0)) : (int\*) 0), int\*: 1, void\*: 0)

#else

#include <stdint.h>

#define JSON\_HEDLEY\_IS\_CONSTEXPR\_(expr) \_Generic((1 ? (void\*) ((intptr\_t) \* 0) : (int\*) 0), int\*: 1, void\*: 0)

#endif

# elif \

defined(JSON\_HEDLEY\_GCC\_VERSION) || \

defined(JSON\_HEDLEY\_INTEL\_VERSION) || \

defined(JSON\_HEDLEY\_TINYC\_VERSION) || \

defined(JSON\_HEDLEY\_TI\_VERSION) || \

defined(\_\_clang\_\_)

# define JSON\_HEDLEY\_IS\_CONSTEXPR\_(expr) ( \

sizeof(void) != \

sizeof(\*( \

1 ? \

((void\*) ((expr) \* 0L) ) : \

((struct { char v[sizeof(void) \* 2]; } \*) 1) \

) \

) \

)

# endif

#endif

#if defined(JSON\_HEDLEY\_IS\_CONSTEXPR\_)

#if !defined(JSON\_HEDLEY\_IS\_CONSTANT)

#define JSON\_HEDLEY\_IS\_CONSTANT(expr) JSON\_HEDLEY\_IS\_CONSTEXPR\_(expr)

#endif

#define JSON\_HEDLEY\_REQUIRE\_CONSTEXPR(expr) (JSON\_HEDLEY\_IS\_CONSTEXPR\_(expr) ? (expr) : (-1))

#else

#if !defined(JSON\_HEDLEY\_IS\_CONSTANT)

#define JSON\_HEDLEY\_IS\_CONSTANT(expr) (0)

#endif

#define JSON\_HEDLEY\_REQUIRE\_CONSTEXPR(expr) (expr)

#endif

#if defined(JSON\_HEDLEY\_BEGIN\_C\_DECLS)

#undef JSON\_HEDLEY\_BEGIN\_C\_DECLS

#endif

#if defined(JSON\_HEDLEY\_END\_C\_DECLS)

#undef JSON\_HEDLEY\_END\_C\_DECLS

#endif

#if defined(JSON\_HEDLEY\_C\_DECL)

#undef JSON\_HEDLEY\_C\_DECL

#endif

#if defined(\_\_cplusplus)

#define JSON\_HEDLEY\_BEGIN\_C\_DECLS extern "C" {

#define JSON\_HEDLEY\_END\_C\_DECLS }

#define JSON\_HEDLEY\_C\_DECL extern "C"

#else

#define JSON\_HEDLEY\_BEGIN\_C\_DECLS

#define JSON\_HEDLEY\_END\_C\_DECLS

#define JSON\_HEDLEY\_C\_DECL

#endif

#if defined(JSON\_HEDLEY\_STATIC\_ASSERT)

#undef JSON\_HEDLEY\_STATIC\_ASSERT

#endif

#if \

!defined(\_\_cplusplus) && ( \

(defined(\_\_STDC\_VERSION\_\_) && (\_\_STDC\_VERSION\_\_ >= 201112L)) || \

JSON\_HEDLEY\_HAS\_FEATURE(c\_static\_assert) || \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(6,0,0) || \

JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0) || \

defined(\_Static\_assert) \

)

# define JSON\_HEDLEY\_STATIC\_ASSERT(expr, message) \_Static\_assert(expr, message)

#elif \

(defined(\_\_cplusplus) && (\_\_cplusplus >= 201103L)) || \

JSON\_HEDLEY\_MSVC\_VERSION\_CHECK(16,0,0) || \

(defined(\_\_cplusplus) && JSON\_HEDLEY\_TI\_VERSION\_CHECK(8,3,0))

# define JSON\_HEDLEY\_STATIC\_ASSERT(expr, message) JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_CPP98\_COMPAT\_WRAP\_(static\_assert(expr, message))

#else

# define JSON\_HEDLEY\_STATIC\_ASSERT(expr, message)

#endif

#if defined(JSON\_HEDLEY\_CONST\_CAST)

#undef JSON\_HEDLEY\_CONST\_CAST

#endif

#if defined(\_\_cplusplus)

# define JSON\_HEDLEY\_CONST\_CAST(T, expr) (const\_cast<T>(expr))

#elif \

JSON\_HEDLEY\_HAS\_WARNING("-Wcast-qual") || \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(4,6,0) || \

JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0)

# define JSON\_HEDLEY\_CONST\_CAST(T, expr) (\_\_extension\_\_ ({ \

JSON\_HEDLEY\_DIAGNOSTIC\_PUSH \

JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_CAST\_QUAL \

((T) (expr)); \

JSON\_HEDLEY\_DIAGNOSTIC\_POP \

}))

#else

# define JSON\_HEDLEY\_CONST\_CAST(T, expr) ((T) (expr))

#endif

#if defined(JSON\_HEDLEY\_REINTERPRET\_CAST)

#undef JSON\_HEDLEY\_REINTERPRET\_CAST

#endif

#if defined(\_\_cplusplus)

#define JSON\_HEDLEY\_REINTERPRET\_CAST(T, expr) (reinterpret\_cast<T>(expr))

#else

#define JSON\_HEDLEY\_REINTERPRET\_CAST(T, expr) (\*((T\*) &(expr)))

#endif

#if defined(JSON\_HEDLEY\_STATIC\_CAST)

#undef JSON\_HEDLEY\_STATIC\_CAST

#endif

#if defined(\_\_cplusplus)

#define JSON\_HEDLEY\_STATIC\_CAST(T, expr) (static\_cast<T>(expr))

#else

#define JSON\_HEDLEY\_STATIC\_CAST(T, expr) ((T) (expr))

#endif

#if defined(JSON\_HEDLEY\_CPP\_CAST)

#undef JSON\_HEDLEY\_CPP\_CAST

#endif

#if defined(\_\_cplusplus)

#define JSON\_HEDLEY\_CPP\_CAST(T, expr) static\_cast<T>(expr)

#else

#define JSON\_HEDLEY\_CPP\_CAST(T, expr) (expr)

#endif

#if defined(JSON\_HEDLEY\_NULL)

#undef JSON\_HEDLEY\_NULL

#endif

#if defined(\_\_cplusplus)

#if \_\_cplusplus >= 201103L

#define JSON\_HEDLEY\_NULL JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_CPP98\_COMPAT\_WRAP\_(nullptr)

#elif defined(NULL)

#define JSON\_HEDLEY\_NULL NULL

#else

#define JSON\_HEDLEY\_NULL JSON\_HEDLEY\_STATIC\_CAST(void\*, 0)

#endif

#elif defined(NULL)

#define JSON\_HEDLEY\_NULL NULL

#else

#define JSON\_HEDLEY\_NULL ((void\*) 0)

#endif

#if defined(JSON\_HEDLEY\_MESSAGE)

#undef JSON\_HEDLEY\_MESSAGE

#endif

#if JSON\_HEDLEY\_HAS\_WARNING("-Wunknown-pragmas")

# define JSON\_HEDLEY\_MESSAGE(msg) \

JSON\_HEDLEY\_DIAGNOSTIC\_PUSH \

JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_UNKNOWN\_PRAGMAS \

JSON\_HEDLEY\_PRAGMA(message msg) \

JSON\_HEDLEY\_DIAGNOSTIC\_POP

#elif \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(4,4,0) || \

JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(13,0,0)

# define JSON\_HEDLEY\_MESSAGE(msg) JSON\_HEDLEY\_PRAGMA(message msg)

#elif JSON\_HEDLEY\_CRAY\_VERSION\_CHECK(5,0,0)

# define JSON\_HEDLEY\_MESSAGE(msg) JSON\_HEDLEY\_PRAGMA(\_CRI message msg)

#elif JSON\_HEDLEY\_IAR\_VERSION\_CHECK(8,0,0)

# define JSON\_HEDLEY\_MESSAGE(msg) JSON\_HEDLEY\_PRAGMA(message(msg))

#elif JSON\_HEDLEY\_PELLES\_VERSION\_CHECK(2,0,0)

# define JSON\_HEDLEY\_MESSAGE(msg) JSON\_HEDLEY\_PRAGMA(message(msg))

#else

# define JSON\_HEDLEY\_MESSAGE(msg)

#endif

#if defined(JSON\_HEDLEY\_WARNING)

#undef JSON\_HEDLEY\_WARNING

#endif

#if JSON\_HEDLEY\_HAS\_WARNING("-Wunknown-pragmas")

# define JSON\_HEDLEY\_WARNING(msg) \

JSON\_HEDLEY\_DIAGNOSTIC\_PUSH \

JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_UNKNOWN\_PRAGMAS \

JSON\_HEDLEY\_PRAGMA(clang warning msg) \

JSON\_HEDLEY\_DIAGNOSTIC\_POP

#elif \

JSON\_HEDLEY\_GCC\_VERSION\_CHECK(4,8,0) || \

JSON\_HEDLEY\_PGI\_VERSION\_CHECK(18,4,0)

# define JSON\_HEDLEY\_WARNING(msg) JSON\_HEDLEY\_PRAGMA(GCC warning msg)

#elif JSON\_HEDLEY\_MSVC\_VERSION\_CHECK(15,0,0)

# define JSON\_HEDLEY\_WARNING(msg) JSON\_HEDLEY\_PRAGMA(message(msg))

#else

# define JSON\_HEDLEY\_WARNING(msg) JSON\_HEDLEY\_MESSAGE(msg)

#endif

#if defined(JSON\_HEDLEY\_REQUIRE)

#undef JSON\_HEDLEY\_REQUIRE

#endif

#if defined(JSON\_HEDLEY\_REQUIRE\_MSG)

#undef JSON\_HEDLEY\_REQUIRE\_MSG

#endif

#if JSON\_HEDLEY\_HAS\_ATTRIBUTE(diagnose\_if)

# if JSON\_HEDLEY\_HAS\_WARNING("-Wgcc-compat")

# define JSON\_HEDLEY\_REQUIRE(expr) \

JSON\_HEDLEY\_DIAGNOSTIC\_PUSH \

\_Pragma("clang diagnostic ignored \"-Wgcc-compat\"") \

\_\_attribute\_\_((diagnose\_if(!(expr), #expr, "error"))) \

JSON\_HEDLEY\_DIAGNOSTIC\_POP

# define JSON\_HEDLEY\_REQUIRE\_MSG(expr,msg) \

JSON\_HEDLEY\_DIAGNOSTIC\_PUSH \

\_Pragma("clang diagnostic ignored \"-Wgcc-compat\"") \

\_\_attribute\_\_((diagnose\_if(!(expr), msg, "error"))) \

JSON\_HEDLEY\_DIAGNOSTIC\_POP

# else

# define JSON\_HEDLEY\_REQUIRE(expr) \_\_attribute\_\_((diagnose\_if(!(expr), #expr, "error")))

# define JSON\_HEDLEY\_REQUIRE\_MSG(expr,msg) \_\_attribute\_\_((diagnose\_if(!(expr), msg, "error")))

# endif

#else

# define JSON\_HEDLEY\_REQUIRE(expr)

# define JSON\_HEDLEY\_REQUIRE\_MSG(expr,msg)

#endif

#if defined(JSON\_HEDLEY\_FLAGS)

#undef JSON\_HEDLEY\_FLAGS

#endif

#if JSON\_HEDLEY\_HAS\_ATTRIBUTE(flag\_enum)

#define JSON\_HEDLEY\_FLAGS \_\_attribute\_\_((\_\_flag\_enum\_\_))

#endif

#if defined(JSON\_HEDLEY\_FLAGS\_CAST)

#undef JSON\_HEDLEY\_FLAGS\_CAST

#endif

#if JSON\_HEDLEY\_INTEL\_VERSION\_CHECK(19,0,0)

# define JSON\_HEDLEY\_FLAGS\_CAST(T, expr) (\_\_extension\_\_ ({ \

JSON\_HEDLEY\_DIAGNOSTIC\_PUSH \

\_Pragma("warning(disable:188)") \

((T) (expr)); \

JSON\_HEDLEY\_DIAGNOSTIC\_POP \

}))

#else

# define JSON\_HEDLEY\_FLAGS\_CAST(T, expr) JSON\_HEDLEY\_STATIC\_CAST(T, expr)

#endif

#if defined(JSON\_HEDLEY\_EMPTY\_BASES)

#undef JSON\_HEDLEY\_EMPTY\_BASES

#endif

#if JSON\_HEDLEY\_MSVC\_VERSION\_CHECK(19,0,23918) && !JSON\_HEDLEY\_MSVC\_VERSION\_CHECK(20,0,0)

#define JSON\_HEDLEY\_EMPTY\_BASES \_\_declspec(empty\_bases)

#else

#define JSON\_HEDLEY\_EMPTY\_BASES

#endif

/\* Remaining macros are deprecated. \*/

#if defined(JSON\_HEDLEY\_GCC\_NOT\_CLANG\_VERSION\_CHECK)

#undef JSON\_HEDLEY\_GCC\_NOT\_CLANG\_VERSION\_CHECK

#endif

#if defined(\_\_clang\_\_)

#define JSON\_HEDLEY\_GCC\_NOT\_CLANG\_VERSION\_CHECK(major,minor,patch) (0)

#else

#define JSON\_HEDLEY\_GCC\_NOT\_CLANG\_VERSION\_CHECK(major,minor,patch) JSON\_HEDLEY\_GCC\_VERSION\_CHECK(major,minor,patch)

#endif

#if defined(JSON\_HEDLEY\_CLANG\_HAS\_ATTRIBUTE)

#undef JSON\_HEDLEY\_CLANG\_HAS\_ATTRIBUTE

#endif

#define JSON\_HEDLEY\_CLANG\_HAS\_ATTRIBUTE(attribute) JSON\_HEDLEY\_HAS\_ATTRIBUTE(attribute)

#if defined(JSON\_HEDLEY\_CLANG\_HAS\_CPP\_ATTRIBUTE)

#undef JSON\_HEDLEY\_CLANG\_HAS\_CPP\_ATTRIBUTE

#endif

#define JSON\_HEDLEY\_CLANG\_HAS\_CPP\_ATTRIBUTE(attribute) JSON\_HEDLEY\_HAS\_CPP\_ATTRIBUTE(attribute)

#if defined(JSON\_HEDLEY\_CLANG\_HAS\_BUILTIN)

#undef JSON\_HEDLEY\_CLANG\_HAS\_BUILTIN

#endif

#define JSON\_HEDLEY\_CLANG\_HAS\_BUILTIN(builtin) JSON\_HEDLEY\_HAS\_BUILTIN(builtin)

#if defined(JSON\_HEDLEY\_CLANG\_HAS\_FEATURE)

#undef JSON\_HEDLEY\_CLANG\_HAS\_FEATURE

#endif

#define JSON\_HEDLEY\_CLANG\_HAS\_FEATURE(feature) JSON\_HEDLEY\_HAS\_FEATURE(feature)

#if defined(JSON\_HEDLEY\_CLANG\_HAS\_EXTENSION)

#undef JSON\_HEDLEY\_CLANG\_HAS\_EXTENSION

#endif

#define JSON\_HEDLEY\_CLANG\_HAS\_EXTENSION(extension) JSON\_HEDLEY\_HAS\_EXTENSION(extension)

#if defined(JSON\_HEDLEY\_CLANG\_HAS\_DECLSPEC\_DECLSPEC\_ATTRIBUTE)

#undef JSON\_HEDLEY\_CLANG\_HAS\_DECLSPEC\_DECLSPEC\_ATTRIBUTE

#endif

#define JSON\_HEDLEY\_CLANG\_HAS\_DECLSPEC\_ATTRIBUTE(attribute) JSON\_HEDLEY\_HAS\_DECLSPEC\_ATTRIBUTE(attribute)

#if defined(JSON\_HEDLEY\_CLANG\_HAS\_WARNING)

#undef JSON\_HEDLEY\_CLANG\_HAS\_WARNING

#endif

#define JSON\_HEDLEY\_CLANG\_HAS\_WARNING(warning) JSON\_HEDLEY\_HAS\_WARNING(warning)

#endif /\* !defined(JSON\_HEDLEY\_VERSION) || (JSON\_HEDLEY\_VERSION < X) \*/

// This file contains all internal macro definitions

// You MUST include macro\_unscope.hpp at the end of json.hpp to undef all of them

// exclude unsupported compilers

#if !defined(JSON\_SKIP\_UNSUPPORTED\_COMPILER\_CHECK)

#if defined(\_\_clang\_\_)

#if (\_\_clang\_major\_\_ \* 10000 + \_\_clang\_minor\_\_ \* 100 + \_\_clang\_patchlevel\_\_) < 30400

#error "unsupported Clang version - see https://github.com/nlohmann/json#supported-compilers"

#endif

#elif defined(\_\_GNUC\_\_) && !(defined(\_\_ICC) || defined(\_\_INTEL\_COMPILER))

#if (\_\_GNUC\_\_ \* 10000 + \_\_GNUC\_MINOR\_\_ \* 100 + \_\_GNUC\_PATCHLEVEL\_\_) < 40800

#error "unsupported GCC version - see https://github.com/nlohmann/json#supported-compilers"

#endif

#endif

#endif

// C++ language standard detection

#if (defined(\_\_cplusplus) && \_\_cplusplus >= 201703L) || (defined(\_HAS\_CXX17) && \_HAS\_CXX17 == 1) // fix for issue #464

#define JSON\_HAS\_CPP\_17

#define JSON\_HAS\_CPP\_14

#elif (defined(\_\_cplusplus) && \_\_cplusplus >= 201402L) || (defined(\_HAS\_CXX14) && \_HAS\_CXX14 == 1)

#define JSON\_HAS\_CPP\_14

#endif

// disable float-equal warnings on GCC/clang

#if defined(\_\_clang\_\_) || defined(\_\_GNUC\_\_) || defined(\_\_GNUG\_\_)

#pragma GCC diagnostic push

#pragma GCC diagnostic ignored "-Wfloat-equal"

#endif

// disable documentation warnings on clang

#if defined(\_\_clang\_\_)

#pragma GCC diagnostic push

#pragma GCC diagnostic ignored "-Wdocumentation"

#endif

// allow to disable exceptions

#if (defined(\_\_cpp\_exceptions) || defined(\_\_EXCEPTIONS) || defined(\_CPPUNWIND)) && !defined(JSON\_NOEXCEPTION)

#define JSON\_THROW(exception) throw exception

#define JSON\_TRY try

#define JSON\_CATCH(exception) catch(exception)

#define JSON\_INTERNAL\_CATCH(exception) catch(exception)

#else

#include <cstdlib>

#define JSON\_THROW(exception) std::abort()

#define JSON\_TRY if(true)

#define JSON\_CATCH(exception) if(false)

#define JSON\_INTERNAL\_CATCH(exception) if(false)

#endif

// override exception macros

#if defined(JSON\_THROW\_USER)

#undef JSON\_THROW

#define JSON\_THROW JSON\_THROW\_USER

#endif

#if defined(JSON\_TRY\_USER)

#undef JSON\_TRY

#define JSON\_TRY JSON\_TRY\_USER

#endif

#if defined(JSON\_CATCH\_USER)

#undef JSON\_CATCH

#define JSON\_CATCH JSON\_CATCH\_USER

#undef JSON\_INTERNAL\_CATCH

#define JSON\_INTERNAL\_CATCH JSON\_CATCH\_USER

#endif

#if defined(JSON\_INTERNAL\_CATCH\_USER)

#undef JSON\_INTERNAL\_CATCH

#define JSON\_INTERNAL\_CATCH JSON\_INTERNAL\_CATCH\_USER

#endif

/\*!

@brief macro to briefly define a mapping between an enum and JSON

@def NLOHMANN\_JSON\_SERIALIZE\_ENUM

@since version 3.4.0

\*/

#define NLOHMANN\_JSON\_SERIALIZE\_ENUM(ENUM\_TYPE, ...) \

template<typename BasicJsonType> \

inline void to\_json(BasicJsonType& j, const ENUM\_TYPE& e) \

{ \

static\_assert(std::is\_enum<ENUM\_TYPE>::value, #ENUM\_TYPE " must be an enum!"); \

static const std::pair<ENUM\_TYPE, BasicJsonType> m[] = \_\_VA\_ARGS\_\_; \

auto it = std::find\_if(std::begin(m), std::end(m), \

[e](const std::pair<ENUM\_TYPE, BasicJsonType>& ej\_pair) -> bool \

{ \

return ej\_pair.first == e; \

}); \

j = ((it != std::end(m)) ? it : std::begin(m))->second; \

} \

template<typename BasicJsonType> \

inline void from\_json(const BasicJsonType& j, ENUM\_TYPE& e) \

{ \

static\_assert(std::is\_enum<ENUM\_TYPE>::value, #ENUM\_TYPE " must be an enum!"); \

static const std::pair<ENUM\_TYPE, BasicJsonType> m[] = \_\_VA\_ARGS\_\_; \

auto it = std::find\_if(std::begin(m), std::end(m), \

[&j](const std::pair<ENUM\_TYPE, BasicJsonType>& ej\_pair) -> bool \

{ \

return ej\_pair.second == j; \

}); \

e = ((it != std::end(m)) ? it : std::begin(m))->first; \

}

// Ugly macros to avoid uglier copy-paste when specializing basic\_json. They

// may be removed in the future once the class is split.

#define NLOHMANN\_BASIC\_JSON\_TPL\_DECLARATION \

template<template<typename, typename, typename...> class ObjectType, \

template<typename, typename...> class ArrayType, \

class StringType, class BooleanType, class NumberIntegerType, \

class NumberUnsignedType, class NumberFloatType, \

template<typename> class AllocatorType, \

template<typename, typename = void> class JSONSerializer>

#define NLOHMANN\_BASIC\_JSON\_TPL \

basic\_json<ObjectType, ArrayType, StringType, BooleanType, \

NumberIntegerType, NumberUnsignedType, NumberFloatType, \

AllocatorType, JSONSerializer>

namespace nlohmann

{

namespace detail

{

////////////////

// exceptions //

////////////////

/\*!

@brief general exception of the @ref basic\_json class

This class is an extension of `std::exception` objects with a member @a id for

exception ids. It is used as the base class for all exceptions thrown by the

@ref basic\_json class. This class can hence be used as "wildcard" to catch

exceptions.

Subclasses:

- @ref parse\_error for exceptions indicating a parse error

- @ref invalid\_iterator for exceptions indicating errors with iterators

- @ref type\_error for exceptions indicating executing a member function with

a wrong type

- @ref out\_of\_range for exceptions indicating access out of the defined range

- @ref other\_error for exceptions indicating other library errors

@internal

@note To have nothrow-copy-constructible exceptions, we internally use

`std::runtime\_error` which can cope with arbitrary-length error messages.

Intermediate strings are built with static functions and then passed to

the actual constructor.

@endinternal

@liveexample{The following code shows how arbitrary library exceptions can be

caught.,exception}

@since version 3.0.0

\*/

class exception : public std::exception

{

public:

/// returns the explanatory string

JSON\_HEDLEY\_RETURNS\_NON\_NULL

const char\* what() const noexcept override

{

return m.what();

}

/// the id of the exception

const int id;

protected:

JSON\_HEDLEY\_NON\_NULL(3)

exception(int id\_, const char\* what\_arg) : id(id\_), m(what\_arg) {}

static std::string name(const std::string& ename, int id\_)

{

return "[json.exception." + ename + "." + std::to\_string(id\_) + "] ";

}

private:

/// an exception object as storage for error messages

std::runtime\_error m;

};

/\*!

@brief exception indicating a parse error

This exception is thrown by the library when a parse error occurs. Parse errors

can occur during the deserialization of JSON text, CBOR, MessagePack, as well

as when using JSON Patch.

Member @a byte holds the byte index of the last read character in the input

file.

Exceptions have ids 1xx.

name / id | example message | description

------------------------------ | --------------- | -------------------------

json.exception.parse\_error.101 | parse error at 2: unexpected end of input; expected string literal | This error indicates a syntax error while deserializing a JSON text. The error message describes that an unexpected token (character) was encountered, and the member @a byte indicates the error position.

json.exception.parse\_error.102 | parse error at 14: missing or wrong low surrogate | JSON uses the `\uxxxx` format to describe Unicode characters. Code points above above 0xFFFF are split into two `\uxxxx` entries ("surrogate pairs"). This error indicates that the surrogate pair is incomplete or contains an invalid code point.

json.exception.parse\_error.103 | parse error: code points above 0x10FFFF are invalid | Unicode supports code points up to 0x10FFFF. Code points above 0x10FFFF are invalid.

json.exception.parse\_error.104 | parse error: JSON patch must be an array of objects | [RFC 6902](https://tools.ietf.org/html/rfc6902) requires a JSON Patch document to be a JSON document that represents an array of objects.

json.exception.parse\_error.105 | parse error: operation must have string member 'op' | An operation of a JSON Patch document must contain exactly one "op" member, whose value indicates the operation to perform. Its value must be one of "add", "remove", "replace", "move", "copy", or "test"; other values are errors.

json.exception.parse\_error.106 | parse error: array index '01' must not begin with '0' | An array index in a JSON Pointer ([RFC 6901](https://tools.ietf.org/html/rfc6901)) may be `0` or any number without a leading `0`.

json.exception.parse\_error.107 | parse error: JSON pointer must be empty or begin with '/' - was: 'foo' | A JSON Pointer must be a Unicode string containing a sequence of zero or more reference tokens, each prefixed by a `/` character.

json.exception.parse\_error.108 | parse error: escape character '~' must be followed with '0' or '1' | In a JSON Pointer, only `~0` and `~1` are valid escape sequences.

json.exception.parse\_error.109 | parse error: array index 'one' is not a number | A JSON Pointer array index must be a number.

json.exception.parse\_error.110 | parse error at 1: cannot read 2 bytes from vector | When parsing CBOR or MessagePack, the byte vector ends before the complete value has been read.

json.exception.parse\_error.112 | parse error at 1: error reading CBOR; last byte: 0xF8 | Not all types of CBOR or MessagePack are supported. This exception occurs if an unsupported byte was read.

json.exception.parse\_error.113 | parse error at 2: expected a CBOR string; last byte: 0x98 | While parsing a map key, a value that is not a string has been read.

json.exception.parse\_error.114 | parse error: Unsupported BSON record type 0x0F | The parsing of the corresponding BSON record type is not implemented (yet).

@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).

@liveexample{The following code shows how a `parse\_error` exception can be

caught.,parse\_error}

@sa - @ref exception for the base class of the library exceptions

@sa - @ref invalid\_iterator for exceptions indicating errors with iterators

@sa - @ref type\_error for exceptions indicating executing a member function with

a wrong type

@sa - @ref out\_of\_range for exceptions indicating access out of the defined range

@sa - @ref other\_error for exceptions indicating other library errors

@since version 3.0.0

\*/

class parse\_error : public exception

{

public:

/\*!

@brief create a parse error exception

@param[in] id\_ the id of the exception

@param[in] pos the position where the error occurred (or with

chars\_read\_total=0 if the position cannot be

determined)

@param[in] what\_arg the explanatory string

@return parse\_error object

\*/

static parse\_error create(int id\_, const position\_t& pos, const std::string& what\_arg)

{

std::string w = exception::name("parse\_error", id\_) + "parse error" +

position\_string(pos) + ": " + what\_arg;

return parse\_error(id\_, pos.chars\_read\_total, w.c\_str());

}

static parse\_error create(int id\_, std::size\_t byte\_, const std::string& what\_arg)

{

std::string w = exception::name("parse\_error", id\_) + "parse error" +

(byte\_ != 0 ? (" at byte " + std::to\_string(byte\_)) : "") +

": " + what\_arg;

return parse\_error(id\_, byte\_, w.c\_str());

}

/\*!

@brief 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).

\*/

const std::size\_t byte;

private:

parse\_error(int id\_, std::size\_t byte\_, const char\* what\_arg)

: exception(id\_, what\_arg), byte(byte\_) {}

static std::string position\_string(const position\_t& pos)

{

return " at line " + std::to\_string(pos.lines\_read + 1) +

", column " + std::to\_string(pos.chars\_read\_current\_line);

}

};

/\*!

@brief exception indicating errors with iterators

This exception is thrown if iterators passed to a library function do not match

the expected semantics.

Exceptions have ids 2xx.

name / id | example message | description

----------------------------------- | --------------- | -------------------------

json.exception.invalid\_iterator.201 | iterators are not compatible | The iterators passed to constructor @ref basic\_json(InputIT first, InputIT last) are not compatible, meaning they do not belong to the same container. Therefore, the range (@a first, @a last) is invalid.

json.exception.invalid\_iterator.202 | iterator does not fit current value | In an erase or insert function, the passed iterator @a pos does not belong to the JSON value for which the function was called. It hence does not define a valid position for the deletion/insertion.

json.exception.invalid\_iterator.203 | iterators do not fit current value | Either iterator passed to function @ref erase(IteratorType first, IteratorType last) does not belong to the JSON value from which values shall be erased. It hence does not define a valid range to delete values from.

json.exception.invalid\_iterator.204 | iterators out of range | When an iterator range for a primitive type (number, boolean, or string) is passed to a constructor or an erase function, this range has to be exactly (@ref begin(), @ref end()), because this is the only way the single stored value is expressed. All other ranges are invalid.

json.exception.invalid\_iterator.205 | iterator out of range | When an iterator for a primitive type (number, boolean, or string) is passed to an erase function, the iterator has to be the @ref begin() iterator, because it is the only way to address the stored value. All other iterators are invalid.

json.exception.invalid\_iterator.206 | cannot construct with iterators from null | The iterators passed to constructor @ref basic\_json(InputIT first, InputIT last) belong to a JSON null value and hence to not define a valid range.

json.exception.invalid\_iterator.207 | cannot use key() for non-object iterators | The key() member function can only be used on iterators belonging to a JSON object, because other types do not have a concept of a key.

json.exception.invalid\_iterator.208 | cannot use operator[] for object iterators | The operator[] to specify a concrete offset cannot be used on iterators belonging to a JSON object, because JSON objects are unordered.

json.exception.invalid\_iterator.209 | cannot use offsets with object iterators | The offset operators (+, -, +=, -=) cannot be used on iterators belonging to a JSON object, because JSON objects are unordered.

json.exception.invalid\_iterator.210 | iterators do not fit | The iterator range passed to the insert function are not compatible, meaning they do not belong to the same container. Therefore, the range (@a first, @a last) is invalid.

json.exception.invalid\_iterator.211 | passed iterators may not belong to container | The iterator range passed to the insert function must not be a subrange of the container to insert to.

json.exception.invalid\_iterator.212 | cannot compare iterators of different containers | When two iterators are compared, they must belong to the same container.

json.exception.invalid\_iterator.213 | cannot compare order of object iterators | The order of object iterators cannot be compared, because JSON objects are unordered.

json.exception.invalid\_iterator.214 | cannot get value | Cannot get value for iterator: Either the iterator belongs to a null value or it is an iterator to a primitive type (number, boolean, or string), but the iterator is different to @ref begin().

@liveexample{The following code shows how an `invalid\_iterator` exception can be

caught.,invalid\_iterator}

@sa - @ref exception for the base class of the library exceptions

@sa - @ref parse\_error for exceptions indicating a parse error

@sa - @ref type\_error for exceptions indicating executing a member function with

a wrong type

@sa - @ref out\_of\_range for exceptions indicating access out of the defined range

@sa - @ref other\_error for exceptions indicating other library errors

@since version 3.0.0

\*/

class invalid\_iterator : public exception

{

public:

static invalid\_iterator create(int id\_, const std::string& what\_arg)

{

std::string w = exception::name("invalid\_iterator", id\_) + what\_arg;

return invalid\_iterator(id\_, w.c\_str());

}

private:

JSON\_HEDLEY\_NON\_NULL(3)

invalid\_iterator(int id\_, const char\* what\_arg)

: exception(id\_, what\_arg) {}

};

/\*!

@brief exception indicating executing a member function with a wrong type

This exception is thrown in case of a type error; that is, a library function is

executed on a JSON value whose type does not match the expected semantics.

Exceptions have ids 3xx.

name / id | example message | description

----------------------------- | --------------- | -------------------------

json.exception.type\_error.301 | cannot create object from initializer list | To create an object from an initializer list, the initializer list must consist only of a list of pairs whose first element is a string. When this constraint is violated, an array is created instead.

json.exception.type\_error.302 | type must be object, but is array | During implicit or explicit value conversion, the JSON type must be compatible to the target type. For instance, a JSON string can only be converted into string types, but not into numbers or boolean types.

json.exception.type\_error.303 | incompatible ReferenceType for get\_ref, actual type is object | To retrieve a reference to a value stored in a @ref basic\_json object with @ref get\_ref, the type of the reference must match the value type. For instance, for a JSON array, the @a ReferenceType must be @ref array\_t &.

json.exception.type\_error.304 | cannot use at() with string | The @ref at() member functions can only be executed for certain JSON types.

json.exception.type\_error.305 | cannot use operator[] with string | The @ref operator[] member functions can only be executed for certain JSON types.

json.exception.type\_error.306 | cannot use value() with string | The @ref value() member functions can only be executed for certain JSON types.

json.exception.type\_error.307 | cannot use erase() with string | The @ref erase() member functions can only be executed for certain JSON types.

json.exception.type\_error.308 | cannot use push\_back() with string | The @ref push\_back() and @ref operator+= member functions can only be executed for certain JSON types.

json.exception.type\_error.309 | cannot use insert() with | The @ref insert() member functions can only be executed for certain JSON types.

json.exception.type\_error.310 | cannot use swap() with number | The @ref swap() member functions can only be executed for certain JSON types.

json.exception.type\_error.311 | cannot use emplace\_back() with string | The @ref emplace\_back() member function can only be executed for certain JSON types.

json.exception.type\_error.312 | cannot use update() with string | The @ref update() member functions can only be executed for certain JSON types.

json.exception.type\_error.313 | invalid value to unflatten | The @ref unflatten function converts an object whose keys are JSON Pointers back into an arbitrary nested JSON value. The JSON Pointers must not overlap, because then the resulting value would not be well defined.

json.exception.type\_error.314 | only objects can be unflattened | The @ref unflatten function only works for an object whose keys are JSON Pointers.

json.exception.type\_error.315 | values in object must be primitive | The @ref unflatten function only works for an object whose keys are JSON Pointers and whose values are primitive.

json.exception.type\_error.316 | invalid UTF-8 byte at index 10: 0x7E | The @ref dump function only works with UTF-8 encoded strings; that is, if you assign a `std::string` to a JSON value, make sure it is UTF-8 encoded. |

json.exception.type\_error.317 | JSON value cannot be serialized to requested format | The dynamic type of the object cannot be represented in the requested serialization format (e.g. a raw `true` or `null` JSON object cannot be serialized to BSON) |

@liveexample{The following code shows how a `type\_error` exception can be

caught.,type\_error}

@sa - @ref exception for the base class of the library exceptions

@sa - @ref parse\_error for exceptions indicating a parse error

@sa - @ref invalid\_iterator for exceptions indicating errors with iterators

@sa - @ref out\_of\_range for exceptions indicating access out of the defined range

@sa - @ref other\_error for exceptions indicating other library errors

@since version 3.0.0

\*/

class type\_error : public exception

{

public:

static type\_error create(int id\_, const std::string& what\_arg)

{

std::string w = exception::name("type\_error", id\_) + what\_arg;

return type\_error(id\_, w.c\_str());

}

private:

JSON\_HEDLEY\_NON\_NULL(3)

type\_error(int id\_, const char\* what\_arg) : exception(id\_, what\_arg) {}

};

/\*!

@brief exception indicating access out of the defined range

This exception is thrown in case a library function is called on an input

parameter that exceeds the expected range, for instance in case of array

indices or nonexisting object keys.

Exceptions have ids 4xx.

name / id | example message | description

------------------------------- | --------------- | -------------------------

json.exception.out\_of\_range.401 | array index 3 is out of range | The provided array index @a i is larger than @a size-1.

json.exception.out\_of\_range.402 | array index '-' (3) is out of range | The special array index `-` in a JSON Pointer never describes a valid element of the array, but the index past the end. That is, it can only be used to add elements at this position, but not to read it.

json.exception.out\_of\_range.403 | key 'foo' not found | The provided key was not found in the JSON object.

json.exception.out\_of\_range.404 | unresolved reference token 'foo' | A reference token in a JSON Pointer could not be resolved.

json.exception.out\_of\_range.405 | JSON pointer has no parent | The JSON Patch operations 'remove' and 'add' can not be applied to the root element of the JSON value.

json.exception.out\_of\_range.406 | number overflow parsing '10E1000' | A parsed number could not be stored as without changing it to NaN or INF.

json.exception.out\_of\_range.407 | number overflow serializing '9223372036854775808' | UBJSON and BSON only support integer numbers up to 9223372036854775807. |

json.exception.out\_of\_range.408 | excessive array size: 8658170730974374167 | The size (following `#`) of an UBJSON array or object exceeds the maximal capacity. |

json.exception.out\_of\_range.409 | BSON key cannot contain code point U+0000 (at byte 2) | Key identifiers to be serialized to BSON cannot contain code point U+0000, since the key is stored as zero-terminated c-string |

@liveexample{The following code shows how an `out\_of\_range` exception can be

caught.,out\_of\_range}

@sa - @ref exception for the base class of the library exceptions

@sa - @ref parse\_error for exceptions indicating a parse error

@sa - @ref invalid\_iterator for exceptions indicating errors with iterators

@sa - @ref type\_error for exceptions indicating executing a member function with

a wrong type

@sa - @ref other\_error for exceptions indicating other library errors

@since version 3.0.0

\*/

class out\_of\_range : public exception

{

public:

static out\_of\_range create(int id\_, const std::string& what\_arg)

{

std::string w = exception::name("out\_of\_range", id\_) + what\_arg;

return out\_of\_range(id\_, w.c\_str());

}

private:

JSON\_HEDLEY\_NON\_NULL(3)

out\_of\_range(int id\_, const char\* what\_arg) : exception(id\_, what\_arg) {}

};

/\*!

@brief exception indicating other library errors

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

other exception types.

Exceptions have ids 5xx.

name / id | example message | description

------------------------------ | --------------- | -------------------------

json.exception.other\_error.501 | unsuccessful: {"op":"test","path":"/baz", "value":"bar"} | A JSON Patch operation 'test' failed. The unsuccessful operation is also printed.

@sa - @ref exception for the base class of the library exceptions

@sa - @ref parse\_error for exceptions indicating a parse error

@sa - @ref invalid\_iterator for exceptions indicating errors with iterators

@sa - @ref type\_error for exceptions indicating executing a member function with

a wrong type

@sa - @ref out\_of\_range for exceptions indicating access out of the defined range

@liveexample{The following code shows how an `other\_error` exception can be

caught.,other\_error}

@since version 3.0.0

\*/

class other\_error : public exception

{

public:

static other\_error create(int id\_, const std::string& what\_arg)

{

std::string w = exception::name("other\_error", id\_) + what\_arg;

return other\_error(id\_, w.c\_str());

}

private:

JSON\_HEDLEY\_NON\_NULL(3)

other\_error(int id\_, const char\* what\_arg) : exception(id\_, what\_arg) {}

};

} // namespace detail

} // namespace nlohmann

// #include <nlohmann/detail/macro\_scope.hpp>

// #include <nlohmann/detail/meta/cpp\_future.hpp>

#include <ciso646> // not

#include <cstddef> // size\_t

#include <type\_traits> // conditional, enable\_if, false\_type, integral\_constant, is\_constructible, is\_integral, is\_same, remove\_cv, remove\_reference, true\_type

namespace nlohmann

{

namespace detail

{

// alias templates to reduce boilerplate

template<bool B, typename T = void>

using enable\_if\_t = typename std::enable\_if<B, T>::type;

template<typename T>

using uncvref\_t = typename std::remove\_cv<typename std::remove\_reference<T>::type>::type;

// implementation of C++14 index\_sequence and affiliates

// source: https://stackoverflow.com/a/32223343

template<std::size\_t... Ints>

struct index\_sequence

{

using type = index\_sequence;

using value\_type = std::size\_t;

static constexpr std::size\_t size() noexcept

{

return sizeof...(Ints);

}

};

template<class Sequence1, class Sequence2>

struct merge\_and\_renumber;

template<std::size\_t... I1, std::size\_t... I2>

struct merge\_and\_renumber<index\_sequence<I1...>, index\_sequence<I2...>>

: index\_sequence < I1..., (sizeof...(I1) + I2)... > {};

template<std::size\_t N>

struct make\_index\_sequence

: merge\_and\_renumber < typename make\_index\_sequence < N / 2 >::type,

typename make\_index\_sequence < N - N / 2 >::type > {};

template<> struct make\_index\_sequence<0> : index\_sequence<> {};

template<> struct make\_index\_sequence<1> : index\_sequence<0> {};

template<typename... Ts>

using index\_sequence\_for = make\_index\_sequence<sizeof...(Ts)>;

// dispatch utility (taken from ranges-v3)

template<unsigned N> struct priority\_tag : priority\_tag < N - 1 > {};

template<> struct priority\_tag<0> {};

// taken from ranges-v3

template<typename T>

struct static\_const

{

static constexpr T value{};

};

template<typename T>

constexpr T static\_const<T>::value;

} // namespace detail

} // namespace nlohmann

// #include <nlohmann/detail/meta/type\_traits.hpp>

#include <ciso646> // not

#include <limits> // numeric\_limits

#include <type\_traits> // false\_type, is\_constructible, is\_integral, is\_same, true\_type

#include <utility> // declval

// #include <nlohmann/detail/iterators/iterator\_traits.hpp>

#include <iterator> // random\_access\_iterator\_tag

// #include <nlohmann/detail/meta/void\_t.hpp>

namespace nlohmann

{

namespace detail

{

template <typename ...Ts> struct make\_void

{

using type = void;

};

template <typename ...Ts> using void\_t = typename make\_void<Ts...>::type;

} // namespace detail

} // namespace nlohmann

// #include <nlohmann/detail/meta/cpp\_future.hpp>

namespace nlohmann

{

namespace detail

{

template <typename It, typename = void>

struct iterator\_types {};

template <typename It>

struct iterator\_types <

It,

void\_t<typename It::difference\_type, typename It::value\_type, typename It::pointer,

typename It::reference, typename It::iterator\_category >>

{

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;

};

// This is required as some compilers implement std::iterator\_traits in a way that

// doesn't work with SFINAE. See https://github.com/nlohmann/json/issues/1341.

template <typename T, typename = void>

struct iterator\_traits

{

};

template <typename T>

struct iterator\_traits < T, enable\_if\_t < !std::is\_pointer<T>::value >>

: iterator\_types<T>

{

};

template <typename T>

struct iterator\_traits<T\*, enable\_if\_t<std::is\_object<T>::value>>

{

using iterator\_category = std::random\_access\_iterator\_tag;

using value\_type = T;

using difference\_type = ptrdiff\_t;

using pointer = T\*;

using reference = T&;

};

} // namespace detail

} // namespace nlohmann

// #include <nlohmann/detail/macro\_scope.hpp>

// #include <nlohmann/detail/meta/cpp\_future.hpp>

// #include <nlohmann/detail/meta/detected.hpp>

#include <type\_traits>

// #include <nlohmann/detail/meta/void\_t.hpp>

// http://en.cppreference.com/w/cpp/experimental/is\_detected

namespace nlohmann

{

namespace detail

{

struct nonesuch

{

nonesuch() = delete;

~nonesuch() = delete;

nonesuch(nonesuch const&) = delete;

nonesuch(nonesuch const&&) = delete;

void operator=(nonesuch const&) = delete;

void operator=(nonesuch&&) = delete;

};

template <class Default,

class AlwaysVoid,

template <class...> class Op,

class... Args>

struct detector

{

using value\_t = std::false\_type;

using type = Default;

};

template <class Default, template <class...> class Op, class... Args>

struct detector<Default, void\_t<Op<Args...>>, Op, Args...>

{

using value\_t = std::true\_type;

using type = Op<Args...>;

};

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 =

std::is\_convertible<detected\_t<Op, Args...>, To>;

} // namespace detail

} // namespace nlohmann

// #include <nlohmann/json\_fwd.hpp>

#ifndef INCLUDE\_NLOHMANN\_JSON\_FWD\_HPP\_

#define INCLUDE\_NLOHMANN\_JSON\_FWD\_HPP\_

#include <cstdint> // int64\_t, uint64\_t

#include <map> // map

#include <memory> // allocator

#include <string> // string

#include <vector> // vector

/\*!

@brief namespace for Niels Lohmann

@see https://github.com/nlohmann

@since version 1.0.0

\*/

namespace nlohmann

{

/\*!

@brief default JSONSerializer template argument

This serializer ignores the template arguments and uses ADL

([argument-dependent lookup](https://en.cppreference.com/w/cpp/language/adl))

for serialization.

\*/

template<typename T = void, typename SFINAE = void>

struct adl\_serializer;

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 basic\_json;

/\*!

@brief JSON Pointer

A JSON pointer defines a string syntax for identifying a specific value

within a JSON document. It can be used with functions `at` and

`operator[]`. Furthermore, JSON pointers are the base for JSON patches.

@sa [RFC 6901](https://tools.ietf.org/html/rfc6901)

@since version 2.0.0

\*/

template<typename BasicJsonType>

class json\_pointer;

/\*!

@brief default JSON class

This type is the default specialization of the @ref basic\_json class which

uses the standard template types.

@since version 1.0.0

\*/

using json = basic\_json<>;

} // namespace nlohmann

#endif // INCLUDE\_NLOHMANN\_JSON\_FWD\_HPP\_

namespace nlohmann

{

/\*!

@brief detail namespace with internal helper functions

This namespace collects functions that should not be exposed,

implementations of some @ref basic\_json methods, and meta-programming helpers.

@since version 2.1.0

\*/

namespace detail

{

/////////////

// helpers //

/////////////

// Note to maintainers:

//

// Every trait in this file expects a non CV-qualified type.

// The only exceptions are in the 'aliases for detected' section

// (i.e. those of the form: decltype(T::member\_function(std::declval<T>())))

//

// In this case, T has to be properly CV-qualified to constraint the function arguments

// (e.g. to\_json(BasicJsonType&, const T&))

template<typename> struct is\_basic\_json : std::false\_type {};

NLOHMANN\_BASIC\_JSON\_TPL\_DECLARATION

struct is\_basic\_json<NLOHMANN\_BASIC\_JSON\_TPL> : std::true\_type {};

//////////////////////////

// aliases for detected //

//////////////////////////

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>

using iterator\_t = typename T::iterator;

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>());

// trait checking if JSONSerializer<T>::from\_json(json const&, udt&) exists

template <typename BasicJsonType, typename T, typename = void>

struct has\_from\_json : std::false\_type {};

template <typename BasicJsonType, typename T>

struct has\_from\_json<BasicJsonType, T,

enable\_if\_t<not is\_basic\_json<T>::value>>

{

using serializer = typename BasicJsonType::template json\_serializer<T, void>;

static constexpr bool value =

is\_detected\_exact<void, from\_json\_function, serializer,

const BasicJsonType&, T&>::value;

};

// This trait checks if JSONSerializer<T>::from\_json(json const&) exists

// this overload is used for non-default-constructible user-defined-types

template <typename BasicJsonType, typename T, typename = void>

struct has\_non\_default\_from\_json : std::false\_type {};

template<typename BasicJsonType, typename T>

struct has\_non\_default\_from\_json<BasicJsonType, T, enable\_if\_t<not is\_basic\_json<T>::value>>

{

using serializer = typename BasicJsonType::template json\_serializer<T, void>;

static constexpr bool value =

is\_detected\_exact<T, from\_json\_function, serializer,

const BasicJsonType&>::value;

};

// This trait checks if BasicJsonType::json\_serializer<T>::to\_json exists

// Do not evaluate the trait when T is a basic\_json type, to avoid template instantiation infinite recursion.

template <typename BasicJsonType, typename T, typename = void>

struct has\_to\_json : std::false\_type {};

template <typename BasicJsonType, typename T>

struct has\_to\_json<BasicJsonType, T, enable\_if\_t<not is\_basic\_json<T>::value>>

{

using serializer = typename BasicJsonType::template json\_serializer<T, void>;

static constexpr bool value =

is\_detected\_exact<void, to\_json\_function, serializer, BasicJsonType&,

T>::value;

};

///////////////////

// is\_ functions //

///////////////////

template <typename T, typename = void>

struct is\_iterator\_traits : std::false\_type {};

template <typename T>

struct is\_iterator\_traits<iterator\_traits<T>>

{

private:

using traits = iterator\_traits<T>;

public:

static constexpr auto 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;

};

// source: https://stackoverflow.com/a/37193089/4116453

template <typename T, typename = void>

struct is\_complete\_type : std::false\_type {};

template <typename T>

struct is\_complete\_type<T, decltype(void(sizeof(T)))> : std::true\_type {};

template <typename BasicJsonType, typename CompatibleObjectType,

typename = void>

struct is\_compatible\_object\_type\_impl : std::false\_type {};

template <typename BasicJsonType, typename CompatibleObjectType>

struct is\_compatible\_object\_type\_impl <

BasicJsonType, CompatibleObjectType,

enable\_if\_t<is\_detected<mapped\_type\_t, CompatibleObjectType>::value and

is\_detected<key\_type\_t, CompatibleObjectType>::value >>

{

using object\_t = typename BasicJsonType::object\_t;

// macOS's is\_constructible does not play well with nonesuch...

static constexpr bool value =

std::is\_constructible<typename object\_t::key\_type,

typename CompatibleObjectType::key\_type>::value and

std::is\_constructible<typename object\_t::mapped\_type,

typename CompatibleObjectType::mapped\_type>::value;

};

template <typename BasicJsonType, typename CompatibleObjectType>

struct is\_compatible\_object\_type

: is\_compatible\_object\_type\_impl<BasicJsonType, CompatibleObjectType> {};

template <typename BasicJsonType, typename ConstructibleObjectType,

typename = void>

struct is\_constructible\_object\_type\_impl : std::false\_type {};

template <typename BasicJsonType, typename ConstructibleObjectType>

struct is\_constructible\_object\_type\_impl <

BasicJsonType, ConstructibleObjectType,

enable\_if\_t<is\_detected<mapped\_type\_t, ConstructibleObjectType>::value and

is\_detected<key\_type\_t, ConstructibleObjectType>::value >>

{

using object\_t = typename BasicJsonType::object\_t;

static constexpr bool value =

(std::is\_default\_constructible<ConstructibleObjectType>::value and

(std::is\_move\_assignable<ConstructibleObjectType>::value or

std::is\_copy\_assignable<ConstructibleObjectType>::value) and

(std::is\_constructible<typename ConstructibleObjectType::key\_type,

typename object\_t::key\_type>::value and

std::is\_same <

typename object\_t::mapped\_type,

typename ConstructibleObjectType::mapped\_type >::value)) or

(has\_from\_json<BasicJsonType,

typename ConstructibleObjectType::mapped\_type>::value or

has\_non\_default\_from\_json <

BasicJsonType,

typename ConstructibleObjectType::mapped\_type >::value);

};

template <typename BasicJsonType, typename ConstructibleObjectType>

struct is\_constructible\_object\_type

: is\_constructible\_object\_type\_impl<BasicJsonType,

ConstructibleObjectType> {};

template <typename BasicJsonType, typename CompatibleStringType,

typename = void>

struct is\_compatible\_string\_type\_impl : std::false\_type {};

template <typename BasicJsonType, typename CompatibleStringType>

struct is\_compatible\_string\_type\_impl <

BasicJsonType, CompatibleStringType,

enable\_if\_t<is\_detected\_exact<typename BasicJsonType::string\_t::value\_type,

value\_type\_t, CompatibleStringType>::value >>

{

static constexpr auto value =

std::is\_constructible<typename BasicJsonType::string\_t, CompatibleStringType>::value;

};

template <typename BasicJsonType, typename ConstructibleStringType>

struct is\_compatible\_string\_type

: is\_compatible\_string\_type\_impl<BasicJsonType, ConstructibleStringType> {};

template <typename BasicJsonType, typename ConstructibleStringType,

typename = void>

struct is\_constructible\_string\_type\_impl : std::false\_type {};

template <typename BasicJsonType, typename ConstructibleStringType>

struct is\_constructible\_string\_type\_impl <

BasicJsonType, ConstructibleStringType,

enable\_if\_t<is\_detected\_exact<typename BasicJsonType::string\_t::value\_type,

value\_type\_t, ConstructibleStringType>::value >>

{

static constexpr auto value =

std::is\_constructible<ConstructibleStringType,

typename BasicJsonType::string\_t>::value;

};

template <typename BasicJsonType, typename ConstructibleStringType>

struct is\_constructible\_string\_type

: is\_constructible\_string\_type\_impl<BasicJsonType, ConstructibleStringType> {};

template <typename BasicJsonType, typename CompatibleArrayType, typename = void>

struct is\_compatible\_array\_type\_impl : std::false\_type {};

template <typename BasicJsonType, typename CompatibleArrayType>

struct is\_compatible\_array\_type\_impl <

BasicJsonType, CompatibleArrayType,

enable\_if\_t<is\_detected<value\_type\_t, CompatibleArrayType>::value and

is\_detected<iterator\_t, CompatibleArrayType>::value and

// This is needed because json\_reverse\_iterator has a ::iterator type...

// Therefore it is detected as a CompatibleArrayType.

// The real fix would be to have an Iterable concept.

not is\_iterator\_traits<

iterator\_traits<CompatibleArrayType>>::value >>

{

static constexpr bool value =

std::is\_constructible<BasicJsonType,

typename CompatibleArrayType::value\_type>::value;

};

template <typename BasicJsonType, typename CompatibleArrayType>

struct is\_compatible\_array\_type

: is\_compatible\_array\_type\_impl<BasicJsonType, CompatibleArrayType> {};

template <typename BasicJsonType, typename ConstructibleArrayType, typename = void>

struct is\_constructible\_array\_type\_impl : std::false\_type {};

template <typename BasicJsonType, typename ConstructibleArrayType>

struct is\_constructible\_array\_type\_impl <

BasicJsonType, ConstructibleArrayType,

enable\_if\_t<std::is\_same<ConstructibleArrayType,

typename BasicJsonType::value\_type>::value >>

: std::true\_type {};

template <typename BasicJsonType, typename ConstructibleArrayType>

struct is\_constructible\_array\_type\_impl <

BasicJsonType, ConstructibleArrayType,

enable\_if\_t<not std::is\_same<ConstructibleArrayType,

typename BasicJsonType::value\_type>::value and

std::is\_default\_constructible<ConstructibleArrayType>::value and

(std::is\_move\_assignable<ConstructibleArrayType>::value or

std::is\_copy\_assignable<ConstructibleArrayType>::value) and

is\_detected<value\_type\_t, ConstructibleArrayType>::value and

is\_detected<iterator\_t, ConstructibleArrayType>::value and

is\_complete\_type<

detected\_t<value\_type\_t, ConstructibleArrayType>>::value >>

{

static constexpr bool value =

// This is needed because json\_reverse\_iterator has a ::iterator type,

// furthermore, std::back\_insert\_iterator (and other iterators) have a

// base class `iterator`... Therefore it is detected as a

// ConstructibleArrayType. The real fix would be to have an Iterable

// concept.

not is\_iterator\_traits<iterator\_traits<ConstructibleArrayType>>::value and

(std::is\_same<typename ConstructibleArrayType::value\_type,

typename BasicJsonType::array\_t::value\_type>::value or

has\_from\_json<BasicJsonType,

typename ConstructibleArrayType::value\_type>::value or

has\_non\_default\_from\_json <

BasicJsonType, typename ConstructibleArrayType::value\_type >::value);

};

template <typename BasicJsonType, typename ConstructibleArrayType>

struct is\_constructible\_array\_type

: is\_constructible\_array\_type\_impl<BasicJsonType, ConstructibleArrayType> {};

template <typename RealIntegerType, typename CompatibleNumberIntegerType,

typename = void>

struct is\_compatible\_integer\_type\_impl : std::false\_type {};

template <typename RealIntegerType, typename CompatibleNumberIntegerType>

struct is\_compatible\_integer\_type\_impl <

RealIntegerType, CompatibleNumberIntegerType,

enable\_if\_t<std::is\_integral<RealIntegerType>::value and

std::is\_integral<CompatibleNumberIntegerType>::value and

not std::is\_same<bool, CompatibleNumberIntegerType>::value >>

{

// is there an assert somewhere on overflows?

using RealLimits = std::numeric\_limits<RealIntegerType>;

using CompatibleLimits = std::numeric\_limits<CompatibleNumberIntegerType>;

static constexpr auto value =

std::is\_constructible<RealIntegerType,

CompatibleNumberIntegerType>::value and

CompatibleLimits::is\_integer and

RealLimits::is\_signed == CompatibleLimits::is\_signed;

};

template <typename RealIntegerType, typename CompatibleNumberIntegerType>

struct is\_compatible\_integer\_type

: is\_compatible\_integer\_type\_impl<RealIntegerType,

CompatibleNumberIntegerType> {};

template <typename BasicJsonType, typename CompatibleType, typename = void>

struct is\_compatible\_type\_impl: std::false\_type {};

template <typename BasicJsonType, typename CompatibleType>

struct is\_compatible\_type\_impl <

BasicJsonType, CompatibleType,

enable\_if\_t<is\_complete\_type<CompatibleType>::value >>

{

static constexpr bool value =

has\_to\_json<BasicJsonType, CompatibleType>::value;

};

template <typename BasicJsonType, typename CompatibleType>

struct is\_compatible\_type

: is\_compatible\_type\_impl<BasicJsonType, CompatibleType> {};

// https://en.cppreference.com/w/cpp/types/conjunction

template<class...> struct conjunction : std::true\_type { };

template<class B1> struct conjunction<B1> : B1 { };

template<class B1, class... Bn>

struct conjunction<B1, Bn...>

: std::conditional<bool(B1::value), conjunction<Bn...>, B1>::type {};

template <typename T1, typename T2>

struct is\_constructible\_tuple : std::false\_type {};

template <typename T1, typename... Args>

struct is\_constructible\_tuple<T1, std::tuple<Args...>> : conjunction<std::is\_constructible<T1, Args>...> {};

} // namespace detail

} // namespace nlohmann

// #include <nlohmann/detail/value\_t.hpp>

#include <array> // array

#include <ciso646> // and

#include <cstddef> // size\_t

#include <cstdint> // uint8\_t

#include <string> // string

namespace nlohmann

{

namespace detail

{

///////////////////////////

// JSON type enumeration //

///////////////////////////

/\*!

@brief the JSON type enumeration

This enumeration collects the different JSON types. It is internally used to

distinguish the stored values, and the functions @ref basic\_json::is\_null(),

@ref basic\_json::is\_object(), @ref basic\_json::is\_array(),

@ref basic\_json::is\_string(), @ref basic\_json::is\_boolean(),

@ref basic\_json::is\_number() (with @ref basic\_json::is\_number\_integer(),

@ref basic\_json::is\_number\_unsigned(), and @ref basic\_json::is\_number\_float()),

@ref basic\_json::is\_discarded(), @ref basic\_json::is\_primitive(), and

@ref 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:

@ref basic\_json::number\_unsigned\_t is used for unsigned integers,

@ref basic\_json::number\_integer\_t is used for signed integers, and

@ref 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.

@sa @ref 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

\*/

enum class value\_t : std::uint8\_t

{

null, ///< null value

object, ///< object (unordered set of name/value pairs)

array, ///< array (ordered collection of values)

string, ///< string value

boolean, ///< boolean value

number\_integer, ///< number value (signed integer)

number\_unsigned, ///< number value (unsigned integer)

number\_float, ///< number value (floating-point)

discarded ///< discarded by the the parser callback function

};

/\*!

@brief comparison operator for JSON types

Returns an ordering that is similar to Python:

- order: null < boolean < number < object < array < string

- furthermore, each type is not smaller than itself

- discarded values are not comparable

@since version 1.0.0

\*/

inline bool operator<(const value\_t lhs, const value\_t rhs) noexcept

{

static constexpr std::array<std::uint8\_t, 8> order = {{

0 /\* null \*/, 3 /\* object \*/, 4 /\* array \*/, 5 /\* string \*/,

1 /\* boolean \*/, 2 /\* integer \*/, 2 /\* unsigned \*/, 2 /\* float \*/

}

};

const auto l\_index = static\_cast<std::size\_t>(lhs);

const auto r\_index = static\_cast<std::size\_t>(rhs);

return l\_index < order.size() and r\_index < order.size() and order[l\_index] < order[r\_index];

}

} // namespace detail

} // namespace nlohmann

namespace nlohmann

{

namespace detail

{

template<typename BasicJsonType>

void from\_json(const BasicJsonType& j, typename std::nullptr\_t& n)

{

if (JSON\_HEDLEY\_UNLIKELY(not j.is\_null()))

{

JSON\_THROW(type\_error::create(302, "type must be null, but is " + std::string(j.type\_name())));

}

n = nullptr;

}

// overloads for basic\_json template parameters

template<typename BasicJsonType, typename ArithmeticType,

enable\_if\_t<std::is\_arithmetic<ArithmeticType>::value and

not std::is\_same<ArithmeticType, typename BasicJsonType::boolean\_t>::value,

int> = 0>

void get\_arithmetic\_value(const BasicJsonType& j, ArithmeticType& val)

{

switch (static\_cast<value\_t>(j))

{

case value\_t::number\_unsigned:

{

val = static\_cast<ArithmeticType>(\*j.template get\_ptr<const typename BasicJsonType::number\_unsigned\_t\*>());

break;

}

case value\_t::number\_integer:

{

val = static\_cast<ArithmeticType>(\*j.template get\_ptr<const typename BasicJsonType::number\_integer\_t\*>());

break;

}

case value\_t::number\_float:

{

val = static\_cast<ArithmeticType>(\*j.template get\_ptr<const typename BasicJsonType::number\_float\_t\*>());

break;

}

default:

JSON\_THROW(type\_error::create(302, "type must be number, but is " + std::string(j.type\_name())));

}

}

template<typename BasicJsonType>

void from\_json(const BasicJsonType& j, typename BasicJsonType::boolean\_t& b)

{

if (JSON\_HEDLEY\_UNLIKELY(not j.is\_boolean()))

{

JSON\_THROW(type\_error::create(302, "type must be boolean, but is " + std::string(j.type\_name())));

}

b = \*j.template get\_ptr<const typename BasicJsonType::boolean\_t\*>();

}

template<typename BasicJsonType>

void from\_json(const BasicJsonType& j, typename BasicJsonType::string\_t& s)

{

if (JSON\_HEDLEY\_UNLIKELY(not j.is\_string()))

{

JSON\_THROW(type\_error::create(302, "type must be string, but is " + std::string(j.type\_name())));

}

s = \*j.template get\_ptr<const typename BasicJsonType::string\_t\*>();

}

template <

typename BasicJsonType, typename ConstructibleStringType,

enable\_if\_t <

is\_constructible\_string\_type<BasicJsonType, ConstructibleStringType>::value and

not std::is\_same<typename BasicJsonType::string\_t,

ConstructibleStringType>::value,

int > = 0 >

void from\_json(const BasicJsonType& j, ConstructibleStringType& s)

{

if (JSON\_HEDLEY\_UNLIKELY(not j.is\_string()))

{

JSON\_THROW(type\_error::create(302, "type must be string, but is " + std::string(j.type\_name())));

}

s = \*j.template get\_ptr<const typename BasicJsonType::string\_t\*>();

}

template<typename BasicJsonType>

void from\_json(const BasicJsonType& j, typename BasicJsonType::number\_float\_t& val)

{

get\_arithmetic\_value(j, val);

}

template<typename BasicJsonType>

void from\_json(const BasicJsonType& j, typename BasicJsonType::number\_unsigned\_t& val)

{

get\_arithmetic\_value(j, val);

}

template<typename BasicJsonType>

void from\_json(const BasicJsonType& j, typename BasicJsonType::number\_integer\_t& val)

{

get\_arithmetic\_value(j, val);

}

template<typename BasicJsonType, typename EnumType,

enable\_if\_t<std::is\_enum<EnumType>::value, int> = 0>

void from\_json(const BasicJsonType& j, EnumType& e)

{

typename std::underlying\_type<EnumType>::type val;

get\_arithmetic\_value(j, val);

e = static\_cast<EnumType>(val);

}

// forward\_list doesn't have an insert method

template<typename BasicJsonType, typename T, typename Allocator,

enable\_if\_t<std::is\_convertible<BasicJsonType, T>::value, int> = 0>

void from\_json(const BasicJsonType& j, std::forward\_list<T, Allocator>& l)

{

if (JSON\_HEDLEY\_UNLIKELY(not j.is\_array()))

{

JSON\_THROW(type\_error::create(302, "type must be array, but is " + std::string(j.type\_name())));

}

l.clear();

std::transform(j.rbegin(), j.rend(),

std::front\_inserter(l), [](const BasicJsonType & i)

{

return i.template get<T>();

});

}

// valarray doesn't have an insert method

template<typename BasicJsonType, typename T,

enable\_if\_t<std::is\_convertible<BasicJsonType, T>::value, int> = 0>

void from\_json(const BasicJsonType& j, std::valarray<T>& l)

{

if (JSON\_HEDLEY\_UNLIKELY(not j.is\_array()))

{

JSON\_THROW(type\_error::create(302, "type must be array, but is " + std::string(j.type\_name())));

}

l.resize(j.size());

std::copy(j.begin(), j.end(), std::begin(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())

{

for (std::size\_t i = 0; i < N; ++i)

{

arr[i] = j.at(i).template get<T>();

}

}

template<typename BasicJsonType>

void from\_json\_array\_impl(const BasicJsonType& j, typename BasicJsonType::array\_t& arr, priority\_tag<3> /\*unused\*/)

{

arr = \*j.template get\_ptr<const typename BasicJsonType::array\_t\*>();

}

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> /\*unused\*/)

-> decltype(j.template get<T>(), void())

{

for (std::size\_t i = 0; i < N; ++i)

{

arr[i] = j.at(i).template get<T>();

}

}

template<typename BasicJsonType, typename ConstructibleArrayType>

auto from\_json\_array\_impl(const BasicJsonType& j, ConstructibleArrayType& arr, priority\_tag<1> /\*unused\*/)

-> decltype(

arr.reserve(std::declval<typename ConstructibleArrayType::size\_type>()),

j.template get<typename ConstructibleArrayType::value\_type>(),

void())

{

using std::end;

ConstructibleArrayType ret;

ret.reserve(j.size());

std::transform(j.begin(), j.end(),

std::inserter(ret, end(ret)), [](const BasicJsonType & i)

{

// get<BasicJsonType>() returns \*this, this won't call a from\_json

// method when value\_type is BasicJsonType

return i.template get<typename ConstructibleArrayType::value\_type>();

});

arr = std::move(ret);

}

template <typename BasicJsonType, typename ConstructibleArrayType>

void from\_json\_array\_impl(const BasicJsonType& j, ConstructibleArrayType& arr,

priority\_tag<0> /\*unused\*/)

{

using std::end;

ConstructibleArrayType ret;

std::transform(

j.begin(), j.end(), std::inserter(ret, end(ret)),

[](const BasicJsonType & i)

{

// get<BasicJsonType>() returns \*this, this won't call a from\_json

// method when value\_type is BasicJsonType

return i.template get<typename ConstructibleArrayType::value\_type>();

});

arr = std::move(ret);

}

template <typename BasicJsonType, typename ConstructibleArrayType,

enable\_if\_t <

is\_constructible\_array\_type<BasicJsonType, ConstructibleArrayType>::value and

not is\_constructible\_object\_type<BasicJsonType, ConstructibleArrayType>::value and

not is\_constructible\_string\_type<BasicJsonType, ConstructibleArrayType>::value and

not 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())

{

if (JSON\_HEDLEY\_UNLIKELY(not j.is\_array()))

{

JSON\_THROW(type\_error::create(302, "type must be array, but is " +

std::string(j.type\_name())));

}

from\_json\_array\_impl(j, arr, priority\_tag<3> {});

}

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)

{

if (JSON\_HEDLEY\_UNLIKELY(not j.is\_object()))

{

JSON\_THROW(type\_error::create(302, "type must be object, but is " + std::string(j.type\_name())));

}

ConstructibleObjectType ret;

auto inner\_object = j.template get\_ptr<const typename BasicJsonType::object\_t\*>();

using value\_type = typename ConstructibleObjectType::value\_type;

std::transform(

inner\_object->begin(), inner\_object->end(),

std::inserter(ret, ret.begin()),

[](typename BasicJsonType::object\_t::value\_type const & p)

{

return value\_type(p.first, p.second.template get<typename ConstructibleObjectType::mapped\_type>());

});

obj = std::move(ret);

}

// overload for arithmetic types, not chosen for basic\_json template arguments

// (BooleanType, etc..); note: Is it really necessary to provide explicit

// overloads for boolean\_t etc. in case of a custom BooleanType which is not

// an arithmetic type?

template<typename BasicJsonType, typename ArithmeticType,

enable\_if\_t <

std::is\_arithmetic<ArithmeticType>::value and

not std::is\_same<ArithmeticType, typename BasicJsonType::number\_unsigned\_t>::value and

not std::is\_same<ArithmeticType, typename BasicJsonType::number\_integer\_t>::value and

not std::is\_same<ArithmeticType, typename BasicJsonType::number\_float\_t>::value and

not std::is\_same<ArithmeticType, typename BasicJsonType::boolean\_t>::value,

int> = 0>

void from\_json(const BasicJsonType& j, ArithmeticType& val)

{

switch (static\_cast<value\_t>(j))

{

case value\_t::number\_unsigned:

{

val = static\_cast<ArithmeticType>(\*j.template get\_ptr<const typename BasicJsonType::number\_unsigned\_t\*>());

break;

}

case value\_t::number\_integer:

{

val = static\_cast<ArithmeticType>(\*j.template get\_ptr<const typename BasicJsonType::number\_integer\_t\*>());

break;

}

case value\_t::number\_float:

{

val = static\_cast<ArithmeticType>(\*j.template get\_ptr<const typename BasicJsonType::number\_float\_t\*>());

break;

}

case value\_t::boolean:

{

val = static\_cast<ArithmeticType>(\*j.template get\_ptr<const typename BasicJsonType::boolean\_t\*>());

break;

}

default:

JSON\_THROW(type\_error::create(302, "type must be number, but is " + std::string(j.type\_name())));

}

}

template<typename BasicJsonType, typename A1, typename A2>

void from\_json(const BasicJsonType& j, std::pair<A1, A2>& p)

{

p = {j.at(0).template get<A1>(), j.at(1).template get<A2>()};

}

template<typename BasicJsonType, typename Tuple, std::size\_t... Idx>

void from\_json\_tuple\_impl(const BasicJsonType& j, Tuple& t, index\_sequence<Idx...> /\*unused\*/)

{

t = std::make\_tuple(j.at(Idx).template get<typename std::tuple\_element<Idx, Tuple>::type>()...);

}

template<typename BasicJsonType, typename... Args>

void from\_json(const BasicJsonType& j, std::tuple<Args...>& t)

{

from\_json\_tuple\_impl(j, t, index\_sequence\_for<Args...> {});

}

template <typename BasicJsonType, typename Key, typename Value, typename Compare, typename Allocator,

typename = enable\_if\_t<not std::is\_constructible<

typename BasicJsonType::string\_t, Key>::value>>

void from\_json(const BasicJsonType& j, std::map<Key, Value, Compare, Allocator>& m)

{

if (JSON\_HEDLEY\_UNLIKELY(not j.is\_array()))

{

JSON\_THROW(type\_error::create(302, "type must be array, but is " + std::string(j.type\_name())));

}

m.clear();

for (const auto& p : j)

{

if (JSON\_HEDLEY\_UNLIKELY(not p.is\_array()))

{

JSON\_THROW(type\_error::create(302, "type must be array, but is " + std::string(p.type\_name())));

}

m.emplace(p.at(0).template get<Key>(), p.at(1).template get<Value>());

}

}

template <typename BasicJsonType, typename Key, typename Value, typename Hash, typename KeyEqual, typename Allocator,

typename = enable\_if\_t<not std::is\_constructible<

typename BasicJsonType::string\_t, Key>::value>>

void from\_json(const BasicJsonType& j, std::unordered\_map<Key, Value, Hash, KeyEqual, Allocator>& m)

{

if (JSON\_HEDLEY\_UNLIKELY(not j.is\_array()))

{

JSON\_THROW(type\_error::create(302, "type must be array, but is " + std::string(j.type\_name())));

}

m.clear();

for (const auto& p : j)

{

if (JSON\_HEDLEY\_UNLIKELY(not p.is\_array()))

{

JSON\_THROW(type\_error::create(302, "type must be array, but is " + std::string(p.type\_name())));

}

m.emplace(p.at(0).template get<Key>(), p.at(1).template get<Value>());

}

}

struct from\_json\_fn

{

template<typename BasicJsonType, typename T>

auto operator()(const BasicJsonType& j, T& val) const

noexcept(noexcept(from\_json(j, val)))

-> decltype(from\_json(j, val), void())

{

return from\_json(j, val);

}

};

} // namespace detail

/// namespace to hold default `from\_json` function

/// to see why this is required:

/// http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4381.html

namespace

{

constexpr const auto& from\_json = detail::static\_const<detail::from\_json\_fn>::value;

} // namespace

} // namespace nlohmann

// #include <nlohmann/detail/conversions/to\_json.hpp>

#include <algorithm> // copy

#include <ciso646> // or, and, not

#include <iterator> // begin, end

#include <string> // string

#include <tuple> // tuple, get

#include <type\_traits> // is\_same, is\_constructible, is\_floating\_point, is\_enum, underlying\_type

#include <utility> // move, forward, declval, pair

#include <valarray> // valarray

#include <vector> // vector

// #include <nlohmann/detail/iterators/iteration\_proxy.hpp>

#include <cstddef> // size\_t

#include <iterator> // input\_iterator\_tag

#include <string> // string, to\_string

#include <tuple> // tuple\_size, get, tuple\_element

// #include <nlohmann/detail/meta/type\_traits.hpp>

// #include <nlohmann/detail/value\_t.hpp>

namespace nlohmann

{

namespace detail

{

template<typename string\_type>

void int\_to\_string( string\_type& target, std::size\_t value )

{

target = std::to\_string(value);

}

template <typename IteratorType> class iteration\_proxy\_value

{

public:

using difference\_type = std::ptrdiff\_t;

using value\_type = iteration\_proxy\_value;

using pointer = value\_type \* ;

using reference = value\_type & ;

using iterator\_category = std::input\_iterator\_tag;

using string\_type = typename std::remove\_cv< typename std::remove\_reference<decltype( std::declval<IteratorType>().key() ) >::type >::type;

private:

/// the iterator

IteratorType anchor;

/// an index for arrays (used to create key names)

std::size\_t array\_index = 0;

/// last stringified array index

mutable std::size\_t array\_index\_last = 0;

/// a string representation of the array index

mutable string\_type array\_index\_str = "0";

/// an empty string (to return a reference for primitive values)

const string\_type empty\_str = "";

public:

explicit iteration\_proxy\_value(IteratorType it) noexcept : anchor(it) {}

/// dereference operator (needed for range-based for)

iteration\_proxy\_value& operator\*()

{

return \*this;

}

/// increment operator (needed for range-based for)

iteration\_proxy\_value& operator++()

{

++anchor;

++array\_index;

return \*this;

}

/// equality operator (needed for InputIterator)

bool operator==(const iteration\_proxy\_value& o) const

{

return anchor == o.anchor;

}

/// inequality operator (needed for range-based for)

bool operator!=(const iteration\_proxy\_value& o) const

{

return anchor != o.anchor;

}

/// return key of the iterator

const string\_type& key() const

{

assert(anchor.m\_object != nullptr);

switch (anchor.m\_object->type())

{

// use integer array index as key

case value\_t::array:

{

if (array\_index != array\_index\_last)

{

int\_to\_string( array\_index\_str, array\_index );

array\_index\_last = array\_index;

}

return array\_index\_str;

}

// use key from the object

case value\_t::object:

return anchor.key();

// use an empty key for all primitive types

default:

return empty\_str;

}

}

/// return value of the iterator

typename IteratorType::reference value() const

{

return anchor.value();

}

};

/// proxy class for the items() function

template<typename IteratorType> class iteration\_proxy

{

private:

/// the container to iterate

typename IteratorType::reference container;

public:

/// construct iteration proxy from a container

explicit iteration\_proxy(typename IteratorType::reference cont) noexcept

: container(cont) {}

/// return iterator begin (needed for range-based for)

iteration\_proxy\_value<IteratorType> begin() noexcept

{

return iteration\_proxy\_value<IteratorType>(container.begin());

}

/// return iterator end (needed for range-based for)

iteration\_proxy\_value<IteratorType> end() noexcept

{

return iteration\_proxy\_value<IteratorType>(container.end());

}

};

// Structured Bindings Support

// For further reference see https://blog.tartanllama.xyz/structured-bindings/

// And see https://github.com/nlohmann/json/pull/1391

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())

{

return i.key();

}

// Structured Bindings Support

// For further reference see https://blog.tartanllama.xyz/structured-bindings/

// And see https://github.com/nlohmann/json/pull/1391

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())

{

return i.value();

}

} // namespace detail

} // namespace nlohmann

// The Addition to the STD Namespace is required to add

// Structured Bindings Support to the iteration\_proxy\_value class

// For further reference see https://blog.tartanllama.xyz/structured-bindings/

// And see https://github.com/nlohmann/json/pull/1391

namespace std

{

#if defined(\_\_clang\_\_)

// Fix: https://github.com/nlohmann/json/issues/1401

#pragma clang diagnostic push

#pragma clang diagnostic ignored "-Wmismatched-tags"

#endif

template <typename IteratorType>

class tuple\_size<::nlohmann::detail::iteration\_proxy\_value<IteratorType>>

: public std::integral\_constant<std::size\_t, 2> {};

template <std::size\_t N, typename IteratorType>

class tuple\_element<N, ::nlohmann::detail::iteration\_proxy\_value<IteratorType >>

{

public:

using type = decltype(

get<N>(std::declval <

::nlohmann::detail::iteration\_proxy\_value<IteratorType >> ()));

};

#if defined(\_\_clang\_\_)

#pragma clang diagnostic pop

#endif

} // namespace std

// #include <nlohmann/detail/meta/cpp\_future.hpp>

// #include <nlohmann/detail/meta/type\_traits.hpp>

// #include <nlohmann/detail/value\_t.hpp>

namespace nlohmann

{

namespace detail

{

//////////////////

// constructors //

//////////////////

template<value\_t> struct external\_constructor;

template<>

struct external\_constructor<value\_t::boolean>

{

template<typename BasicJsonType>

static void construct(BasicJsonType& j, typename BasicJsonType::boolean\_t b) noexcept

{

j.m\_type = value\_t::boolean;

j.m\_value = b;

j.assert\_invariant();

}

};

template<>

struct external\_constructor<value\_t::string>

{

template<typename BasicJsonType>

static void construct(BasicJsonType& j, const typename BasicJsonType::string\_t& s)

{

j.m\_type = value\_t::string;

j.m\_value = s;

j.assert\_invariant();

}

template<typename BasicJsonType>

static void construct(BasicJsonType& j, typename BasicJsonType::string\_t&& s)

{

j.m\_type = value\_t::string;

j.m\_value = std::move(s);

j.assert\_invariant();

}

template<typename BasicJsonType, typename CompatibleStringType,

enable\_if\_t<not std::is\_same<CompatibleStringType, typename BasicJsonType::string\_t>::value,

int> = 0>

static void construct(BasicJsonType& j, const CompatibleStringType& str)

{

j.m\_type = value\_t::string;

j.m\_value.string = j.template create<typename BasicJsonType::string\_t>(str);

j.assert\_invariant();

}

};

template<>

struct external\_constructor<value\_t::number\_float>

{

template<typename BasicJsonType>

static void construct(BasicJsonType& j, typename BasicJsonType::number\_float\_t val) noexcept

{

j.m\_type = value\_t::number\_float;

j.m\_value = val;

j.assert\_invariant();

}

};

template<>

struct external\_constructor<value\_t::number\_unsigned>

{

template<typename BasicJsonType>

static void construct(BasicJsonType& j, typename BasicJsonType::number\_unsigned\_t val) noexcept

{

j.m\_type = value\_t::number\_unsigned;

j.m\_value = val;

j.assert\_invariant();

}

};

template<>

struct external\_constructor<value\_t::number\_integer>

{

template<typename BasicJsonType>

static void construct(BasicJsonType& j, typename BasicJsonType::number\_integer\_t val) noexcept

{

j.m\_type = value\_t::number\_integer;

j.m\_value = val;

j.assert\_invariant();

}

};

template<>

struct external\_constructor<value\_t::array>

{

template<typename BasicJsonType>

static void construct(BasicJsonType& j, const typename BasicJsonType::array\_t& arr)

{

j.m\_type = value\_t::array;

j.m\_value = arr;

j.assert\_invariant();

}

template<typename BasicJsonType>

static void construct(BasicJsonType& j, typename BasicJsonType::array\_t&& arr)

{

j.m\_type = value\_t::array;

j.m\_value = std::move(arr);

j.assert\_invariant();

}

template<typename BasicJsonType, typename CompatibleArrayType,

enable\_if\_t<not std::is\_same<CompatibleArrayType, typename BasicJsonType::array\_t>::value,

int> = 0>

static void construct(BasicJsonType& j, const CompatibleArrayType& arr)

{

using std::begin;

using std::end;

j.m\_type = value\_t::array;

j.m\_value.array = j.template create<typename BasicJsonType::array\_t>(begin(arr), end(arr));

j.assert\_invariant();

}

template<typename BasicJsonType>

static void construct(BasicJsonType& j, const std::vector<bool>& arr)

{

j.m\_type = value\_t::array;

j.m\_value = value\_t::array;

j.m\_value.array->reserve(arr.size());

for (const bool x : arr)

{

j.m\_value.array->push\_back(x);

}

j.assert\_invariant();

}

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)

{

j.m\_type = value\_t::array;

j.m\_value = value\_t::array;

j.m\_value.array->resize(arr.size());

if (arr.size() > 0)

{

std::copy(std::begin(arr), std::end(arr), j.m\_value.array->begin());

}

j.assert\_invariant();

}

};

template<>

struct external\_constructor<value\_t::object>

{

template<typename BasicJsonType>

static void construct(BasicJsonType& j, const typename BasicJsonType::object\_t& obj)

{

j.m\_type = value\_t::object;

j.m\_value = obj;

j.assert\_invariant();

}

template<typename BasicJsonType>

static void construct(BasicJsonType& j, typename BasicJsonType::object\_t&& obj)

{

j.m\_type = value\_t::object;

j.m\_value = std::move(obj);

j.assert\_invariant();

}

template<typename BasicJsonType, typename CompatibleObjectType,

enable\_if\_t<not std::is\_same<CompatibleObjectType, typename BasicJsonType::object\_t>::value, int> = 0>

static void construct(BasicJsonType& j, const CompatibleObjectType& obj)

{

using std::begin;

using std::end;

j.m\_type = value\_t::object;

j.m\_value.object = j.template create<typename BasicJsonType::object\_t>(begin(obj), end(obj));

j.assert\_invariant();

}

};

/////////////

// to\_json //

/////////////

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

{

external\_constructor<value\_t::boolean>::construct(j, b);

}

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)

{

external\_constructor<value\_t::string>::construct(j, s);

}

template<typename BasicJsonType>

void to\_json(BasicJsonType& j, typename BasicJsonType::string\_t&& s)

{

external\_constructor<value\_t::string>::construct(j, std::move(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

{

external\_constructor<value\_t::number\_float>::construct(j, static\_cast<typename BasicJsonType::number\_float\_t>(val));

}

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

{

external\_constructor<value\_t::number\_unsigned>::construct(j, static\_cast<typename BasicJsonType::number\_unsigned\_t>(val));

}

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

{

external\_constructor<value\_t::number\_integer>::construct(j, static\_cast<typename BasicJsonType::number\_integer\_t>(val));

}

template<typename BasicJsonType, typename EnumType,

enable\_if\_t<std::is\_enum<EnumType>::value, int> = 0>

void to\_json(BasicJsonType& j, EnumType e) noexcept

{

using underlying\_type = typename std::underlying\_type<EnumType>::type;

external\_constructor<value\_t::number\_integer>::construct(j, static\_cast<underlying\_type>(e));

}

template<typename BasicJsonType>

void to\_json(BasicJsonType& j, const std::vector<bool>& e)

{

external\_constructor<value\_t::array>::construct(j, e);

}

template <typename BasicJsonType, typename CompatibleArrayType,

enable\_if\_t<is\_compatible\_array\_type<BasicJsonType,

CompatibleArrayType>::value and

not is\_compatible\_object\_type<

BasicJsonType, CompatibleArrayType>::value and

not is\_compatible\_string\_type<BasicJsonType, CompatibleArrayType>::value and

not is\_basic\_json<CompatibleArrayType>::value,

int> = 0>

void to\_json(BasicJsonType& j, const CompatibleArrayType& arr)

{

external\_constructor<value\_t::array>::construct(j, arr);

}

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)

{

external\_constructor<value\_t::array>::construct(j, std::move(arr));

}

template<typename BasicJsonType>

void to\_json(BasicJsonType& j, typename BasicJsonType::array\_t&& arr)

{

external\_constructor<value\_t::array>::construct(j, std::move(arr));

}

template<typename BasicJsonType, typename CompatibleObjectType,

enable\_if\_t<is\_compatible\_object\_type<BasicJsonType, CompatibleObjectType>::value and not is\_basic\_json<CompatibleObjectType>::value, int> = 0>

void to\_json(BasicJsonType& j, const CompatibleObjectType& obj)

{

external\_constructor<value\_t::object>::construct(j, obj);

}

template<typename BasicJsonType>

void to\_json(BasicJsonType& j, typename BasicJsonType::object\_t&& obj)

{

external\_constructor<value\_t::object>::construct(j, std::move(obj));

}

template <

typename BasicJsonType, typename T, std::size\_t N,

enable\_if\_t<not std::is\_constructible<typename BasicJsonType::string\_t,

const T(&)[N]>::value,

int> = 0 >

void to\_json(BasicJsonType& j, const T(&arr)[N])

{

external\_constructor<value\_t::array>::construct(j, arr);

}

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)

{

j = { p.first, p.second };

}

// for https://github.com/nlohmann/json/pull/1134

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)

{

j = { {b.key(), b.value()} };

}

template<typename BasicJsonType, typename Tuple, std::size\_t... Idx>

void to\_json\_tuple\_impl(BasicJsonType& j, const Tuple& t, index\_sequence<Idx...> /\*unused\*/)

{

j = { std::get<Idx>(t)... };

}

template<typename BasicJsonType, typename T, enable\_if\_t<is\_constructible\_tuple<BasicJsonType, T>::value, int > = 0>

void to\_json(BasicJsonType& j, const T& t)

{

to\_json\_tuple\_impl(j, t, make\_index\_sequence<std::tuple\_size<T>::value> {});

}

struct to\_json\_fn

{

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())

{

return to\_json(j, std::forward<T>(val));

}

};

} // namespace detail

/// namespace to hold default `to\_json` function

namespace

{

constexpr const auto& to\_json = detail::static\_const<detail::to\_json\_fn>::value;

} // namespace

} // namespace nlohmann

namespace nlohmann

{

template<typename, typename>

struct adl\_serializer

{

/\*!

@brief convert a JSON value to any value type

This function is usually called by the `get()` function of the

@ref basic\_json class (either explicit or via conversion operators).

@param[in] j JSON value to read from

@param[in,out] val value to write to

\*/

template<typename BasicJsonType, typename ValueType>

static auto from\_json(BasicJsonType&& j, ValueType& val) noexcept(

noexcept(::nlohmann::from\_json(std::forward<BasicJsonType>(j), val)))

-> decltype(::nlohmann::from\_json(std::forward<BasicJsonType>(j), val), void())

{

::nlohmann::from\_json(std::forward<BasicJsonType>(j), val);

}

/\*!

@brief convert any value type to a JSON value

This function is usually called by the constructors of the @ref basic\_json

class.

@param[in,out] j JSON value to write to

@param[in] val value to read from

\*/

template <typename BasicJsonType, typename ValueType>

static auto to\_json(BasicJsonType& j, ValueType&& val) noexcept(

noexcept(::nlohmann::to\_json(j, std::forward<ValueType>(val))))

-> decltype(::nlohmann::to\_json(j, std::forward<ValueType>(val)), void())

{

::nlohmann::to\_json(j, std::forward<ValueType>(val));

}

};

} // namespace nlohmann

// #include <nlohmann/detail/conversions/from\_json.hpp>

// #include <nlohmann/detail/conversions/to\_json.hpp>

// #include <nlohmann/detail/exceptions.hpp>

// #include <nlohmann/detail/input/binary\_reader.hpp>

#include <algorithm> // generate\_n

#include <array> // array

#include <cassert> // assert

#include <cmath> // ldexp

#include <cstddef> // size\_t

#include <cstdint> // uint8\_t, uint16\_t, uint32\_t, uint64\_t

#include <cstdio> // snprintf

#include <cstring> // memcpy

#include <iterator> // back\_inserter

#include <limits> // numeric\_limits

#include <string> // char\_traits, string

#include <utility> // make\_pair, move

// #include <nlohmann/detail/exceptions.hpp>

// #include <nlohmann/detail/input/input\_adapters.hpp>

#include <array> // array

#include <cassert> // assert

#include <cstddef> // size\_t

#include <cstdio> //FILE \*

#include <cstring> // strlen

#include <istream> // istream

#include <iterator> // begin, end, iterator\_traits, random\_access\_iterator\_tag, distance, next

#include <memory> // shared\_ptr, make\_shared, addressof

#include <numeric> // accumulate

#include <string> // string, char\_traits

#include <type\_traits> // enable\_if, is\_base\_of, is\_pointer, is\_integral, remove\_pointer

#include <utility> // pair, declval

// #include <nlohmann/detail/iterators/iterator\_traits.hpp>

// #include <nlohmann/detail/macro\_scope.hpp>

namespace nlohmann

{

namespace detail

{

/// the supported input formats

enum class input\_format\_t { json, cbor, msgpack, ubjson, bson };

////////////////////

// input adapters //

////////////////////

/\*!

@brief abstract input adapter interface

Produces a stream of std::char\_traits<char>::int\_type characters from a

std::istream, a buffer, or some other input type. Accepts the return of

exactly one non-EOF character for future input. The int\_type characters

returned consist of all valid char values as positive values (typically

unsigned char), plus an EOF value outside that range, specified by the value

of the function std::char\_traits<char>::eof(). This value is typically -1, but

could be any arbitrary value which is not a valid char value.

\*/

struct input\_adapter\_protocol

{

/// get a character [0,255] or std::char\_traits<char>::eof().

virtual std::char\_traits<char>::int\_type get\_character() = 0;

virtual ~input\_adapter\_protocol() = default;

};

/// a type to simplify interfaces

using input\_adapter\_t = std::shared\_ptr<input\_adapter\_protocol>;

/\*!

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.

\*/

class file\_input\_adapter : public input\_adapter\_protocol

{

public:

JSON\_HEDLEY\_NON\_NULL(2)

explicit file\_input\_adapter(std::FILE\* f) noexcept

: m\_file(f)

{}

// make class move-only

file\_input\_adapter(const file\_input\_adapter&) = delete;

file\_input\_adapter(file\_input\_adapter&&) = default;

file\_input\_adapter& operator=(const file\_input\_adapter&) = delete;

file\_input\_adapter& operator=(file\_input\_adapter&&) = default;

~file\_input\_adapter() override = default;

std::char\_traits<char>::int\_type get\_character() noexcept override

{

return std::fgetc(m\_file);

}

private:

/// the file pointer to read from

std::FILE\* m\_file;

};

/\*!

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.

\*/

class input\_stream\_adapter : public input\_adapter\_protocol

{

public:

~input\_stream\_adapter() override

{

// clear stream flags; we use underlying streambuf I/O, do not

// maintain ifstream flags, except eof

is.clear(is.rdstate() & std::ios::eofbit);

}

explicit input\_stream\_adapter(std::istream& i)

: is(i), sb(\*i.rdbuf())

{}

// delete because of pointer members

input\_stream\_adapter(const input\_stream\_adapter&) = delete;

input\_stream\_adapter& operator=(input\_stream\_adapter&) = delete;

input\_stream\_adapter(input\_stream\_adapter&&) = delete;

input\_stream\_adapter& operator=(input\_stream\_adapter&&) = delete;

// std::istream/std::streambuf use std::char\_traits<char>::to\_int\_type, to

// ensure that std::char\_traits<char>::eof() and the character 0xFF do not

// end up as the same value, eg. 0xFFFFFFFF.

std::char\_traits<char>::int\_type get\_character() override

{

auto res = sb.sbumpc();

// set eof manually, as we don't use the istream interface.

if (res == EOF)

{

is.clear(is.rdstate() | std::ios::eofbit);

}

return res;

}

private:

/// the associated input stream

std::istream& is;

std::streambuf& sb;

};

/// input adapter for buffer input

class input\_buffer\_adapter : public input\_adapter\_protocol

{

public:

input\_buffer\_adapter(const char\* b, const std::size\_t l) noexcept

: cursor(b), limit(b == nullptr ? nullptr : (b + l))

{}

// delete because of pointer members

input\_buffer\_adapter(const input\_buffer\_adapter&) = delete;

input\_buffer\_adapter& operator=(input\_buffer\_adapter&) = delete;

input\_buffer\_adapter(input\_buffer\_adapter&&) = delete;

input\_buffer\_adapter& operator=(input\_buffer\_adapter&&) = delete;

~input\_buffer\_adapter() override = default;

std::char\_traits<char>::int\_type get\_character() noexcept override

{

if (JSON\_HEDLEY\_LIKELY(cursor < limit))

{

assert(cursor != nullptr and limit != nullptr);

return std::char\_traits<char>::to\_int\_type(\*(cursor++));

}

return std::char\_traits<char>::eof();

}

private:

/// pointer to the current character

const char\* cursor;

/// pointer past the last character

const char\* const limit;

};

template<typename WideStringType, size\_t T>

struct wide\_string\_input\_helper

{

// UTF-32

static void fill\_buffer(const WideStringType& str,

size\_t& current\_wchar,

std::array<std::char\_traits<char>::int\_type, 4>& utf8\_bytes,

size\_t& utf8\_bytes\_index,

size\_t& utf8\_bytes\_filled)

{

utf8\_bytes\_index = 0;

if (current\_wchar == str.size())

{

utf8\_bytes[0] = std::char\_traits<char>::eof();

utf8\_bytes\_filled = 1;

}

else

{

// get the current character

const auto wc = static\_cast<unsigned int>(str[current\_wchar++]);

// UTF-32 to UTF-8 encoding

if (wc < 0x80)

{

utf8\_bytes[0] = static\_cast<std::char\_traits<char>::int\_type>(wc);

utf8\_bytes\_filled = 1;

}

else if (wc <= 0x7FF)

{

utf8\_bytes[0] = static\_cast<std::char\_traits<char>::int\_type>(0xC0u | ((wc >> 6u) & 0x1Fu));

utf8\_bytes[1] = static\_cast<std::char\_traits<char>::int\_type>(0x80u | (wc & 0x3Fu));

utf8\_bytes\_filled = 2;

}

else if (wc <= 0xFFFF)

{

utf8\_bytes[0] = static\_cast<std::char\_traits<char>::int\_type>(0xE0u | ((wc >> 12u) & 0x0Fu));

utf8\_bytes[1] = static\_cast<std::char\_traits<char>::int\_type>(0x80u | ((wc >> 6u) & 0x3Fu));

utf8\_bytes[2] = static\_cast<std::char\_traits<char>::int\_type>(0x80u | (wc & 0x3Fu));

utf8\_bytes\_filled = 3;

}

else if (wc <= 0x10FFFF)

{

utf8\_bytes[0] = static\_cast<std::char\_traits<char>::int\_type>(0xF0u | ((wc >> 18u) & 0x07u));

utf8\_bytes[1] = static\_cast<std::char\_traits<char>::int\_type>(0x80u | ((wc >> 12u) & 0x3Fu));

utf8\_bytes[2] = static\_cast<std::char\_traits<char>::int\_type>(0x80u | ((wc >> 6u) & 0x3Fu));

utf8\_bytes[3] = static\_cast<std::char\_traits<char>::int\_type>(0x80u | (wc & 0x3Fu));

utf8\_bytes\_filled = 4;

}

else

{

// unknown character

utf8\_bytes[0] = static\_cast<std::char\_traits<char>::int\_type>(wc);

utf8\_bytes\_filled = 1;

}

}

}

};

template<typename WideStringType>

struct wide\_string\_input\_helper<WideStringType, 2>

{

// UTF-16

static void fill\_buffer(const WideStringType& str,

size\_t& current\_wchar,

std::array<std::char\_traits<char>::int\_type, 4>& utf8\_bytes,

size\_t& utf8\_bytes\_index,

size\_t& utf8\_bytes\_filled)

{

utf8\_bytes\_index = 0;

if (current\_wchar == str.size())

{

utf8\_bytes[0] = std::char\_traits<char>::eof();

utf8\_bytes\_filled = 1;

}

else

{

// get the current character

const auto wc = static\_cast<unsigned int>(str[current\_wchar++]);

// UTF-16 to UTF-8 encoding

if (wc < 0x80)

{

utf8\_bytes[0] = static\_cast<std::char\_traits<char>::int\_type>(wc);

utf8\_bytes\_filled = 1;

}

else if (wc <= 0x7FF)

{

utf8\_bytes[0] = static\_cast<std::char\_traits<char>::int\_type>(0xC0u | ((wc >> 6u)));

utf8\_bytes[1] = static\_cast<std::char\_traits<char>::int\_type>(0x80u | (wc & 0x3Fu));

utf8\_bytes\_filled = 2;

}

else if (0xD800 > wc or wc >= 0xE000)

{

utf8\_bytes[0] = static\_cast<std::char\_traits<char>::int\_type>(0xE0u | ((wc >> 12u)));

utf8\_bytes[1] = static\_cast<std::char\_traits<char>::int\_type>(0x80u | ((wc >> 6u) & 0x3Fu));

utf8\_bytes[2] = static\_cast<std::char\_traits<char>::int\_type>(0x80u | (wc & 0x3Fu));

utf8\_bytes\_filled = 3;

}

else

{

if (current\_wchar < str.size())

{

const auto wc2 = static\_cast<unsigned int>(str[current\_wchar++]);

const auto charcode = 0x10000u + (((wc & 0x3FFu) << 10u) | (wc2 & 0x3FFu));

utf8\_bytes[0] = static\_cast<std::char\_traits<char>::int\_type>(0xF0u | (charcode >> 18u));

utf8\_bytes[1] = static\_cast<std::char\_traits<char>::int\_type>(0x80u | ((charcode >> 12u) & 0x3Fu));

utf8\_bytes[2] = static\_cast<std::char\_traits<char>::int\_type>(0x80u | ((charcode >> 6u) & 0x3Fu));

utf8\_bytes[3] = static\_cast<std::char\_traits<char>::int\_type>(0x80u | (charcode & 0x3Fu));

utf8\_bytes\_filled = 4;

}

else

{

// unknown character

++current\_wchar;

utf8\_bytes[0] = static\_cast<std::char\_traits<char>::int\_type>(wc);

utf8\_bytes\_filled = 1;

}

}

}

}

};

template<typename WideStringType>

class wide\_string\_input\_adapter : public input\_adapter\_protocol

{

public:

explicit wide\_string\_input\_adapter(const WideStringType& w) noexcept

: str(w)

{}

std::char\_traits<char>::int\_type get\_character() noexcept override

{

// check if buffer needs to be filled

if (utf8\_bytes\_index == utf8\_bytes\_filled)

{

fill\_buffer<sizeof(typename WideStringType::value\_type)>();

assert(utf8\_bytes\_filled > 0);

assert(utf8\_bytes\_index == 0);

}

// use buffer

assert(utf8\_bytes\_filled > 0);

assert(utf8\_bytes\_index < utf8\_bytes\_filled);

return utf8\_bytes[utf8\_bytes\_index++];

}

private:

template<size\_t T>

void fill\_buffer()

{

wide\_string\_input\_helper<WideStringType, T>::fill\_buffer(str, current\_wchar, utf8\_bytes, utf8\_bytes\_index, utf8\_bytes\_filled);

}

/// the wstring to process

const WideStringType& str;

/// index of the current wchar in str

std::size\_t current\_wchar = 0;

/// a buffer for UTF-8 bytes

std::array<std::char\_traits<char>::int\_type, 4> utf8\_bytes = {{0, 0, 0, 0}};

/// index to the utf8\_codes array for the next valid byte

std::size\_t utf8\_bytes\_index = 0;

/// number of valid bytes in the utf8\_codes array

std::size\_t utf8\_bytes\_filled = 0;

};

class input\_adapter

{

public:

// native support

JSON\_HEDLEY\_NON\_NULL(2)

input\_adapter(std::FILE\* file)

: ia(std::make\_shared<file\_input\_adapter>(file)) {}

/// input adapter for input stream

input\_adapter(std::istream& i)

: ia(std::make\_shared<input\_stream\_adapter>(i)) {}

/// input adapter for input stream

input\_adapter(std::istream&& i)

: ia(std::make\_shared<input\_stream\_adapter>(i)) {}

input\_adapter(const std::wstring& ws)

: ia(std::make\_shared<wide\_string\_input\_adapter<std::wstring>>(ws)) {}

input\_adapter(const std::u16string& ws)

: ia(std::make\_shared<wide\_string\_input\_adapter<std::u16string>>(ws)) {}

input\_adapter(const std::u32string& ws)

: ia(std::make\_shared<wide\_string\_input\_adapter<std::u32string>>(ws)) {}

/// input adapter for buffer

template<typename CharT,

typename std::enable\_if<

std::is\_pointer<CharT>::value and

std::is\_integral<typename std::remove\_pointer<CharT>::type>::value and

sizeof(typename std::remove\_pointer<CharT>::type) == 1,

int>::type = 0>

input\_adapter(CharT b, std::size\_t l)

: ia(std::make\_shared<input\_buffer\_adapter>(reinterpret\_cast<const char\*>(b), l)) {}

// derived support

/// input adapter for string literal

template<typename CharT,

typename std::enable\_if<

std::is\_pointer<CharT>::value and

std::is\_integral<typename std::remove\_pointer<CharT>::type>::value and

sizeof(typename std::remove\_pointer<CharT>::type) == 1,

int>::type = 0>

input\_adapter(CharT b)

: input\_adapter(reinterpret\_cast<const char\*>(b),

std::strlen(reinterpret\_cast<const char\*>(b))) {}

/// input adapter for iterator range with contiguous storage

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>

input\_adapter(IteratorType first, IteratorType last)

{

#ifndef NDEBUG

// assertion to check that the iterator range is indeed contiguous,

// see http://stackoverflow.com/a/35008842/266378 for more discussion

const auto is\_contiguous = std::accumulate(

first, last, std::pair<bool, int>(true, 0),

[&first](std::pair<bool, int> res, decltype(\*first) val)

{

res.first &= (val == \*(std::next(std::addressof(\*first), res.second++)));

return res;

}).first;

assert(is\_contiguous);

#endif

// assertion to check that each element is 1 byte long

static\_assert(

sizeof(typename iterator\_traits<IteratorType>::value\_type) == 1,

"each element in the iterator range must have the size of 1 byte");

const auto len = static\_cast<size\_t>(std::distance(first, last));

if (JSON\_HEDLEY\_LIKELY(len > 0))

{

// there is at least one element: use the address of first

ia = std::make\_shared<input\_buffer\_adapter>(reinterpret\_cast<const char\*>(&(\*first)), len);

}

else

{

// the address of first cannot be used: use nullptr

ia = std::make\_shared<input\_buffer\_adapter>(nullptr, len);

}

}

/// input adapter for array

template<class T, std::size\_t N>

input\_adapter(T (&array)[N])

: input\_adapter(std::begin(array), std::end(array)) {}

/// input adapter for contiguous container

template<class ContiguousContainer, typename

std::enable\_if<not std::is\_pointer<ContiguousContainer>::value and

std::is\_base\_of<std::random\_access\_iterator\_tag, typename iterator\_traits<decltype(std::begin(std::declval<ContiguousContainer const>()))>::iterator\_category>::value,

int>::type = 0>

input\_adapter(const ContiguousContainer& c)

: input\_adapter(std::begin(c), std::end(c)) {}

operator input\_adapter\_t()

{

return ia;

}

private:

/// the actual adapter

input\_adapter\_t ia = nullptr;

};

} // namespace detail

} // namespace nlohmann

// #include <nlohmann/detail/input/json\_sax.hpp>

#include <cassert> // assert

#include <cstddef>

#include <string> // string

#include <utility> // move

#include <vector> // vector

// #include <nlohmann/detail/exceptions.hpp>

// #include <nlohmann/detail/macro\_scope.hpp>

namespace nlohmann

{

/\*!

@brief SAX interface

This class describes the SAX interface used by @ref 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.

\*/

template<typename BasicJsonType>

struct json\_sax

{

/// type for (signed) integers

using number\_integer\_t = typename BasicJsonType::number\_integer\_t;

/// type for unsigned integers

using number\_unsigned\_t = typename BasicJsonType::number\_unsigned\_t;

/// type for floating-point numbers

using number\_float\_t = typename BasicJsonType::number\_float\_t;

/// type for strings

using string\_t = typename BasicJsonType::string\_t;

/\*!

@brief a null value was read

@return whether parsing should proceed

\*/

virtual bool null() = 0;

/\*!

@brief a boolean value was read

@param[in] val boolean value

@return whether parsing should proceed

\*/

virtual bool boolean(bool val) = 0;

/\*!

@brief an integer number was read

@param[in] val integer value

@return whether parsing should proceed

\*/

virtual bool number\_integer(number\_integer\_t val) = 0;

/\*!

@brief an unsigned integer number was read

@param[in] val unsigned integer value

@return whether parsing should proceed

\*/

virtual bool number\_unsigned(number\_unsigned\_t val) = 0;

/\*!

@brief an floating-point number was read

@param[in] val floating-point value

@param[in] s raw token value

@return whether parsing should proceed

\*/

virtual bool number\_float(number\_float\_t val, const string\_t& s) = 0;

/\*!

@brief a string was read

@param[in] val string value

@return whether parsing should proceed

@note It is safe to move the passed string.

\*/

virtual bool string(string\_t& val) = 0;

/\*!

@brief the beginning of an object was read

@param[in] elements number of object elements or -1 if unknown

@return whether parsing should proceed

@note binary formats may report the number of elements

\*/

virtual bool start\_object(std::size\_t elements) = 0;

/\*!

@brief an object key was read

@param[in] val object key

@return whether parsing should proceed

@note It is safe to move the passed string.

\*/

virtual bool key(string\_t& val) = 0;

/\*!

@brief the end of an object was read

@return whether parsing should proceed

\*/

virtual bool end\_object() = 0;

/\*!

@brief the beginning of an array was read

@param[in] elements number of array elements or -1 if unknown

@return whether parsing should proceed

@note binary formats may report the number of elements

\*/

virtual bool start\_array(std::size\_t elements) = 0;

/\*!

@brief the end of an array was read

@return whether parsing should proceed

\*/

virtual bool end\_array() = 0;

/\*!

@brief a parse error occurred

@param[in] position the position in the input where the error occurs

@param[in] last\_token the last read token

@param[in] ex an exception object describing the error

@return whether parsing should proceed (must return false)

\*/

virtual bool parse\_error(std::size\_t position,

const std::string& last\_token,

const detail::exception& ex) = 0;

virtual ~json\_sax() = default;

};

namespace detail

{

/\*!

@brief SAX implementation to create a JSON value from SAX events

This class implements the @ref 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.

@tparam BasicJsonType the JSON type

\*/

template<typename BasicJsonType>

class json\_sax\_dom\_parser

{

public:

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;

/\*!

@param[in, out] r reference to a JSON value that is manipulated while

parsing

@param[in] allow\_exceptions\_ whether parse errors yield exceptions

\*/

explicit json\_sax\_dom\_parser(BasicJsonType& r, const bool allow\_exceptions\_ = true)

: root(r), allow\_exceptions(allow\_exceptions\_)

{}

// make class move-only

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;

~json\_sax\_dom\_parser() = default;

bool null()

{

handle\_value(nullptr);

return true;

}

bool boolean(bool val)

{

handle\_value(val);

return true;

}

bool number\_integer(number\_integer\_t val)

{

handle\_value(val);

return true;

}

bool number\_unsigned(number\_unsigned\_t val)

{

handle\_value(val);

return true;

}

bool number\_float(number\_float\_t val, const string\_t& /\*unused\*/)

{

handle\_value(val);

return true;

}

bool string(string\_t& val)

{

handle\_value(val);

return true;

}

bool start\_object(std::size\_t len)

{

ref\_stack.push\_back(handle\_value(BasicJsonType::value\_t::object));

if (JSON\_HEDLEY\_UNLIKELY(len != std::size\_t(-1) and len > ref\_stack.back()->max\_size()))

{

JSON\_THROW(out\_of\_range::create(408,

"excessive object size: " + std::to\_string(len)));

}

return true;

}

bool key(string\_t& val)

{

// add null at given key and store the reference for later

object\_element = &(ref\_stack.back()->m\_value.object->operator[](val));

return true;

}

bool end\_object()

{

ref\_stack.pop\_back();

return true;

}

bool start\_array(std::size\_t len)

{

ref\_stack.push\_back(handle\_value(BasicJsonType::value\_t::array));

if (JSON\_HEDLEY\_UNLIKELY(len != std::size\_t(-1) and len > ref\_stack.back()->max\_size()))

{

JSON\_THROW(out\_of\_range::create(408,

"excessive array size: " + std::to\_string(len)));

}

return true;

}

bool end\_array()

{

ref\_stack.pop\_back();

return true;

}

bool parse\_error(std::size\_t /\*unused\*/, const std::string& /\*unused\*/,

const detail::exception& ex)

{

errored = true;

if (allow\_exceptions)

{

// determine the proper exception type from the id

switch ((ex.id / 100) % 100)

{

case 1:

JSON\_THROW(\*static\_cast<const detail::parse\_error\*>(&ex));

case 4:

JSON\_THROW(\*static\_cast<const detail::out\_of\_range\*>(&ex));

// LCOV\_EXCL\_START

case 2:

JSON\_THROW(\*static\_cast<const detail::invalid\_iterator\*>(&ex));

case 3:

JSON\_THROW(\*static\_cast<const detail::type\_error\*>(&ex));

case 5:

JSON\_THROW(\*static\_cast<const detail::other\_error\*>(&ex));

default:

assert(false);

// LCOV\_EXCL\_STOP

}

}

return false;

}

constexpr bool is\_errored() const

{

return errored;

}

private:

/\*!

@invariant If the ref stack is empty, then the passed value will be the new

root.

@invariant If the ref stack contains a value, then it is an array or an

object to which we can add elements

\*/

template<typename Value>

JSON\_HEDLEY\_RETURNS\_NON\_NULL

BasicJsonType\* handle\_value(Value&& v)

{

if (ref\_stack.empty())

{

root = BasicJsonType(std::forward<Value>(v));

return &root;

}

assert(ref\_stack.back()->is\_array() or ref\_stack.back()->is\_object());

if (ref\_stack.back()->is\_array())

{

ref\_stack.back()->m\_value.array->emplace\_back(std::forward<Value>(v));

return &(ref\_stack.back()->m\_value.array->back());

}

assert(ref\_stack.back()->is\_object());

assert(object\_element);

\*object\_element = BasicJsonType(std::forward<Value>(v));

return object\_element;

}

/// the parsed JSON value

BasicJsonType& root;

/// stack to model hierarchy of values

std::vector<BasicJsonType\*> ref\_stack {};

/// helper to hold the reference for the next object element

BasicJsonType\* object\_element = nullptr;

/// whether a syntax error occurred

bool errored = false;

/// whether to throw exceptions in case of errors

const bool allow\_exceptions = true;

};

template<typename BasicJsonType>

class json\_sax\_dom\_callback\_parser

{

public:

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 parser\_callback\_t = typename BasicJsonType::parser\_callback\_t;

using parse\_event\_t = typename BasicJsonType::parse\_event\_t;

json\_sax\_dom\_callback\_parser(BasicJsonType& r,

const parser\_callback\_t cb,

const bool allow\_exceptions\_ = true)

: root(r), callback(cb), allow\_exceptions(allow\_exceptions\_)

{

keep\_stack.push\_back(true);

}

// make class move-only

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;

~json\_sax\_dom\_callback\_parser() = default;

bool null()

{

handle\_value(nullptr);

return true;

}

bool boolean(bool val)

{

handle\_value(val);

return true;

}

bool number\_integer(number\_integer\_t val)

{

handle\_value(val);

return true;

}

bool number\_unsigned(number\_unsigned\_t val)

{

handle\_value(val);

return true;

}

bool number\_float(number\_float\_t val, const string\_t& /\*unused\*/)

{

handle\_value(val);

return true;

}

bool string(string\_t& val)

{

handle\_value(val);

return true;

}

bool start\_object(std::size\_t len)

{

// check callback for object start

const bool keep = callback(static\_cast<int>(ref\_stack.size()), parse\_event\_t::object\_start, discarded);

keep\_stack.push\_back(keep);

auto val = handle\_value(BasicJsonType::value\_t::object, true);

ref\_stack.push\_back(val.second);

// check object limit

if (ref\_stack.back() and JSON\_HEDLEY\_UNLIKELY(len != std::size\_t(-1) and len > ref\_stack.back()->max\_size()))

{

JSON\_THROW(out\_of\_range::create(408, "excessive object size: " + std::to\_string(len)));

}

return true;

}

bool key(string\_t& val)

{

BasicJsonType k = BasicJsonType(val);

// check callback for key

const bool keep = callback(static\_cast<int>(ref\_stack.size()), parse\_event\_t::key, k);

key\_keep\_stack.push\_back(keep);

// add discarded value at given key and store the reference for later

if (keep and ref\_stack.back())

{

object\_element = &(ref\_stack.back()->m\_value.object->operator[](val) = discarded);

}

return true;

}

bool end\_object()

{

if (ref\_stack.back() and not callback(static\_cast<int>(ref\_stack.size()) - 1, parse\_event\_t::object\_end, \*ref\_stack.back()))

{

// discard object

\*ref\_stack.back() = discarded;

}

assert(not ref\_stack.empty());

assert(not keep\_stack.empty());

ref\_stack.pop\_back();

keep\_stack.pop\_back();

if (not ref\_stack.empty() and ref\_stack.back() and ref\_stack.back()->is\_object())

{

// remove discarded value

for (auto it = ref\_stack.back()->begin(); it != ref\_stack.back()->end(); ++it)

{

if (it->is\_discarded())

{

ref\_stack.back()->erase(it);

break;

}

}

}

return true;

}

bool start\_array(std::size\_t len)

{

const bool keep = callback(static\_cast<int>(ref\_stack.size()), parse\_event\_t::array\_start, discarded);

keep\_stack.push\_back(keep);

auto val = handle\_value(BasicJsonType::value\_t::array, true);

ref\_stack.push\_back(val.second);

// check array limit

if (ref\_stack.back() and JSON\_HEDLEY\_UNLIKELY(len != std::size\_t(-1) and len > ref\_stack.back()->max\_size()))

{

JSON\_THROW(out\_of\_range::create(408, "excessive array size: " + std::to\_string(len)));

}

return true;

}

bool end\_array()

{

bool keep = true;

if (ref\_stack.back())

{

keep = callback(static\_cast<int>(ref\_stack.size()) - 1, parse\_event\_t::array\_end, \*ref\_stack.back());

if (not keep)

{

// discard array

\*ref\_stack.back() = discarded;

}

}

assert(not ref\_stack.empty());

assert(not keep\_stack.empty());

ref\_stack.pop\_back();

keep\_stack.pop\_back();

// remove discarded value

if (not keep and not ref\_stack.empty() and ref\_stack.back()->is\_array())

{

ref\_stack.back()->m\_value.array->pop\_back();

}

return true;

}

bool parse\_error(std::size\_t /\*unused\*/, const std::string& /\*unused\*/,

const detail::exception& ex)

{

errored = true;

if (allow\_exceptions)

{

// determine the proper exception type from the id

switch ((ex.id / 100) % 100)

{

case 1:

JSON\_THROW(\*static\_cast<const detail::parse\_error\*>(&ex));

case 4:

JSON\_THROW(\*static\_cast<const detail::out\_of\_range\*>(&ex));

// LCOV\_EXCL\_START

case 2:

JSON\_THROW(\*static\_cast<const detail::invalid\_iterator\*>(&ex));

case 3:

JSON\_THROW(\*static\_cast<const detail::type\_error\*>(&ex));

case 5:

JSON\_THROW(\*static\_cast<const detail::other\_error\*>(&ex));

default:

assert(false);

// LCOV\_EXCL\_STOP

}

}

return false;

}

constexpr bool is\_errored() const

{

return errored;

}

private:

/\*!

@param[in] v value to add to the JSON value we build during parsing

@param[in] skip\_callback whether we should skip calling the callback

function; this is required after start\_array() and

start\_object() SAX events, because otherwise we would call the

callback function with an empty array or object, respectively.

@invariant If the ref stack is empty, then the passed value will be the new

root.

@invariant If the ref stack contains a value, then it is an array or an

object to which we can add elements

@return pair of boolean (whether value should be kept) and pointer (to the

passed value in the ref\_stack hierarchy; nullptr if not kept)

\*/

template<typename Value>

std::pair<bool, BasicJsonType\*> handle\_value(Value&& v, const bool skip\_callback = false)

{

assert(not keep\_stack.empty());

// do not handle this value if we know it would be added to a discarded

// container

if (not keep\_stack.back())

{

return {false, nullptr};

}

// create value

auto value = BasicJsonType(std::forward<Value>(v));

// check callback

const bool keep = skip\_callback or callback(static\_cast<int>(ref\_stack.size()), parse\_event\_t::value, value);

// do not handle this value if we just learnt it shall be discarded

if (not keep)

{

return {false, nullptr};

}

if (ref\_stack.empty())

{

root = std::move(value);

return {true, &root};

}

// skip this value if we already decided to skip the parent

// (https://github.com/nlohmann/json/issues/971#issuecomment-413678360)

if (not ref\_stack.back())

{

return {false, nullptr};

}

// we now only expect arrays and objects

assert(ref\_stack.back()->is\_array() or ref\_stack.back()->is\_object());

// array

if (ref\_stack.back()->is\_array())

{

ref\_stack.back()->m\_value.array->push\_back(std::move(value));

return {true, &(ref\_stack.back()->m\_value.array->back())};

}

// object

assert(ref\_stack.back()->is\_object());

// check if we should store an element for the current key

assert(not key\_keep\_stack.empty());

const bool store\_element = key\_keep\_stack.back();

key\_keep\_stack.pop\_back();

if (not store\_element)

{

return {false, nullptr};

}

assert(object\_element);

\*object\_element = std::move(value);

return {true, object\_element};

}

/// the parsed JSON value

BasicJsonType& root;

/// stack to model hierarchy of values

std::vector<BasicJsonType\*> ref\_stack {};

/// stack to manage which values to keep

std::vector<bool> keep\_stack {};

/// stack to manage which object keys to keep

std::vector<bool> key\_keep\_stack {};

/// helper to hold the reference for the next object element

BasicJsonType\* object\_element = nullptr;

/// whether a syntax error occurred

bool errored = false;

/// callback function

const parser\_callback\_t callback = nullptr;

/// whether to throw exceptions in case of errors

const bool allow\_exceptions = true;

/// a discarded value for the callback

BasicJsonType discarded = BasicJsonType::value\_t::discarded;

};

template<typename BasicJsonType>

class json\_sax\_acceptor

{

public:

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;

bool null()

{

return true;

}

bool boolean(bool /\*unused\*/)

{

return true;

}

bool number\_integer(number\_integer\_t /\*unused\*/)

{

return true;

}

bool number\_unsigned(number\_unsigned\_t /\*unused\*/)

{

return true;

}

bool number\_float(number\_float\_t /\*unused\*/, const string\_t& /\*unused\*/)

{

return true;

}

bool string(string\_t& /\*unused\*/)

{

return true;

}

bool start\_object(std::size\_t /\*unused\*/ = std::size\_t(-1))

{

return true;

}

bool key(string\_t& /\*unused\*/)

{

return true;

}

bool end\_object()

{

return true;

}

bool start\_array(std::size\_t /\*unused\*/ = std::size\_t(-1))

{

return true;

}

bool end\_array()

{

return true;

}

bool parse\_error(std::size\_t /\*unused\*/, const std::string& /\*unused\*/, const detail::exception& /\*unused\*/)

{

return false;

}

};

} // namespace detail

} // namespace nlohmann

// #include <nlohmann/detail/macro\_scope.hpp>

// #include <nlohmann/detail/meta/is\_sax.hpp>

#include <cstdint> // size\_t

#include <utility> // declval

#include <string> // string

// #include <nlohmann/detail/meta/detected.hpp>

// #include <nlohmann/detail/meta/type\_traits.hpp>

namespace nlohmann

{

namespace detail

{

template <typename T>

using null\_function\_t = decltype(std::declval<T&>().null());

template <typename T>

using boolean\_function\_t =

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

template <typename T, typename Integer>

using number\_integer\_function\_t =

decltype(std::declval<T&>().number\_integer(std::declval<Integer>()));

template <typename T, typename Unsigned>

using number\_unsigned\_function\_t =

decltype(std::declval<T&>().number\_unsigned(std::declval<Unsigned>()));

template <typename T, typename Float, typename String>

using number\_float\_function\_t = decltype(std::declval<T&>().number\_float(

std::declval<Float>(), std::declval<const String&>()));

template <typename T, typename String>

using string\_function\_t =

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

template <typename T>

using start\_object\_function\_t =

decltype(std::declval<T&>().start\_object(std::declval<std::size\_t>()));

template <typename T, typename String>

using key\_function\_t =

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

template <typename T>

using end\_object\_function\_t = decltype(std::declval<T&>().end\_object());

template <typename T>

using start\_array\_function\_t =

decltype(std::declval<T&>().start\_array(std::declval<std::size\_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 = decltype(std::declval<T&>().parse\_error(

std::declval<std::size\_t>(), std::declval<const std::string&>(),

std::declval<const Exception&>()));

template <typename SAX, typename BasicJsonType>

struct is\_sax

{

private:

static\_assert(is\_basic\_json<BasicJsonType>::value,

"BasicJsonType must be of type basic\_json<...>");

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 exception\_t = typename BasicJsonType::exception;

public:

static constexpr bool 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, 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;

};

template <typename SAX, typename BasicJsonType>

struct is\_sax\_static\_asserts

{

private:

static\_assert(is\_basic\_json<BasicJsonType>::value,

"BasicJsonType must be of type basic\_json<...>");

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 exception\_t = typename BasicJsonType::exception;

public:

static\_assert(is\_detected\_exact<bool, null\_function\_t, SAX>::value,

"Missing/invalid function: bool null()");

static\_assert(is\_detected\_exact<bool, boolean\_function\_t, SAX>::value,

"Missing/invalid function: bool boolean(bool)");

static\_assert(is\_detected\_exact<bool, boolean\_function\_t, SAX>::value,

"Missing/invalid function: bool boolean(bool)");

static\_assert(

is\_detected\_exact<bool, number\_integer\_function\_t, SAX,

number\_integer\_t>::value,

"Missing/invalid function: bool number\_integer(number\_integer\_t)");

static\_assert(

is\_detected\_exact<bool, number\_unsigned\_function\_t, SAX,

number\_unsigned\_t>::value,

"Missing/invalid function: bool number\_unsigned(number\_unsigned\_t)");

static\_assert(is\_detected\_exact<bool, number\_float\_function\_t, SAX,

number\_float\_t, string\_t>::value,

"Missing/invalid function: bool number\_float(number\_float\_t, const string\_t&)");

static\_assert(

is\_detected\_exact<bool, string\_function\_t, SAX, string\_t>::value,

"Missing/invalid function: bool string(string\_t&)");

static\_assert(is\_detected\_exact<bool, start\_object\_function\_t, SAX>::value,

"Missing/invalid function: bool start\_object(std::size\_t)");

static\_assert(is\_detected\_exact<bool, key\_function\_t, SAX, string\_t>::value,

"Missing/invalid function: bool key(string\_t&)");

static\_assert(is\_detected\_exact<bool, end\_object\_function\_t, SAX>::value,

"Missing/invalid function: bool end\_object()");

static\_assert(is\_detected\_exact<bool, start\_array\_function\_t, SAX>::value,

"Missing/invalid function: bool start\_array(std::size\_t)");

static\_assert(is\_detected\_exact<bool, end\_array\_function\_t, SAX>::value,

"Missing/invalid function: bool end\_array()");

static\_assert(

is\_detected\_exact<bool, parse\_error\_function\_t, SAX, exception\_t>::value,

"Missing/invalid function: bool parse\_error(std::size\_t, const "

"std::string&, const exception&)");

};

} // namespace detail

} // namespace nlohmann

// #include <nlohmann/detail/value\_t.hpp>

namespace nlohmann

{

namespace detail

{

///////////////////

// binary reader //

///////////////////

/\*!

@brief deserialization of CBOR, MessagePack, and UBJSON values

\*/

template<typename BasicJsonType, typename SAX = json\_sax\_dom\_parser<BasicJsonType>>

class binary\_reader

{

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 json\_sax\_t = SAX;

public:

/\*!

@brief create a binary reader

@param[in] adapter input adapter to read from

\*/

explicit binary\_reader(input\_adapter\_t adapter) : ia(std::move(adapter))

{

(void)detail::is\_sax\_static\_asserts<SAX, BasicJsonType> {};

assert(ia);

}

// make class move-only

binary\_reader(const binary\_reader&) = delete;

binary\_reader(binary\_reader&&) = default;

binary\_reader& operator=(const binary\_reader&) = delete;

binary\_reader& operator=(binary\_reader&&) = default;

~binary\_reader() = default;

/\*!

@param[in] format the binary format to parse

@param[in] sax\_ a SAX event processor

@param[in] strict whether to expect the input to be consumed completed

@return

\*/

JSON\_HEDLEY\_NON\_NULL(3)

bool sax\_parse(const input\_format\_t format,

json\_sax\_t\* sax\_,

const bool strict = true)

{

sax = sax\_;

bool result = false;

switch (format)

{

case input\_format\_t::bson:

result = parse\_bson\_internal();

break;

case input\_format\_t::cbor:

result = parse\_cbor\_internal();

break;

case input\_format\_t::msgpack:

result = parse\_msgpack\_internal();

break;

case input\_format\_t::ubjson:

result = parse\_ubjson\_internal();

break;

default: // LCOV\_EXCL\_LINE

assert(false); // LCOV\_EXCL\_LINE

}

// strict mode: next byte must be EOF

if (result and strict)

{

if (format == input\_format\_t::ubjson)

{

get\_ignore\_noop();

}

else

{

get();

}

if (JSON\_HEDLEY\_UNLIKELY(current != std::char\_traits<char>::eof()))

{

return sax->parse\_error(chars\_read, get\_token\_string(),

parse\_error::create(110, chars\_read, exception\_message(format, "expected end of input; last byte: 0x" + get\_token\_string(), "value")));

}

}

return result;

}

/\*!

@brief determine system byte order

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

@note from http://stackoverflow.com/a/1001328/266378

\*/

static constexpr bool little\_endianess(int num = 1) noexcept

{

return \*reinterpret\_cast<char\*>(&num) == 1;

}

private:

//////////

// BSON //

//////////

/\*!

@brief Reads in a BSON-object and passes it to the SAX-parser.

@return whether a valid BSON-value was passed to the SAX parser

\*/

bool parse\_bson\_internal()

{

std::int32\_t document\_size;

get\_number<std::int32\_t, true>(input\_format\_t::bson, document\_size);

if (JSON\_HEDLEY\_UNLIKELY(not sax->start\_object(std::size\_t(-1))))

{

return false;

}

if (JSON\_HEDLEY\_UNLIKELY(not parse\_bson\_element\_list(/\*is\_array\*/false)))

{

return false;

}

return sax->end\_object();

}

/\*!

@brief Parses a C-style string from the BSON input.

@param[in, out] result A reference to the string variable where the read

string is to be stored.

@return `true` if the \x00-byte indicating the end of the string was

encountered before the EOF; false` indicates an unexpected EOF.

\*/

bool get\_bson\_cstr(string\_t& result)

{

auto out = std::back\_inserter(result);

while (true)

{

get();

if (JSON\_HEDLEY\_UNLIKELY(not unexpect\_eof(input\_format\_t::bson, "cstring")))

{

return false;

}

if (current == 0x00)

{

return true;

}

\*out++ = static\_cast<char>(current);

}

return true;

}

/\*!

@brief Parses a zero-terminated string of length @a len from the BSON

input.

@param[in] len The length (including the zero-byte at the end) of the

string to be read.

@param[in, out] result A reference to the string variable where the read

string is to be stored.

@tparam NumberType The type of the length @a len

@pre len >= 1

@return `true` if the string was successfully parsed

\*/

template<typename NumberType>

bool get\_bson\_string(const NumberType len, string\_t& result)

{

if (JSON\_HEDLEY\_UNLIKELY(len < 1))

{

auto last\_token = get\_token\_string();

return sax->parse\_error(chars\_read, last\_token, parse\_error::create(112, chars\_read, exception\_message(input\_format\_t::bson, "string length must be at least 1, is " + std::to\_string(len), "string")));

}

return get\_string(input\_format\_t::bson, len - static\_cast<NumberType>(1), result) and get() != std::char\_traits<char>::eof();

}

/\*!

@brief Read a BSON document element of the given @a element\_type.

@param[in] element\_type The BSON element type, c.f. http://bsonspec.org/spec.html

@param[in] element\_type\_parse\_position The position in the input stream,

where the `element\_type` was read.

@warning Not all BSON element types are supported yet. An unsupported

@a element\_type will give rise to a parse\_error.114:

Unsupported BSON record type 0x...

@return whether a valid BSON-object/array was passed to the SAX parser

\*/

bool parse\_bson\_element\_internal(const int element\_type,

const std::size\_t element\_type\_parse\_position)

{

switch (element\_type)

{

case 0x01: // double

{

double number;

return get\_number<double, true>(input\_format\_t::bson, number) and sax->number\_float(static\_cast<number\_float\_t>(number), "");

}

case 0x02: // string

{

std::int32\_t len;

string\_t value;

return get\_number<std::int32\_t, true>(input\_format\_t::bson, len) and get\_bson\_string(len, value) and sax->string(value);

}

case 0x03: // object

{

return parse\_bson\_internal();

}

case 0x04: // array

{

return parse\_bson\_array();

}

case 0x08: // boolean

{

return sax->boolean(get() != 0);

}

case 0x0A: // null

{

return sax->null();

}

case 0x10: // int32

{

std::int32\_t value;

return get\_number<std::int32\_t, true>(input\_format\_t::bson, value) and sax->number\_integer(value);

}

case 0x12: // int64

{

std::int64\_t value;

return get\_number<std::int64\_t, true>(input\_format\_t::bson, value) and sax->number\_integer(value);

}

default: // anything else not supported (yet)

{

std::array<char, 3> cr{{}};

(std::snprintf)(cr.data(), cr.size(), "%.2hhX", static\_cast<unsigned char>(element\_type));

return sax->parse\_error(element\_type\_parse\_position, std::string(cr.data()), parse\_error::create(114, element\_type\_parse\_position, "Unsupported BSON record type 0x" + std::string(cr.data())));

}

}

}

/\*!

@brief Read a BSON element list (as specified in the BSON-spec)

The same binary layout is used for objects and arrays, hence it must be

indicated with the argument @a is\_array which one is expected

(true --> array, false --> object).

@param[in] is\_array Determines if the element list being read is to be

treated as an object (@a is\_array == false), or as an

array (@a is\_array == true).

@return whether a valid BSON-object/array was passed to the SAX parser

\*/

bool parse\_bson\_element\_list(const bool is\_array)

{

string\_t key;

while (int element\_type = get())

{

if (JSON\_HEDLEY\_UNLIKELY(not unexpect\_eof(input\_format\_t::bson, "element list")))

{

return false;

}

const std::size\_t element\_type\_parse\_position = chars\_read;

if (JSON\_HEDLEY\_UNLIKELY(not get\_bson\_cstr(key)))

{

return false;

}

if (not is\_array and not sax->key(key))

{

return false;

}

if (JSON\_HEDLEY\_UNLIKELY(not parse\_bson\_element\_internal(element\_type, element\_type\_parse\_position)))

{

return false;

}

// get\_bson\_cstr only appends

key.clear();

}

return true;

}

/\*!

@brief Reads an array from the BSON input and passes it to the SAX-parser.

@return whether a valid BSON-array was passed to the SAX parser

\*/

bool parse\_bson\_array()

{

std::int32\_t document\_size;

get\_number<std::int32\_t, true>(input\_format\_t::bson, document\_size);

if (JSON\_HEDLEY\_UNLIKELY(not sax->start\_array(std::size\_t(-1))))

{

return false;

}

if (JSON\_HEDLEY\_UNLIKELY(not parse\_bson\_element\_list(/\*is\_array\*/true)))

{

return false;

}

return sax->end\_array();

}

//////////

// CBOR //

//////////

/\*!

@param[in] get\_char whether a new character should be retrieved from the

input (true, default) or whether the last read

character should be considered instead

@return whether a valid CBOR value was passed to the SAX parser

\*/

bool parse\_cbor\_internal(const bool get\_char = true)

{

switch (get\_char ? get() : current)

{

// EOF

case std::char\_traits<char>::eof():

return unexpect\_eof(input\_format\_t::cbor, "value");

// Integer 0x00..0x17 (0..23)

case 0x00:

case 0x01:

case 0x02:

case 0x03:

case 0x04:

case 0x05:

case 0x06:

case 0x07:

case 0x08:

case 0x09:

case 0x0A:

case 0x0B:

case 0x0C:

case 0x0D:

case 0x0E:

case 0x0F:

case 0x10:

case 0x11:

case 0x12:

case 0x13:

case 0x14:

case 0x15:

case 0x16:

case 0x17:

return sax->number\_unsigned(static\_cast<number\_unsigned\_t>(current));

case 0x18: // Unsigned integer (one-byte uint8\_t follows)

{

std::uint8\_t number;

return get\_number(input\_format\_t::cbor, number) and sax->number\_unsigned(number);

}

case 0x19: // Unsigned integer (two-byte uint16\_t follows)

{

std::uint16\_t number;

return get\_number(input\_format\_t::cbor, number) and sax->number\_unsigned(number);

}

case 0x1A: // Unsigned integer (four-byte uint32\_t follows)

{

std::uint32\_t number;

return get\_number(input\_format\_t::cbor, number) and sax->number\_unsigned(number);

}

case 0x1B: // Unsigned integer (eight-byte uint64\_t follows)

{

std::uint64\_t number;

return get\_number(input\_format\_t::cbor, number) and sax->number\_unsigned(number);

}

// Negative integer -1-0x00..-1-0x17 (-1..-24)

case 0x20:

case 0x21:

case 0x22:

case 0x23:

case 0x24:

case 0x25:

case 0x26:

case 0x27:

case 0x28:

case 0x29:

case 0x2A:

case 0x2B:

case 0x2C:

case 0x2D:

case 0x2E:

case 0x2F:

case 0x30:

case 0x31:

case 0x32:

case 0x33:

case 0x34:

case 0x35:

case 0x36:

case 0x37:

return sax->number\_integer(static\_cast<std::int8\_t>(0x20 - 1 - current));

case 0x38: // Negative integer (one-byte uint8\_t follows)

{

std::uint8\_t number;

return get\_number(input\_format\_t::cbor, number) and sax->number\_integer(static\_cast<number\_integer\_t>(-1) - number);

}

case 0x39: // Negative integer -1-n (two-byte uint16\_t follows)

{

std::uint16\_t number;

return get\_number(input\_format\_t::cbor, number) and sax->number\_integer(static\_cast<number\_integer\_t>(-1) - number);

}

case 0x3A: // Negative integer -1-n (four-byte uint32\_t follows)

{

std::uint32\_t number;

return get\_number(input\_format\_t::cbor, number) and sax->number\_integer(static\_cast<number\_integer\_t>(-1) - number);

}

case 0x3B: // Negative integer -1-n (eight-byte uint64\_t follows)

{

std::uint64\_t number;

return get\_number(input\_format\_t::cbor, number) and sax->number\_integer(static\_cast<number\_integer\_t>(-1)

- static\_cast<number\_integer\_t>(number));

}

// UTF-8 string (0x00..0x17 bytes follow)

case 0x60:

case 0x61:

case 0x62:

case 0x63:

case 0x64:

case 0x65:

case 0x66:

case 0x67:

case 0x68:

case 0x69:

case 0x6A:

case 0x6B:

case 0x6C:

case 0x6D:

case 0x6E:

case 0x6F:

case 0x70:

case 0x71:

case 0x72:

case 0x73:

case 0x74:

case 0x75:

case 0x76:

case 0x77:

case 0x78: // UTF-8 string (one-byte uint8\_t for n follows)

case 0x79: // UTF-8 string (two-byte uint16\_t for n follow)

case 0x7A: // UTF-8 string (four-byte uint32\_t for n follow)

case 0x7B: // UTF-8 string (eight-byte uint64\_t for n follow)

case 0x7F: // UTF-8 string (indefinite length)

{

string\_t s;

return get\_cbor\_string(s) and sax->string(s);

}

// array (0x00..0x17 data items follow)

case 0x80:

case 0x81:

case 0x82:

case 0x83:

case 0x84:

case 0x85:

case 0x86:

case 0x87:

case 0x88:

case 0x89:

case 0x8A:

case 0x8B:

case 0x8C:

case 0x8D:

case 0x8E:

case 0x8F:

case 0x90:

case 0x91:

case 0x92:

case 0x93:

case 0x94:

case 0x95:

case 0x96:

case 0x97:

return get\_cbor\_array(static\_cast<std::size\_t>(static\_cast<unsigned int>(current) & 0x1Fu));

case 0x98: // array (one-byte uint8\_t for n follows)

{

std::uint8\_t len;

return get\_number(input\_format\_t::cbor, len) and get\_cbor\_array(static\_cast<std::size\_t>(len));

}

case 0x99: // array (two-byte uint16\_t for n follow)

{

std::uint16\_t len;

return get\_number(input\_format\_t::cbor, len) and get\_cbor\_array(static\_cast<std::size\_t>(len));

}

case 0x9A: // array (four-byte uint32\_t for n follow)

{

std::uint32\_t len;

return get\_number(input\_format\_t::cbor, len) and get\_cbor\_array(static\_cast<std::size\_t>(len));

}

case 0x9B: // array (eight-byte uint64\_t for n follow)

{

std::uint64\_t len;

return get\_number(input\_format\_t::cbor, len) and get\_cbor\_array(static\_cast<std::size\_t>(len));

}

case 0x9F: // array (indefinite length)

return get\_cbor\_array(std::size\_t(-1));

// map (0x00..0x17 pairs of data items follow)

case 0xA0:

case 0xA1:

case 0xA2:

case 0xA3:

case 0xA4:

case 0xA5:

case 0xA6:

case 0xA7:

case 0xA8:

case 0xA9:

case 0xAA:

case 0xAB:

case 0xAC:

case 0xAD:

case 0xAE:

case 0xAF:

case 0xB0:

case 0xB1:

case 0xB2:

case 0xB3:

case 0xB4:

case 0xB5:

case 0xB6:

case 0xB7:

return get\_cbor\_object(static\_cast<std::size\_t>(static\_cast<unsigned int>(current) & 0x1Fu));

case 0xB8: // map (one-byte uint8\_t for n follows)

{

std::uint8\_t len;

return get\_number(input\_format\_t::cbor, len) and get\_cbor\_object(static\_cast<std::size\_t>(len));

}

case 0xB9: // map (two-byte uint16\_t for n follow)

{

std::uint16\_t len;

return get\_number(input\_format\_t::cbor, len) and get\_cbor\_object(static\_cast<std::size\_t>(len));

}

case 0xBA: // map (four-byte uint32\_t for n follow)

{

std::uint32\_t len;

return get\_number(input\_format\_t::cbor, len) and get\_cbor\_object(static\_cast<std::size\_t>(len));

}

case 0xBB: // map (eight-byte uint64\_t for n follow)

{

std::uint64\_t len;

return get\_number(input\_format\_t::cbor, len) and get\_cbor\_object(static\_cast<std::size\_t>(len));

}

case 0xBF: // map (indefinite length)

return get\_cbor\_object(std::size\_t(-1));

case 0xF4: // false

return sax->boolean(false);

case 0xF5: // true

return sax->boolean(true);

case 0xF6: // null

return sax->null();

case 0xF9: // Half-Precision Float (two-byte IEEE 754)

{

const int byte1\_raw = get();

if (JSON\_HEDLEY\_UNLIKELY(not unexpect\_eof(input\_format\_t::cbor, "number")))

{

return false;

}

const int byte2\_raw = get();

if (JSON\_HEDLEY\_UNLIKELY(not unexpect\_eof(input\_format\_t::cbor, "number")))

{

return false;

}

const auto byte1 = static\_cast<unsigned char>(byte1\_raw);

const auto byte2 = static\_cast<unsigned char>(byte2\_raw);

// code from RFC 7049, Appendix D, Figure 3:

// As half-precision floating-point numbers were only added

// to IEEE 754 in 2008, today's programming platforms often

// still only have limited support for them. It is very

// easy to include at least decoding support for them even

// without such support. An example of a small decoder for

// half-precision floating-point numbers in the C language

// is shown in Fig. 3.

const auto half = static\_cast<unsigned int>((byte1 << 8u) + byte2);

const double val = [&half]

{

const int exp = (half >> 10u) & 0x1Fu;

const unsigned int mant = half & 0x3FFu;

assert(0 <= exp and exp <= 32);

assert(mant <= 1024);

switch (exp)

{

case 0:

return std::ldexp(mant, -24);

case 31:

return (mant == 0)

? std::numeric\_limits<double>::infinity()

: std::numeric\_limits<double>::quiet\_NaN();

default:

return std::ldexp(mant + 1024, exp - 25);

}

}();

return sax->number\_float((half & 0x8000u) != 0

? static\_cast<number\_float\_t>(-val)

: static\_cast<number\_float\_t>(val), "");

}

case 0xFA: // Single-Precision Float (four-byte IEEE 754)

{

float number;

return get\_number(input\_format\_t::cbor, number) and sax->number\_float(static\_cast<number\_float\_t>(number), "");

}

case 0xFB: // Double-Precision Float (eight-byte IEEE 754)

{

double number;

return get\_number(input\_format\_t::cbor, number) and sax->number\_float(static\_cast<number\_float\_t>(number), "");

}

default: // anything else (0xFF is handled inside the other types)

{

auto last\_token = get\_token\_string();

return sax->parse\_error(chars\_read, last\_token, parse\_error::create(112, chars\_read, exception\_message(input\_format\_t::cbor, "invalid byte: 0x" + last\_token, "value")));

}

}

}

/\*!

@brief reads a CBOR string

This function first reads starting bytes to determine the expected

string length and then copies this number of bytes into a string.

Additionally, CBOR's strings with indefinite lengths are supported.

@param[out] result created string

@return whether string creation completed

\*/

bool get\_cbor\_string(string\_t& result)

{

if (JSON\_HEDLEY\_UNLIKELY(not unexpect\_eof(input\_format\_t::cbor, "string")))

{

return false;

}

switch (current)

{

// UTF-8 string (0x00..0x17 bytes follow)

case 0x60:

case 0x61:

case 0x62:

case 0x63:

case 0x64:

case 0x65:

case 0x66:

case 0x67:

case 0x68:

case 0x69:

case 0x6A:

case 0x6B:

case 0x6C:

case 0x6D:

case 0x6E:

case 0x6F:

case 0x70:

case 0x71:

case 0x72:

case 0x73:

case 0x74:

case 0x75:

case 0x76:

case 0x77:

{

return get\_string(input\_format\_t::cbor, static\_cast<unsigned int>(current) & 0x1Fu, result);

}

case 0x78: // UTF-8 string (one-byte uint8\_t for n follows)

{

std::uint8\_t len;

return get\_number(input\_format\_t::cbor, len) and get\_string(input\_format\_t::cbor, len, result);

}

case 0x79: // UTF-8 string (two-byte uint16\_t for n follow)

{

std::uint16\_t len;

return get\_number(input\_format\_t::cbor, len) and get\_string(input\_format\_t::cbor, len, result);

}

case 0x7A: // UTF-8 string (four-byte uint32\_t for n follow)

{

std::uint32\_t len;

return get\_number(input\_format\_t::cbor, len) and get\_string(input\_format\_t::cbor, len, result);

}

case 0x7B: // UTF-8 string (eight-byte uint64\_t for n follow)

{

std::uint64\_t len;

return get\_number(input\_format\_t::cbor, len) and get\_string(input\_format\_t::cbor, len, result);

}

case 0x7F: // UTF-8 string (indefinite length)

{

while (get() != 0xFF)

{

string\_t chunk;

if (not get\_cbor\_string(chunk))

{

return false;

}

result.append(chunk);

}

return true;

}

default:

{

auto last\_token = get\_token\_string();

return sax->parse\_error(chars\_read, last\_token, parse\_error::create(113, chars\_read, exception\_message(input\_format\_t::cbor, "expected length specification (0x60-0x7B) or indefinite string type (0x7F); last byte: 0x" + last\_token, "string")));

}

}

}

/\*!

@param[in] len the length of the array or std::size\_t(-1) for an

array of indefinite size

@return whether array creation completed

\*/

bool get\_cbor\_array(const std::size\_t len)

{

if (JSON\_HEDLEY\_UNLIKELY(not sax->start\_array(len)))

{

return false;

}

if (len != std::size\_t(-1))

{

for (std::size\_t i = 0; i < len; ++i)

{

if (JSON\_HEDLEY\_UNLIKELY(not parse\_cbor\_internal()))

{

return false;

}

}

}

else

{

while (get() != 0xFF)

{

if (JSON\_HEDLEY\_UNLIKELY(not parse\_cbor\_internal(false)))

{

return false;

}

}

}

return sax->end\_array();

}

/\*!

@param[in] len the length of the object or std::size\_t(-1) for an

object of indefinite size

@return whether object creation completed

\*/

bool get\_cbor\_object(const std::size\_t len)

{

if (JSON\_HEDLEY\_UNLIKELY(not sax->start\_object(len)))

{

return false;

}

string\_t key;

if (len != std::size\_t(-1))

{

for (std::size\_t i = 0; i < len; ++i)

{

get();

if (JSON\_HEDLEY\_UNLIKELY(not get\_cbor\_string(key) or not sax->key(key)))

{

return false;

}

if (JSON\_HEDLEY\_UNLIKELY(not parse\_cbor\_internal()))

{

return false;

}

key.clear();

}

}

else

{

while (get() != 0xFF)

{

if (JSON\_HEDLEY\_UNLIKELY(not get\_cbor\_string(key) or not sax->key(key)))

{

return false;

}

if (JSON\_HEDLEY\_UNLIKELY(not parse\_cbor\_internal()))

{

return false;

}

key.clear();

}

}

return sax->end\_object();

}

/////////////

// MsgPack //

/////////////

/\*!

@return whether a valid MessagePack value was passed to the SAX parser

\*/

bool parse\_msgpack\_internal()

{

switch (get())

{

// EOF

case std::char\_traits<char>::eof():

return unexpect\_eof(input\_format\_t::msgpack, "value");

// positive fixint

case 0x00:

case 0x01:

case 0x02:

case 0x03:

case 0x04:

case 0x05:

case 0x06:

case 0x07:

case 0x08:

case 0x09:

case 0x0A:

case 0x0B:

case 0x0C:

case 0x0D:

case 0x0E:

case 0x0F:

case 0x10:

case 0x11:

case 0x12:

case 0x13:

case 0x14:

case 0x15:

case 0x16:

case 0x17:

case 0x18:

case 0x19:

case 0x1A:

case 0x1B:

case 0x1C:

case 0x1D:

case 0x1E:

case 0x1F:

case 0x20:

case 0x21:

case 0x22:

case 0x23:

case 0x24:

case 0x25:

case 0x26:

case 0x27:

case 0x28:

case 0x29:

case 0x2A:

case 0x2B:

case 0x2C:

case 0x2D:

case 0x2E:

case 0x2F:

case 0x30:

case 0x31:

case 0x32:

case 0x33:

case 0x34:

case 0x35:

case 0x36:

case 0x37:

case 0x38:

case 0x39:

case 0x3A:

case 0x3B:

case 0x3C:

case 0x3D:

case 0x3E:

case 0x3F:

case 0x40:

case 0x41:

case 0x42:

case 0x43:

case 0x44:

case 0x45:

case 0x46:

case 0x47:

case 0x48:

case 0x49:

case 0x4A:

case 0x4B:

case 0x4C:

case 0x4D:

case 0x4E:

case 0x4F:

case 0x50:

case 0x51:

case 0x52:

case 0x53:

case 0x54:

case 0x55:

case 0x56:

case 0x57:

case 0x58:

case 0x59:

case 0x5A:

case 0x5B:

case 0x5C:

case 0x5D:

case 0x5E:

case 0x5F:

case 0x60:

case 0x61:

case 0x62:

case 0x63:

case 0x64:

case 0x65:

case 0x66:

case 0x67:

case 0x68:

case 0x69:

case 0x6A:

case 0x6B:

case 0x6C:

case 0x6D:

case 0x6E:

case 0x6F:

case 0x70:

case 0x71:

case 0x72:

case 0x73:

case 0x74:

case 0x75:

case 0x76:

case 0x77:

case 0x78:

case 0x79:

case 0x7A:

case 0x7B:

case 0x7C:

case 0x7D:

case 0x7E:

case 0x7F:

return sax->number\_unsigned(static\_cast<number\_unsigned\_t>(current));

// fixmap

case 0x80:

case 0x81:

case 0x82:

case 0x83:

case 0x84:

case 0x85:

case 0x86:

case 0x87:

case 0x88:

case 0x89:

case 0x8A:

case 0x8B:

case 0x8C:

case 0x8D:

case 0x8E:

case 0x8F:

return get\_msgpack\_object(static\_cast<std::size\_t>(static\_cast<unsigned int>(current) & 0x0Fu));

// fixarray

case 0x90:

case 0x91:

case 0x92:

case 0x93:

case 0x94:

case 0x95:

case 0x96:

case 0x97:

case 0x98:

case 0x99:

case 0x9A:

case 0x9B:

case 0x9C:

case 0x9D:

case 0x9E:

case 0x9F:

return get\_msgpack\_array(static\_cast<std::size\_t>(static\_cast<unsigned int>(current) & 0x0Fu));

// fixstr

case 0xA0:

case 0xA1:

case 0xA2:

case 0xA3:

case 0xA4:

case 0xA5:

case 0xA6:

case 0xA7:

case 0xA8:

case 0xA9:

case 0xAA:

case 0xAB:

case 0xAC:

case 0xAD:

case 0xAE:

case 0xAF:

case 0xB0:

case 0xB1:

case 0xB2:

case 0xB3:

case 0xB4:

case 0xB5:

case 0xB6:

case 0xB7:

case 0xB8:

case 0xB9:

case 0xBA:

case 0xBB:

case 0xBC:

case 0xBD:

case 0xBE:

case 0xBF:

case 0xD9: // str 8

case 0xDA: // str 16

case 0xDB: // str 32

{

string\_t s;

return get\_msgpack\_string(s) and sax->string(s);

}

case 0xC0: // nil

return sax->null();

case 0xC2: // false

return sax->boolean(false);

case 0xC3: // true

return sax->boolean(true);

case 0xCA: // float 32

{

float number;

return get\_number(input\_format\_t::msgpack, number) and sax->number\_float(static\_cast<number\_float\_t>(number), "");

}

case 0xCB: // float 64

{

double number;

return get\_number(input\_format\_t::msgpack, number) and sax->number\_float(static\_cast<number\_float\_t>(number), "");

}

case 0xCC: // uint 8

{

std::uint8\_t number;

return get\_number(input\_format\_t::msgpack, number) and sax->number\_unsigned(number);

}

case 0xCD: // uint 16

{

std::uint16\_t number;

return get\_number(input\_format\_t::msgpack, number) and sax->number\_unsigned(number);

}

case 0xCE: // uint 32

{

std::uint32\_t number;

return get\_number(input\_format\_t::msgpack, number) and sax->number\_unsigned(number);

}

case 0xCF: // uint 64

{

std::uint64\_t number;

return get\_number(input\_format\_t::msgpack, number) and sax->number\_unsigned(number);

}

case 0xD0: // int 8

{

std::int8\_t number;

return get\_number(input\_format\_t::msgpack, number) and sax->number\_integer(number);

}

case 0xD1: // int 16

{

std::int16\_t number;

return get\_number(input\_format\_t::msgpack, number) and sax->number\_integer(number);

}

case 0xD2: // int 32

{

std::int32\_t number;

return get\_number(input\_format\_t::msgpack, number) and sax->number\_integer(number);

}

case 0xD3: // int 64

{

std::int64\_t number;

return get\_number(input\_format\_t::msgpack, number) and sax->number\_integer(number);

}

case 0xDC: // array 16

{

std::uint16\_t len;

return get\_number(input\_format\_t::msgpack, len) and get\_msgpack\_array(static\_cast<std::size\_t>(len));

}

case 0xDD: // array 32

{

std::uint32\_t len;

return get\_number(input\_format\_t::msgpack, len) and get\_msgpack\_array(static\_cast<std::size\_t>(len));

}

case 0xDE: // map 16

{

std::uint16\_t len;

return get\_number(input\_format\_t::msgpack, len) and get\_msgpack\_object(static\_cast<std::size\_t>(len));

}

case 0xDF: // map 32

{

std::uint32\_t len;

return get\_number(input\_format\_t::msgpack, len) and get\_msgpack\_object(static\_cast<std::size\_t>(len));

}

// negative fixint

case 0xE0:

case 0xE1:

case 0xE2:

case 0xE3:

case 0xE4:

case 0xE5:

case 0xE6:

case 0xE7:

case 0xE8:

case 0xE9:

case 0xEA:

case 0xEB:

case 0xEC:

case 0xED:

case 0xEE:

case 0xEF:

case 0xF0:

case 0xF1:

case 0xF2:

case 0xF3:

case 0xF4:

case 0xF5:

case 0xF6:

case 0xF7:

case 0xF8:

case 0xF9:

case 0xFA:

case 0xFB:

case 0xFC:

case 0xFD:

case 0xFE:

case 0xFF:

return sax->number\_integer(static\_cast<std::int8\_t>(current));

default: // anything else

{

auto last\_token = get\_token\_string();

return sax->parse\_error(chars\_read, last\_token, parse\_error::create(112, chars\_read, exception\_message(input\_format\_t::msgpack, "invalid byte: 0x" + last\_token, "value")));

}

}

}

/\*!

@brief reads a MessagePack string

This function first reads starting bytes to determine the expected

string length and then copies this number of bytes into a string.

@param[out] result created string

@return whether string creation completed

\*/

bool get\_msgpack\_string(string\_t& result)

{

if (JSON\_HEDLEY\_UNLIKELY(not unexpect\_eof(input\_format\_t::msgpack, "string")))

{

return false;

}

switch (current)

{

// fixstr

case 0xA0:

case 0xA1:

case 0xA2:

case 0xA3:

case 0xA4:

case 0xA5:

case 0xA6:

case 0xA7:

case 0xA8:

case 0xA9:

case 0xAA:

case 0xAB:

case 0xAC:

case 0xAD:

case 0xAE:

case 0xAF:

case 0xB0:

case 0xB1:

case 0xB2:

case 0xB3:

case 0xB4:

case 0xB5:

case 0xB6:

case 0xB7:

case 0xB8:

case 0xB9:

case 0xBA:

case 0xBB:

case 0xBC:

case 0xBD:

case 0xBE:

case 0xBF:

{

return get\_string(input\_format\_t::msgpack, static\_cast<unsigned int>(current) & 0x1Fu, result);

}

case 0xD9: // str 8

{

std::uint8\_t len;

return get\_number(input\_format\_t::msgpack, len) and get\_string(input\_format\_t::msgpack, len, result);

}

case 0xDA: // str 16

{

std::uint16\_t len;

return get\_number(input\_format\_t::msgpack, len) and get\_string(input\_format\_t::msgpack, len, result);

}

case 0xDB: // str 32

{

std::uint32\_t len;

return get\_number(input\_format\_t::msgpack, len) and get\_string(input\_format\_t::msgpack, len, result);

}

default:

{

auto last\_token = get\_token\_string();

return sax->parse\_error(chars\_read, last\_token, parse\_error::create(113, chars\_read, exception\_message(input\_format\_t::msgpack, "expected length specification (0xA0-0xBF, 0xD9-0xDB); last byte: 0x" + last\_token, "string")));

}

}

}

/\*!

@param[in] len the length of the array

@return whether array creation completed

\*/

bool get\_msgpack\_array(const std::size\_t len)

{

if (JSON\_HEDLEY\_UNLIKELY(not sax->start\_array(len)))

{

return false;

}

for (std::size\_t i = 0; i < len; ++i)

{

if (JSON\_HEDLEY\_UNLIKELY(not parse\_msgpack\_internal()))

{

return false;

}

}

return sax->end\_array();

}

/\*!

@param[in] len the length of the object

@return whether object creation completed

\*/

bool get\_msgpack\_object(const std::size\_t len)

{

if (JSON\_HEDLEY\_UNLIKELY(not sax->start\_object(len)))

{

return false;

}

string\_t key;

for (std::size\_t i = 0; i < len; ++i)

{

get();

if (JSON\_HEDLEY\_UNLIKELY(not get\_msgpack\_string(key) or not sax->key(key)))

{

return false;

}

if (JSON\_HEDLEY\_UNLIKELY(not parse\_msgpack\_internal()))

{

return false;

}

key.clear();

}

return sax->end\_object();

}

////////////

// UBJSON //

////////////

/\*!

@param[in] get\_char whether a new character should be retrieved from the

input (true, default) or whether the last read

character should be considered instead

@return whether a valid UBJSON value was passed to the SAX parser

\*/

bool parse\_ubjson\_internal(const bool get\_char = true)

{

return get\_ubjson\_value(get\_char ? get\_ignore\_noop() : current);

}

/\*!

@brief reads a UBJSON string

This function is either called after reading the 'S' byte explicitly

indicating a string, or in case of an object key where the 'S' byte can be

left out.

@param[out] result created string

@param[in] get\_char whether a new character should be retrieved from the

input (true, default) or whether the last read

character should be considered instead

@return whether string creation completed

\*/

bool get\_ubjson\_string(string\_t& result, const bool get\_char = true)

{

if (get\_char)

{

get(); // TODO(niels): may we ignore N here?

}

if (JSON\_HEDLEY\_UNLIKELY(not unexpect\_eof(input\_format\_t::ubjson, "value")))

{

return false;

}

switch (current)

{

case 'U':

{

std::uint8\_t len;

return get\_number(input\_format\_t::ubjson, len) and get\_string(input\_format\_t::ubjson, len, result);

}

case 'i':

{

std::int8\_t len;

return get\_number(input\_format\_t::ubjson, len) and get\_string(input\_format\_t::ubjson, len, result);

}

case 'I':

{

std::int16\_t len;

return get\_number(input\_format\_t::ubjson, len) and get\_string(input\_format\_t::ubjson, len, result);

}

case 'l':

{

std::int32\_t len;

return get\_number(input\_format\_t::ubjson, len) and get\_string(input\_format\_t::ubjson, len, result);

}

case 'L':

{

std::int64\_t len;

return get\_number(input\_format\_t::ubjson, len) and get\_string(input\_format\_t::ubjson, len, result);

}

default:

auto last\_token = get\_token\_string();

return sax->parse\_error(chars\_read, last\_token, parse\_error::create(113, chars\_read, exception\_message(input\_format\_t::ubjson, "expected length type specification (U, i, I, l, L); last byte: 0x" + last\_token, "string")));

}

}

/\*!

@param[out] result determined size

@return whether size determination completed

\*/

bool get\_ubjson\_size\_value(std::size\_t& result)

{

switch (get\_ignore\_noop())

{

case 'U':

{

std::uint8\_t number;

if (JSON\_HEDLEY\_UNLIKELY(not get\_number(input\_format\_t::ubjson, number)))

{

return false;

}

result = static\_cast<std::size\_t>(number);

return true;

}

case 'i':

{

std::int8\_t number;

if (JSON\_HEDLEY\_UNLIKELY(not get\_number(input\_format\_t::ubjson, number)))

{

return false;

}

result = static\_cast<std::size\_t>(number);

return true;

}

case 'I':

{

std::int16\_t number;

if (JSON\_HEDLEY\_UNLIKELY(not get\_number(input\_format\_t::ubjson, number)))

{

return false;

}

result = static\_cast<std::size\_t>(number);

return true;

}

case 'l':

{

std::int32\_t number;

if (JSON\_HEDLEY\_UNLIKELY(not get\_number(input\_format\_t::ubjson, number)))

{

return false;

}

result = static\_cast<std::size\_t>(number);

return true;

}

case 'L':

{

std::int64\_t number;

if (JSON\_HEDLEY\_UNLIKELY(not get\_number(input\_format\_t::ubjson, number)))

{

return false;

}

result = static\_cast<std::size\_t>(number);

return true;

}

default:

{

auto last\_token = get\_token\_string();

return sax->parse\_error(chars\_read, last\_token, parse\_error::create(113, chars\_read, exception\_message(input\_format\_t::ubjson, "expected length type specification (U, i, I, l, L) after '#'; last byte: 0x" + last\_token, "size")));

}

}

}

/\*!

@brief determine the type and size for a container

In the optimized UBJSON format, a type and a size can be provided to allow

for a more compact representation.

@param[out] result pair of the size and the type

@return whether pair creation completed

\*/

bool get\_ubjson\_size\_type(std::pair<std::size\_t, int>& result)

{

result.first = string\_t::npos; // size

result.second = 0; // type

get\_ignore\_noop();

if (current == '$')

{

result.second = get(); // must not ignore 'N', because 'N' maybe the type

if (JSON\_HEDLEY\_UNLIKELY(not unexpect\_eof(input\_format\_t::ubjson, "type")))

{

return false;

}

get\_ignore\_noop();

if (JSON\_HEDLEY\_UNLIKELY(current != '#'))

{

if (JSON\_HEDLEY\_UNLIKELY(not unexpect\_eof(input\_format\_t::ubjson, "value")))

{

return false;

}

auto last\_token = get\_token\_string();

return sax->parse\_error(chars\_read, last\_token, parse\_error::create(112, chars\_read, exception\_message(input\_format\_t::ubjson, "expected '#' after type information; last byte: 0x" + last\_token, "size")));

}

return get\_ubjson\_size\_value(result.first);

}

if (current == '#')

{

return get\_ubjson\_size\_value(result.first);

}

return true;

}

/\*!

@param prefix the previously read or set type prefix

@return whether value creation completed

\*/

bool get\_ubjson\_value(const int prefix)

{

switch (prefix)

{

case std::char\_traits<char>::eof(): // EOF

return unexpect\_eof(input\_format\_t::ubjson, "value");

case 'T': // true

return sax->boolean(true);

case 'F': // false

return sax->boolean(false);

case 'Z': // null

return sax->null();

case 'U':

{

std::uint8\_t number;

return get\_number(input\_format\_t::ubjson, number) and sax->number\_unsigned(number);

}

case 'i':

{

std::int8\_t number;

return get\_number(input\_format\_t::ubjson, number) and sax->number\_integer(number);

}

case 'I':

{

std::int16\_t number;

return get\_number(input\_format\_t::ubjson, number) and sax->number\_integer(number);

}

case 'l':

{

std::int32\_t number;

return get\_number(input\_format\_t::ubjson, number) and sax->number\_integer(number);

}

case 'L':

{

std::int64\_t number;

return get\_number(input\_format\_t::ubjson, number) and sax->number\_integer(number);

}

case 'd':

{

float number;

return get\_number(input\_format\_t::ubjson, number) and sax->number\_float(static\_cast<number\_float\_t>(number), "");

}

case 'D':

{

double number;

return get\_number(input\_format\_t::ubjson, number) and sax->number\_float(static\_cast<number\_float\_t>(number), "");

}

case 'C': // char

{

get();

if (JSON\_HEDLEY\_UNLIKELY(not unexpect\_eof(input\_format\_t::ubjson, "char")))

{

return false;

}

if (JSON\_HEDLEY\_UNLIKELY(current > 127))

{

auto last\_token = get\_token\_string();

return sax->parse\_error(chars\_read, last\_token, parse\_error::create(113, chars\_read, exception\_message(input\_format\_t::ubjson, "byte after 'C' must be in range 0x00..0x7F; last byte: 0x" + last\_token, "char")));

}

string\_t s(1, static\_cast<char>(current));

return sax->string(s);

}

case 'S': // string

{

string\_t s;

return get\_ubjson\_string(s) and sax->string(s);

}

case '[': // array

return get\_ubjson\_array();

case '{': // object

return get\_ubjson\_object();

default: // anything else

{

auto last\_token = get\_token\_string();

return sax->parse\_error(chars\_read, last\_token, parse\_error::create(112, chars\_read, exception\_message(input\_format\_t::ubjson, "invalid byte: 0x" + last\_token, "value")));

}

}

}

/\*!

@return whether array creation completed

\*/

bool get\_ubjson\_array()

{

std::pair<std::size\_t, int> size\_and\_type;

if (JSON\_HEDLEY\_UNLIKELY(not get\_ubjson\_size\_type(size\_and\_type)))

{

return false;

}

if (size\_and\_type.first != string\_t::npos)

{

if (JSON\_HEDLEY\_UNLIKELY(not sax->start\_array(size\_and\_type.first)))

{

return false;

}

if (size\_and\_type.second != 0)

{

if (size\_and\_type.second != 'N')

{

for (std::size\_t i = 0; i < size\_and\_type.first; ++i)

{

if (JSON\_HEDLEY\_UNLIKELY(not get\_ubjson\_value(size\_and\_type.second)))

{

return false;

}

}

}

}

else

{

for (std::size\_t i = 0; i < size\_and\_type.first; ++i)

{

if (JSON\_HEDLEY\_UNLIKELY(not parse\_ubjson\_internal()))

{

return false;

}

}

}

}

else

{

if (JSON\_HEDLEY\_UNLIKELY(not sax->start\_array(std::size\_t(-1))))

{

return false;

}

while (current != ']')

{

if (JSON\_HEDLEY\_UNLIKELY(not parse\_ubjson\_internal(false)))

{

return false;

}

get\_ignore\_noop();

}

}

return sax->end\_array();

}

/\*!

@return whether object creation completed

\*/

bool get\_ubjson\_object()

{

std::pair<std::size\_t, int> size\_and\_type;

if (JSON\_HEDLEY\_UNLIKELY(not get\_ubjson\_size\_type(size\_and\_type)))

{

return false;

}

string\_t key;

if (size\_and\_type.first != string\_t::npos)

{

if (JSON\_HEDLEY\_UNLIKELY(not sax->start\_object(size\_and\_type.first)))

{

return false;

}

if (size\_and\_type.second != 0)

{

for (std::size\_t i = 0; i < size\_and\_type.first; ++i)

{

if (JSON\_HEDLEY\_UNLIKELY(not get\_ubjson\_string(key) or not sax->key(key)))

{

return false;

}

if (JSON\_HEDLEY\_UNLIKELY(not get\_ubjson\_value(size\_and\_type.second)))

{

return false;

}

key.clear();

}

}

else

{

for (std::size\_t i = 0; i < size\_and\_type.first; ++i)

{

if (JSON\_HEDLEY\_UNLIKELY(not get\_ubjson\_string(key) or not sax->key(key)))

{

return false;

}

if (JSON\_HEDLEY\_UNLIKELY(not parse\_ubjson\_internal()))

{

return false;

}

key.clear();

}

}

}

else

{

if (JSON\_HEDLEY\_UNLIKELY(not sax->start\_object(std::size\_t(-1))))

{

return false;

}

while (current != '}')

{

if (JSON\_HEDLEY\_UNLIKELY(not get\_ubjson\_string(key, false) or not sax->key(key)))

{

return false;

}

if (JSON\_HEDLEY\_UNLIKELY(not parse\_ubjson\_internal()))

{

return false;

}

get\_ignore\_noop();

key.clear();

}

}

return sax->end\_object();

}

///////////////////////

// Utility functions //

///////////////////////

/\*!

@brief get next character from the input

This function provides the interface to the used input adapter. It does

not throw in case the input reached EOF, but returns a -'ve valued

`std::char\_traits<char>::eof()` in that case.

@return character read from the input

\*/

int get()

{

++chars\_read;

return current = ia->get\_character();

}

/\*!

@return character read from the input after ignoring all 'N' entries

\*/

int get\_ignore\_noop()

{

do

{

get();

}

while (current == 'N');

return current;

}

/\*

@brief read a number from the input

@tparam NumberType the type of the number

@param[in] format the current format (for diagnostics)

@param[out] result number of type @a NumberType

@return whether conversion completed

@note This function needs to respect the system's endianess, because

bytes in CBOR, MessagePack, and UBJSON are stored in network order

(big endian) and therefore need reordering on little endian systems.

\*/

template<typename NumberType, bool InputIsLittleEndian = false>

bool get\_number(const input\_format\_t format, NumberType& result)

{

// step 1: read input into array with system's byte order

std::array<std::uint8\_t, sizeof(NumberType)> vec;

for (std::size\_t i = 0; i < sizeof(NumberType); ++i)

{

get();

if (JSON\_HEDLEY\_UNLIKELY(not unexpect\_eof(format, "number")))

{

return false;

}

// reverse byte order prior to conversion if necessary

if (is\_little\_endian != InputIsLittleEndian)

{

vec[sizeof(NumberType) - i - 1] = static\_cast<std::uint8\_t>(current);

}

else

{

vec[i] = static\_cast<std::uint8\_t>(current); // LCOV\_EXCL\_LINE

}

}

// step 2: convert array into number of type T and return

std::memcpy(&result, vec.data(), sizeof(NumberType));

return true;

}

/\*!

@brief create a string by reading characters from the input

@tparam NumberType the type of the number

@param[in] format the current format (for diagnostics)

@param[in] len number of characters to read

@param[out] result string created by reading @a len bytes

@return whether string creation completed

@note We can not reserve @a len bytes for the result, because @a len

may be too large. Usually, @ref unexpect\_eof() detects the end of

the input before we run out of string memory.

\*/

template<typename NumberType>

bool get\_string(const input\_format\_t format,

const NumberType len,

string\_t& result)

{

bool success = true;

std::generate\_n(std::back\_inserter(result), len, [this, &success, &format]()

{

get();

if (JSON\_HEDLEY\_UNLIKELY(not unexpect\_eof(format, "string")))

{

success = false;

}

return static\_cast<char>(current);

});

return success;

}

/\*!

@param[in] format the current format (for diagnostics)

@param[in] context further context information (for diagnostics)

@return whether the last read character is not EOF

\*/

JSON\_HEDLEY\_NON\_NULL(3)

bool unexpect\_eof(const input\_format\_t format, const char\* context) const

{

if (JSON\_HEDLEY\_UNLIKELY(current == std::char\_traits<char>::eof()))

{

return sax->parse\_error(chars\_read, "<end of file>",

parse\_error::create(110, chars\_read, exception\_message(format, "unexpected end of input", context)));

}

return true;

}

/\*!

@return a string representation of the last read byte

\*/

std::string get\_token\_string() const

{

std::array<char, 3> cr{{}};

(std::snprintf)(cr.data(), cr.size(), "%.2hhX", static\_cast<unsigned char>(current));

return std::string{cr.data()};

}

/\*!

@param[in] format the current format

@param[in] detail a detailed error message

@param[in] context further context information

@return a message string to use in the parse\_error exceptions

\*/

std::string exception\_message(const input\_format\_t format,

const std::string& detail,

const std::string& context) const

{

std::string error\_msg = "syntax error while parsing ";

switch (format)

{

case input\_format\_t::cbor:

error\_msg += "CBOR";

break;

case input\_format\_t::msgpack:

error\_msg += "MessagePack";

break;

case input\_format\_t::ubjson:

error\_msg += "UBJSON";

break;

case input\_format\_t::bson:

error\_msg += "BSON";

break;

default: // LCOV\_EXCL\_LINE

assert(false); // LCOV\_EXCL\_LINE

}

return error\_msg + " " + context + ": " + detail;

}

private:

/// input adapter

input\_adapter\_t ia = nullptr;

/// the current character

int current = std::char\_traits<char>::eof();

/// the number of characters read

std::size\_t chars\_read = 0;

/// whether we can assume little endianess

const bool is\_little\_endian = little\_endianess();

/// the SAX parser

json\_sax\_t\* sax = nullptr;

};

} // namespace detail

} // namespace nlohmann

// #include <nlohmann/detail/input/input\_adapters.hpp>

// #include <nlohmann/detail/input/lexer.hpp>

#include <array> // array

#include <clocale> // localeconv

#include <cstddef> // size\_t

#include <cstdio> // snprintf

#include <cstdlib> // strtof, strtod, strtold, strtoll, strtoull

#include <initializer\_list> // initializer\_list

#include <string> // char\_traits, string

#include <utility> // move

#include <vector> // vector

// #include <nlohmann/detail/input/input\_adapters.hpp>

// #include <nlohmann/detail/input/position\_t.hpp>

// #include <nlohmann/detail/macro\_scope.hpp>

namespace nlohmann

{

namespace detail

{

///////////

// lexer //

///////////

/\*!

@brief lexical analysis

This class organizes the lexical analysis during JSON deserialization.

\*/

template<typename BasicJsonType>

class lexer

{

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;

public:

/// token types for the parser

enum class token\_type

{

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)

};

/// return name of values of type token\_type (only used for errors)

JSON\_HEDLEY\_RETURNS\_NON\_NULL

JSON\_HEDLEY\_CONST

static const char\* token\_type\_name(const token\_type t) noexcept

{

switch (t)

{

case token\_type::uninitialized:

return "<uninitialized>";

case token\_type::literal\_true:

return "true literal";

case token\_type::literal\_false:

return "false literal";

case token\_type::literal\_null:

return "null literal";

case token\_type::value\_string:

return "string literal";

case lexer::token\_type::value\_unsigned:

case lexer::token\_type::value\_integer:

case lexer::token\_type::value\_float:

return "number literal";

case token\_type::begin\_array:

return "'['";

case token\_type::begin\_object:

return "'{'";

case token\_type::end\_array:

return "']'";

case token\_type::end\_object:

return "'}'";

case token\_type::name\_separator:

return "':'";

case token\_type::value\_separator:

return "','";

case token\_type::parse\_error:

return "<parse error>";

case token\_type::end\_of\_input:

return "end of input";

case token\_type::literal\_or\_value:

return "'[', '{', or a literal";

// LCOV\_EXCL\_START

default: // catch non-enum values

return "unknown token";

// LCOV\_EXCL\_STOP

}

}

explicit lexer(detail::input\_adapter\_t&& adapter)

: ia(std::move(adapter)), decimal\_point\_char(get\_decimal\_point()) {}

// delete because of pointer members

lexer(const lexer&) = delete;

lexer(lexer&&) = delete;

lexer& operator=(lexer&) = delete;

lexer& operator=(lexer&&) = delete;

~lexer() = default;

private:

/////////////////////

// locales

/////////////////////

/// return the locale-dependent decimal point

JSON\_HEDLEY\_PURE

static char get\_decimal\_point() noexcept

{

const auto loc = localeconv();

assert(loc != nullptr);

return (loc->decimal\_point == nullptr) ? '.' : \*(loc->decimal\_point);

}

/////////////////////

// scan functions

/////////////////////

/\*!

@brief get codepoint from 4 hex characters following `\u`

For input "\u c1 c2 c3 c4" the codepoint is:

(c1 \* 0x1000) + (c2 \* 0x0100) + (c3 \* 0x0010) + c4

= (c1 << 12) + (c2 << 8) + (c3 << 4) + (c4 << 0)

Furthermore, the possible characters '0'..'9', 'A'..'F', and 'a'..'f'

must be converted to the integers 0x0..0x9, 0xA..0xF, 0xA..0xF, resp. The

conversion is done by subtracting the offset (0x30, 0x37, and 0x57)

between the ASCII value of the character and the desired integer value.

@return codepoint (0x0000..0xFFFF) or -1 in case of an error (e.g. EOF or

non-hex character)

\*/

int get\_codepoint()

{

// this function only makes sense after reading `\u`

assert(current == 'u');

int codepoint = 0;

const auto factors = { 12u, 8u, 4u, 0u };

for (const auto factor : factors)

{

get();

if (current >= '0' and current <= '9')

{

codepoint += static\_cast<int>((static\_cast<unsigned int>(current) - 0x30u) << factor);

}

else if (current >= 'A' and current <= 'F')

{

codepoint += static\_cast<int>((static\_cast<unsigned int>(current) - 0x37u) << factor);

}

else if (current >= 'a' and current <= 'f')

{

codepoint += static\_cast<int>((static\_cast<unsigned int>(current) - 0x57u) << factor);

}

else

{

return -1;

}

}

assert(0x0000 <= codepoint and codepoint <= 0xFFFF);

return codepoint;

}

/\*!

@brief check if the next byte(s) are inside a given range

Adds the current byte and, for each passed range, reads a new byte and

checks if it is inside the range. If a violation was detected, set up an

error message and return false. Otherwise, return true.

@param[in] ranges list of integers; interpreted as list of pairs of

inclusive lower and upper bound, respectively

@pre The passed list @a ranges must have 2, 4, or 6 elements; that is,

1, 2, or 3 pairs. This precondition is enforced by an assertion.

@return true if and only if no range violation was detected

\*/

bool next\_byte\_in\_range(std::initializer\_list<int> ranges)

{

assert(ranges.size() == 2 or ranges.size() == 4 or ranges.size() == 6);

add(current);

for (auto range = ranges.begin(); range != ranges.end(); ++range)

{

get();

if (JSON\_HEDLEY\_LIKELY(\*range <= current and current <= \*(++range)))

{

add(current);

}

else

{

error\_message = "invalid string: ill-formed UTF-8 byte";

return false;

}

}

return true;

}

/\*!

@brief scan a string literal

This function scans a string according to Sect. 7 of RFC 7159. While

scanning, bytes are escaped and copied into buffer token\_buffer. Then the

function returns successfully, token\_buffer is \*not\* null-terminated (as it

may contain \0 bytes), and token\_buffer.size() is the number of bytes in the

string.

@return token\_type::value\_string if string could be successfully scanned,

token\_type::parse\_error otherwise

@note In case of errors, variable error\_message contains a textual

description.

\*/

token\_type scan\_string()

{

// reset token\_buffer (ignore opening quote)

reset();

// we entered the function by reading an open quote

assert(current == '\"');

while (true)

{

// get next character

switch (get())

{

// end of file while parsing string

case std::char\_traits<char>::eof():

{

error\_message = "invalid string: missing closing quote";

return token\_type::parse\_error;

}

// closing quote

case '\"':

{

return token\_type::value\_string;

}

// escapes

case '\\':

{

switch (get())

{

// quotation mark

case '\"':

add('\"');

break;

// reverse solidus

case '\\':

add('\\');

break;

// solidus

case '/':

add('/');

break;

// backspace

case 'b':

add('\b');

break;

// form feed

case 'f':

add('\f');

break;

// line feed

case 'n':

add('\n');

break;

// carriage return

case 'r':

add('\r');

break;

// tab

case 't':

add('\t');

break;

// unicode escapes

case 'u':

{

const int codepoint1 = get\_codepoint();

int codepoint = codepoint1; // start with codepoint1

if (JSON\_HEDLEY\_UNLIKELY(codepoint1 == -1))

{

error\_message = "invalid string: '\\u' must be followed by 4 hex digits";

return token\_type::parse\_error;

}

// check if code point is a high surrogate

if (0xD800 <= codepoint1 and codepoint1 <= 0xDBFF)

{

// expect next \uxxxx entry

if (JSON\_HEDLEY\_LIKELY(get() == '\\' and get() == 'u'))

{

const int codepoint2 = get\_codepoint();

if (JSON\_HEDLEY\_UNLIKELY(codepoint2 == -1))

{

error\_message = "invalid string: '\\u' must be followed by 4 hex digits";

return token\_type::parse\_error;

}

// check if codepoint2 is a low surrogate

if (JSON\_HEDLEY\_LIKELY(0xDC00 <= codepoint2 and codepoint2 <= 0xDFFF))

{

// overwrite codepoint

codepoint = static\_cast<int>(

// high surrogate occupies the most significant 22 bits

(static\_cast<unsigned int>(codepoint1) << 10u)

// low surrogate occupies the least significant 15 bits

+ static\_cast<unsigned int>(codepoint2)

// there is still the 0xD800, 0xDC00 and 0x10000 noise

// in the result so we have to subtract with:

// (0xD800 << 10) + DC00 - 0x10000 = 0x35FDC00

- 0x35FDC00u);

}

else

{

error\_message = "invalid string: surrogate U+DC00..U+DFFF must be followed by U+DC00..U+DFFF";

return token\_type::parse\_error;

}

}

else

{

error\_message = "invalid string: surrogate U+DC00..U+DFFF must be followed by U+DC00..U+DFFF";

return token\_type::parse\_error;

}

}

else

{

if (JSON\_HEDLEY\_UNLIKELY(0xDC00 <= codepoint1 and codepoint1 <= 0xDFFF))

{

error\_message = "invalid string: surrogate U+DC00..U+DFFF must follow U+D800..U+DBFF";

return token\_type::parse\_error;

}

}

// result of the above calculation yields a proper codepoint

assert(0x00 <= codepoint and codepoint <= 0x10FFFF);

// translate codepoint into bytes

if (codepoint < 0x80)

{

// 1-byte characters: 0xxxxxxx (ASCII)

add(codepoint);

}

else if (codepoint <= 0x7FF)

{

// 2-byte characters: 110xxxxx 10xxxxxx

add(static\_cast<int>(0xC0u | (static\_cast<unsigned int>(codepoint) >> 6u)));

add(static\_cast<int>(0x80u | (static\_cast<unsigned int>(codepoint) & 0x3Fu)));

}

else if (codepoint <= 0xFFFF)

{

// 3-byte characters: 1110xxxx 10xxxxxx 10xxxxxx

add(static\_cast<int>(0xE0u | (static\_cast<unsigned int>(codepoint) >> 12u)));

add(static\_cast<int>(0x80u | ((static\_cast<unsigned int>(codepoint) >> 6u) & 0x3Fu)));

add(static\_cast<int>(0x80u | (static\_cast<unsigned int>(codepoint) & 0x3Fu)));

}

else

{

// 4-byte characters: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

add(static\_cast<int>(0xF0u | (static\_cast<unsigned int>(codepoint) >> 18u)));

add(static\_cast<int>(0x80u | ((static\_cast<unsigned int>(codepoint) >> 12u) & 0x3Fu)));

add(static\_cast<int>(0x80u | ((static\_cast<unsigned int>(codepoint) >> 6u) & 0x3Fu)));

add(static\_cast<int>(0x80u | (static\_cast<unsigned int>(codepoint) & 0x3Fu)));

}

break;

}

// other characters after escape

default:

error\_message = "invalid string: forbidden character after backslash";

return token\_type::parse\_error;

}

break;

}

// invalid control characters

case 0x00:

{

error\_message = "invalid string: control character U+0000 (NUL) must be escaped to \\u0000";

return token\_type::parse\_error;

}

case 0x01:

{

error\_message = "invalid string: control character U+0001 (SOH) must be escaped to \\u0001";

return token\_type::parse\_error;

}

case 0x02:

{

error\_message = "invalid string: control character U+0002 (STX) must be escaped to \\u0002";

return token\_type::parse\_error;

}

case 0x03:

{

error\_message = "invalid string: control character U+0003 (ETX) must be escaped to \\u0003";

return token\_type::parse\_error;

}

case 0x04:

{

error\_message = "invalid string: control character U+0004 (EOT) must be escaped to \\u0004";

return token\_type::parse\_error;

}

case 0x05:

{

error\_message = "invalid string: control character U+0005 (ENQ) must be escaped to \\u0005";

return token\_type::parse\_error;

}

case 0x06:

{

error\_message = "invalid string: control character U+0006 (ACK) must be escaped to \\u0006";

return token\_type::parse\_error;

}

case 0x07:

{

error\_message = "invalid string: control character U+0007 (BEL) must be escaped to \\u0007";

return token\_type::parse\_error;

}

case 0x08:

{

error\_message = "invalid string: control character U+0008 (BS) must be escaped to \\u0008 or \\b";

return token\_type::parse\_error;

}

case 0x09:

{

error\_message = "invalid string: control character U+0009 (HT) must be escaped to \\u0009 or \\t";

return token\_type::parse\_error;

}

case 0x0A:

{

error\_message = "invalid string: control character U+000A (LF) must be escaped to \\u000A or \\n";

return token\_type::parse\_error;

}

case 0x0B:

{

error\_message = "invalid string: control character U+000B (VT) must be escaped to \\u000B";

return token\_type::parse\_error;

}

case 0x0C:

{

error\_message = "invalid string: control character U+000C (FF) must be escaped to \\u000C or \\f";

return token\_type::parse\_error;

}

case 0x0D:

{

error\_message = "invalid string: control character U+000D (CR) must be escaped to \\u000D or \\r";

return token\_type::parse\_error;

}

case 0x0E:

{

error\_message = "invalid string: control character U+000E (SO) must be escaped to \\u000E";

return token\_type::parse\_error;

}

case 0x0F:

{

error\_message = "invalid string: control character U+000F (SI) must be escaped to \\u000F";

return token\_type::parse\_error;

}

case 0x10:

{

error\_message = "invalid string: control character U+0010 (DLE) must be escaped to \\u0010";

return token\_type::parse\_error;

}

case 0x11:

{

error\_message = "invalid string: control character U+0011 (DC1) must be escaped to \\u0011";

return token\_type::parse\_error;

}

case 0x12:

{

error\_message = "invalid string: control character U+0012 (DC2) must be escaped to \\u0012";

return token\_type::parse\_error;

}

case 0x13:

{

error\_message = "invalid string: control character U+0013 (DC3) must be escaped to \\u0013";

return token\_type::parse\_error;

}

case 0x14:

{

error\_message = "invalid string: control character U+0014 (DC4) must be escaped to \\u0014";

return token\_type::parse\_error;

}

case 0x15:

{

error\_message = "invalid string: control character U+0015 (NAK) must be escaped to \\u0015";

return token\_type::parse\_error;

}

case 0x16:

{

error\_message = "invalid string: control character U+0016 (SYN) must be escaped to \\u0016";

return token\_type::parse\_error;

}

case 0x17:

{

error\_message = "invalid string: control character U+0017 (ETB) must be escaped to \\u0017";

return token\_type::parse\_error;

}

case 0x18:

{

error\_message = "invalid string: control character U+0018 (CAN) must be escaped to \\u0018";

return token\_type::parse\_error;

}

case 0x19:

{

error\_message = "invalid string: control character U+0019 (EM) must be escaped to \\u0019";

return token\_type::parse\_error;

}

case 0x1A:

{

error\_message = "invalid string: control character U+001A (SUB) must be escaped to \\u001A";

return token\_type::parse\_error;

}

case 0x1B:

{

error\_message = "invalid string: control character U+001B (ESC) must be escaped to \\u001B";

return token\_type::parse\_error;

}

case 0x1C:

{

error\_message = "invalid string: control character U+001C (FS) must be escaped to \\u001C";

return token\_type::parse\_error;

}

case 0x1D:

{

error\_message = "invalid string: control character U+001D (GS) must be escaped to \\u001D";

return token\_type::parse\_error;

}

case 0x1E:

{

error\_message = "invalid string: control character U+001E (RS) must be escaped to \\u001E";

return token\_type::parse\_error;

}

case 0x1F:

{

error\_message = "invalid string: control character U+001F (US) must be escaped to \\u001F";

return token\_type::parse\_error;

}

// U+0020..U+007F (except U+0022 (quote) and U+005C (backspace))

case 0x20:

case 0x21:

case 0x23:

case 0x24:

case 0x25:

case 0x26:

case 0x27:

case 0x28:

case 0x29:

case 0x2A:

case 0x2B:

case 0x2C:

case 0x2D:

case 0x2E:

case 0x2F:

case 0x30:

case 0x31:

case 0x32:

case 0x33:

case 0x34:

case 0x35:

case 0x36:

case 0x37:

case 0x38:

case 0x39:

case 0x3A:

case 0x3B:

case 0x3C:

case 0x3D:

case 0x3E:

case 0x3F:

case 0x40:

case 0x41:

case 0x42:

case 0x43:

case 0x44:

case 0x45:

case 0x46:

case 0x47:

case 0x48:

case 0x49:

case 0x4A:

case 0x4B:

case 0x4C:

case 0x4D:

case 0x4E:

case 0x4F:

case 0x50:

case 0x51:

case 0x52:

case 0x53:

case 0x54:

case 0x55:

case 0x56:

case 0x57:

case 0x58:

case 0x59:

case 0x5A:

case 0x5B:

case 0x5D:

case 0x5E:

case 0x5F:

case 0x60:

case 0x61:

case 0x62:

case 0x63:

case 0x64:

case 0x65:

case 0x66:

case 0x67:

case 0x68:

case 0x69:

case 0x6A:

case 0x6B:

case 0x6C:

case 0x6D:

case 0x6E:

case 0x6F:

case 0x70:

case 0x71:

case 0x72:

case 0x73:

case 0x74:

case 0x75:

case 0x76:

case 0x77:

case 0x78:

case 0x79:

case 0x7A:

case 0x7B:

case 0x7C:

case 0x7D:

case 0x7E:

case 0x7F:

{

add(current);

break;

}

// U+0080..U+07FF: bytes C2..DF 80..BF

case 0xC2:

case 0xC3:

case 0xC4:

case 0xC5:

case 0xC6:

case 0xC7:

case 0xC8:

case 0xC9:

case 0xCA:

case 0xCB:

case 0xCC:

case 0xCD:

case 0xCE:

case 0xCF:

case 0xD0:

case 0xD1:

case 0xD2:

case 0xD3:

case 0xD4:

case 0xD5:

case 0xD6:

case 0xD7:

case 0xD8:

case 0xD9:

case 0xDA:

case 0xDB:

case 0xDC:

case 0xDD:

case 0xDE:

case 0xDF:

{

if (JSON\_HEDLEY\_UNLIKELY(not next\_byte\_in\_range({0x80, 0xBF})))

{

return token\_type::parse\_error;

}

break;

}

// U+0800..U+0FFF: bytes E0 A0..BF 80..BF

case 0xE0:

{

if (JSON\_HEDLEY\_UNLIKELY(not (next\_byte\_in\_range({0xA0, 0xBF, 0x80, 0xBF}))))

{

return token\_type::parse\_error;

}

break;

}

// U+1000..U+CFFF: bytes E1..EC 80..BF 80..BF

// U+E000..U+FFFF: bytes EE..EF 80..BF 80..BF

case 0xE1:

case 0xE2:

case 0xE3:

case 0xE4:

case 0xE5:

case 0xE6:

case 0xE7:

case 0xE8:

case 0xE9:

case 0xEA:

case 0xEB:

case 0xEC:

case 0xEE:

case 0xEF:

{

if (JSON\_HEDLEY\_UNLIKELY(not (next\_byte\_in\_range({0x80, 0xBF, 0x80, 0xBF}))))

{

return token\_type::parse\_error;

}

break;

}

// U+D000..U+D7FF: bytes ED 80..9F 80..BF

case 0xED:

{

if (JSON\_HEDLEY\_UNLIKELY(not (next\_byte\_in\_range({0x80, 0x9F, 0x80, 0xBF}))))

{

return token\_type::parse\_error;

}

break;

}

// U+10000..U+3FFFF F0 90..BF 80..BF 80..BF

case 0xF0:

{

if (JSON\_HEDLEY\_UNLIKELY(not (next\_byte\_in\_range({0x90, 0xBF, 0x80, 0xBF, 0x80, 0xBF}))))

{

return token\_type::parse\_error;

}

break;

}

// U+40000..U+FFFFF F1..F3 80..BF 80..BF 80..BF

case 0xF1:

case 0xF2:

case 0xF3:

{

if (JSON\_HEDLEY\_UNLIKELY(not (next\_byte\_in\_range({0x80, 0xBF, 0x80, 0xBF, 0x80, 0xBF}))))

{

return token\_type::parse\_error;

}

break;

}

// U+100000..U+10FFFF F4 80..8F 80..BF 80..BF

case 0xF4:

{

if (JSON\_HEDLEY\_UNLIKELY(not (next\_byte\_in\_range({0x80, 0x8F, 0x80, 0xBF, 0x80, 0xBF}))))

{

return token\_type::parse\_error;

}

break;

}

// remaining bytes (80..C1 and F5..FF) are ill-formed

default:

{

error\_message = "invalid string: ill-formed UTF-8 byte";

return token\_type::parse\_error;

}

}

}

}

JSON\_HEDLEY\_NON\_NULL(2)

static void strtof(float& f, const char\* str, char\*\* endptr) noexcept

{

f = std::strtof(str, endptr);

}

JSON\_HEDLEY\_NON\_NULL(2)

static void strtof(double& f, const char\* str, char\*\* endptr) noexcept

{

f = std::strtod(str, endptr);

}

JSON\_HEDLEY\_NON\_NULL(2)

static void strtof(long double& f, const char\* str, char\*\* endptr) noexcept

{

f = std::strtold(str, endptr);

}

/\*!

@brief scan a number literal

This function scans a string according to Sect. 6 of RFC 7159.

The function is realized with a deterministic finite state machine derived

from the grammar described in RFC 7159. Starting in state "init", the

input is read and used to determined the next state. Only state "done"

accepts the number. State "error" is a trap state to model errors. In the

table below, "anything" means any character but the ones listed before.

state | 0 | 1-9 | e E | + | - | . | anything

---------|----------|----------|----------|---------|---------|----------|-----------

init | zero | any1 | [error] | [error] | minus | [error] | [error]

minus | zero | any1 | [error] | [error] | [error] | [error] | [error]

zero | done | done | exponent | done | done | decimal1 | done

any1 | any1 | any1 | exponent | done | done | decimal1 | done

decimal1 | decimal2 | [error] | [error] | [error] | [error] | [error] | [error]

decimal2 | decimal2 | decimal2 | exponent | done | done | done | done

exponent | any2 | any2 | [error] | sign | sign | [error] | [error]

sign | any2 | any2 | [error] | [error] | [error] | [error] | [error]

any2 | any2 | any2 | done | done | done | done | done

The state machine is realized with one label per state (prefixed with

"scan\_number\_") and `goto` statements between them. The state machine

contains cycles, but any cycle can be left when EOF is read. Therefore,

the function is guaranteed to terminate.

During scanning, the read bytes are stored in token\_buffer. This string is

then converted to a signed integer, an unsigned integer, or a

floating-point number.

@return token\_type::value\_unsigned, token\_type::value\_integer, or

token\_type::value\_float if number could be successfully scanned,

token\_type::parse\_error otherwise

@note The scanner is independent of the current locale. Internally, the

locale's decimal point is used instead of `.` to work with the

locale-dependent converters.

\*/

token\_type scan\_number() // lgtm [cpp/use-of-goto]

{

// reset token\_buffer to store the number's bytes

reset();

// the type of the parsed number; initially set to unsigned; will be

// changed if minus sign, decimal point or exponent is read

token\_type number\_type = token\_type::value\_unsigned;

// state (init): we just found out we need to scan a number

switch (current)

{

case '-':

{

add(current);

goto scan\_number\_minus;

}

case '0':

{

add(current);

goto scan\_number\_zero;

}

case '1':

case '2':

case '3':

case '4':

case '5':

case '6':

case '7':

case '8':

case '9':

{

add(current);

goto scan\_number\_any1;

}

// all other characters are rejected outside scan\_number()

default: // LCOV\_EXCL\_LINE

assert(false); // LCOV\_EXCL\_LINE

}

scan\_number\_minus:

// state: we just parsed a leading minus sign

number\_type = token\_type::value\_integer;

switch (get())

{

case '0':

{

add(current);

goto scan\_number\_zero;

}

case '1':

case '2':

case '3':

case '4':

case '5':

case '6':

case '7':

case '8':

case '9':

{

add(current);

goto scan\_number\_any1;

}

default:

{

error\_message = "invalid number; expected digit after '-'";

return token\_type::parse\_error;

}

}

scan\_number\_zero:

// state: we just parse a zero (maybe with a leading minus sign)

switch (get())

{

case '.':

{

add(decimal\_point\_char);

goto scan\_number\_decimal1;

}

case 'e':

case 'E':

{

add(current);

goto scan\_number\_exponent;

}

default:

goto scan\_number\_done;

}

scan\_number\_any1:

// state: we just parsed a number 0-9 (maybe with a leading minus sign)

switch (get())

{

case '0':

case '1':

case '2':

case '3':

case '4':

case '5':

case '6':

case '7':

case '8':

case '9':

{

add(current);

goto scan\_number\_any1;

}

case '.':

{

add(decimal\_point\_char);

goto scan\_number\_decimal1;

}

case 'e':

case 'E':

{

add(current);

goto scan\_number\_exponent;

}

default:

goto scan\_number\_done;

}

scan\_number\_decimal1:

// state: we just parsed a decimal point

number\_type = token\_type::value\_float;

switch (get())

{

case '0':

case '1':

case '2':

case '3':

case '4':

case '5':

case '6':

case '7':

case '8':

case '9':

{

add(current);

goto scan\_number\_decimal2;

}

default:

{

error\_message = "invalid number; expected digit after '.'";

return token\_type::parse\_error;

}

}

scan\_number\_decimal2:

// we just parsed at least one number after a decimal point

switch (get())

{

case '0':

case '1':

case '2':

case '3':

case '4':

case '5':

case '6':

case '7':

case '8':

case '9':

{

add(current);

goto scan\_number\_decimal2;

}

case 'e':

case 'E':

{

add(current);

goto scan\_number\_exponent;

}

default:

goto scan\_number\_done;

}

scan\_number\_exponent:

// we just parsed an exponent

number\_type = token\_type::value\_float;

switch (get())

{

case '+':

case '-':

{

add(current);

goto scan\_number\_sign;

}

case '0':

case '1':

case '2':

case '3':

case '4':

case '5':

case '6':

case '7':

case '8':

case '9':

{

add(current);

goto scan\_number\_any2;

}

default:

{

error\_message =

"invalid number; expected '+', '-', or digit after exponent";

return token\_type::parse\_error;

}

}

scan\_number\_sign:

// we just parsed an exponent sign

switch (get())

{

case '0':

case '1':

case '2':

case '3':

case '4':

case '5':

case '6':

case '7':

case '8':

case '9':

{

add(current);

goto scan\_number\_any2;

}

default:

{

error\_message = "invalid number; expected digit after exponent sign";

return token\_type::parse\_error;

}

}

scan\_number\_any2:

// we just parsed a number after the exponent or exponent sign

switch (get())

{

case '0':

case '1':

case '2':

case '3':

case '4':

case '5':

case '6':

case '7':

case '8':

case '9':

{

add(current);

goto scan\_number\_any2;

}

default:

goto scan\_number\_done;

}

scan\_number\_done:

// unget the character after the number (we only read it to know that

// we are done scanning a number)

unget();

char\* endptr = nullptr;

errno = 0;

// try to parse integers first and fall back to floats

if (number\_type == token\_type::value\_unsigned)

{

const auto x = std::strtoull(token\_buffer.data(), &endptr, 10);

// we checked the number format before

assert(endptr == token\_buffer.data() + token\_buffer.size());

if (errno == 0)

{

value\_unsigned = static\_cast<number\_unsigned\_t>(x);

if (value\_unsigned == x)

{

return token\_type::value\_unsigned;

}

}

}

else if (number\_type == token\_type::value\_integer)

{

const auto x = std::strtoll(token\_buffer.data(), &endptr, 10);

// we checked the number format before

assert(endptr == token\_buffer.data() + token\_buffer.size());

if (errno == 0)

{

value\_integer = static\_cast<number\_integer\_t>(x);

if (value\_integer == x)

{

return token\_type::value\_integer;

}

}

}

// this code is reached if we parse a floating-point number or if an

// integer conversion above failed

strtof(value\_float, token\_buffer.data(), &endptr);

// we checked the number format before

assert(endptr == token\_buffer.data() + token\_buffer.size());

return token\_type::value\_float;

}

/\*!

@param[in] literal\_text the literal text to expect

@param[in] length the length of the passed literal text

@param[in] return\_type the token type to return on success

\*/

JSON\_HEDLEY\_NON\_NULL(2)

token\_type scan\_literal(const char\* literal\_text, const std::size\_t length,

token\_type return\_type)

{

assert(current == literal\_text[0]);

for (std::size\_t i = 1; i < length; ++i)

{

if (JSON\_HEDLEY\_UNLIKELY(get() != literal\_text[i]))

{

error\_message = "invalid literal";

return token\_type::parse\_error;

}

}

return return\_type;

}

/////////////////////

// input management

/////////////////////

/// reset token\_buffer; current character is beginning of token

void reset() noexcept

{

token\_buffer.clear();

token\_string.clear();

token\_string.push\_back(std::char\_traits<char>::to\_char\_type(current));

}

/\*

@brief get next character from the input

This function provides the interface to the used input adapter. It does

not throw in case the input reached EOF, but returns a

`std::char\_traits<char>::eof()` in that case. Stores the scanned characters

for use in error messages.

@return character read from the input

\*/

std::char\_traits<char>::int\_type get()

{

++position.chars\_read\_total;

++position.chars\_read\_current\_line;

if (next\_unget)

{

// just reset the next\_unget variable and work with current

next\_unget = false;

}

else

{

current = ia->get\_character();

}

if (JSON\_HEDLEY\_LIKELY(current != std::char\_traits<char>::eof()))

{

token\_string.push\_back(std::char\_traits<char>::to\_char\_type(current));

}

if (current == '\n')

{

++position.lines\_read;

position.chars\_read\_current\_line = 0;

}

return current;

}

/\*!

@brief unget current character (read it again on next get)

We implement unget by setting variable next\_unget to true. The input is not

changed - we just simulate ungetting by modifying chars\_read\_total,

chars\_read\_current\_line, and token\_string. The next call to get() will

behave as if the unget character is read again.

\*/

void unget()

{

next\_unget = true;

--position.chars\_read\_total;

// in case we "unget" a newline, we have to also decrement the lines\_read

if (position.chars\_read\_current\_line == 0)

{

if (position.lines\_read > 0)

{

--position.lines\_read;

}

}

else

{

--position.chars\_read\_current\_line;

}

if (JSON\_HEDLEY\_LIKELY(current != std::char\_traits<char>::eof()))

{

assert(not token\_string.empty());

token\_string.pop\_back();

}

}

/// add a character to token\_buffer

void add(int c)

{

token\_buffer.push\_back(std::char\_traits<char>::to\_char\_type(c));

}

public:

/////////////////////

// value getters

/////////////////////

/// return integer value

constexpr number\_integer\_t get\_number\_integer() const noexcept

{

return value\_integer;

}

/// return unsigned integer value

constexpr number\_unsigned\_t get\_number\_unsigned() const noexcept

{

return value\_unsigned;

}

/// return floating-point value

constexpr number\_float\_t get\_number\_float() const noexcept

{

return value\_float;

}

/// return current string value (implicitly resets the token; useful only once)

string\_t& get\_string()

{

return token\_buffer;

}

/////////////////////

// diagnostics

/////////////////////

/// return position of last read token

constexpr position\_t get\_position() const noexcept

{

return position;

}

/// 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.

std::string get\_token\_string() const

{

// escape control characters

std::string result;

for (const auto c : token\_string)

{

if ('\x00' <= c and c <= '\x1F')

{

// escape control characters

std::array<char, 9> cs{{}};

(std::snprintf)(cs.data(), cs.size(), "<U+%.4X>", static\_cast<unsigned char>(c));

result += cs.data();

}

else

{

// add character as is

result.push\_back(c);

}

}

return result;

}

/// return syntax error message

JSON\_HEDLEY\_RETURNS\_NON\_NULL

constexpr const char\* get\_error\_message() const noexcept

{

return error\_message;

}

/////////////////////

// actual scanner

/////////////////////

/\*!

@brief skip the UTF-8 byte order mark

@return true iff there is no BOM or the correct BOM has been skipped

\*/

bool skip\_bom()

{

if (get() == 0xEF)

{

// check if we completely parse the BOM

return get() == 0xBB and get() == 0xBF;

}

// the first character is not the beginning of the BOM; unget it to

// process is later

unget();

return true;

}

token\_type scan()

{

// initially, skip the BOM

if (position.chars\_read\_total == 0 and not skip\_bom())

{

error\_message = "invalid BOM; must be 0xEF 0xBB 0xBF if given";

return token\_type::parse\_error;

}

// read next character and ignore whitespace

do

{

get();

}

while (current == ' ' or current == '\t' or current == '\n' or current == '\r');

switch (current)

{

// structural characters

case '[':

return token\_type::begin\_array;

case ']':

return token\_type::end\_array;

case '{':

return token\_type::begin\_object;

case '}':

return token\_type::end\_object;

case ':':

return token\_type::name\_separator;

case ',':

return token\_type::value\_separator;

// literals

case 't':

return scan\_literal("true", 4, token\_type::literal\_true);

case 'f':

return scan\_literal("false", 5, token\_type::literal\_false);

case 'n':

return scan\_literal("null", 4, token\_type::literal\_null);

// string

case '\"':

return scan\_string();

// number

case '-':

case '0':

case '1':

case '2':

case '3':

case '4':

case '5':

case '6':

case '7':

case '8':

case '9':

return scan\_number();

// end of input (the null byte is needed when parsing from

// string literals)

case '\0':

case std::char\_traits<char>::eof():

return token\_type::end\_of\_input;

// error

default:

error\_message = "invalid literal";

return token\_type::parse\_error;

}

}

private:

/// input adapter

detail::input\_adapter\_t ia = nullptr;

/// the current character

std::char\_traits<char>::int\_type current = std::char\_traits<char>::eof();

/// whether the next get() call should just return current

bool next\_unget = false;

/// the start position of the current token

position\_t position {};

/// raw input token string (for error messages)

std::vector<char> token\_string {};

/// buffer for variable-length tokens (numbers, strings)

string\_t token\_buffer {};

/// a description of occurred lexer errors

const char\* error\_message = "";

// number values

number\_integer\_t value\_integer = 0;

number\_unsigned\_t value\_unsigned = 0;

number\_float\_t value\_float = 0;

/// the decimal point

const char decimal\_point\_char = '.';

};

} // namespace detail

} // namespace nlohmann

// #include <nlohmann/detail/input/parser.hpp>

#include <cassert> // assert

#include <cmath> // isfinite

#include <cstdint> // uint8\_t

#include <functional> // function

#include <string> // string

#include <utility> // move

#include <vector> // vector

// #include <nlohmann/detail/exceptions.hpp>

// #include <nlohmann/detail/input/input\_adapters.hpp>

// #include <nlohmann/detail/input/json\_sax.hpp>

// #include <nlohmann/detail/input/lexer.hpp>

// #include <nlohmann/detail/macro\_scope.hpp>

// #include <nlohmann/detail/meta/is\_sax.hpp>

// #include <nlohmann/detail/value\_t.hpp>

namespace nlohmann

{

namespace detail

{

////////////

// parser //

////////////

/\*!

@brief syntax analysis

This class implements a recursive decent parser.

\*/

template<typename BasicJsonType>

class parser

{

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 lexer\_t = lexer<BasicJsonType>;

using token\_type = typename lexer\_t::token\_type;

public:

enum class parse\_event\_t : uint8\_t

{

/// the parser read `{` and started to process a JSON object

object\_start,

/// the parser read `}` and finished processing a JSON object

object\_end,

/// the parser read `[` and started to process a JSON array

array\_start,

/// the parser read `]` and finished processing a JSON array

array\_end,

/// the parser read a key of a value in an object

key,

/// the parser finished reading a JSON value

value

};

using parser\_callback\_t =

std::function<bool(int depth, parse\_event\_t event, BasicJsonType& parsed)>;

/// a parser reading from an input adapter

explicit parser(detail::input\_adapter\_t&& adapter,

const parser\_callback\_t cb = nullptr,

const bool allow\_exceptions\_ = true)

: callback(cb), m\_lexer(std::move(adapter)), allow\_exceptions(allow\_exceptions\_)

{

// read first token

get\_token();

}

/\*!

@brief public parser interface

@param[in] strict whether to expect the last token to be EOF

@param[in,out] result parsed JSON value

@throw parse\_error.101 in case of an unexpected token

@throw parse\_error.102 if to\_unicode fails or surrogate error

@throw parse\_error.103 if to\_unicode fails

\*/

void parse(const bool strict, BasicJsonType& result)

{

if (callback)

{

json\_sax\_dom\_callback\_parser<BasicJsonType> sdp(result, callback, allow\_exceptions);

sax\_parse\_internal(&sdp);

result.assert\_invariant();

// in strict mode, input must be completely read

if (strict and (get\_token() != token\_type::end\_of\_input))

{

sdp.parse\_error(m\_lexer.get\_position(),

m\_lexer.get\_token\_string(),

parse\_error::create(101, m\_lexer.get\_position(),

exception\_message(token\_type::end\_of\_input, "value")));

}

// in case of an error, return discarded value

if (sdp.is\_errored())

{

result = value\_t::discarded;

return;

}

// set top-level value to null if it was discarded by the callback

// function

if (result.is\_discarded())

{

result = nullptr;

}

}

else

{

json\_sax\_dom\_parser<BasicJsonType> sdp(result, allow\_exceptions);

sax\_parse\_internal(&sdp);

result.assert\_invariant();

// in strict mode, input must be completely read

if (strict and (get\_token() != token\_type::end\_of\_input))

{

sdp.parse\_error(m\_lexer.get\_position(),

m\_lexer.get\_token\_string(),

parse\_error::create(101, m\_lexer.get\_position(),

exception\_message(token\_type::end\_of\_input, "value")));

}

// in case of an error, return discarded value

if (sdp.is\_errored())

{

result = value\_t::discarded;

return;

}

}

}

/\*!

@brief public accept interface

@param[in] strict whether to expect the last token to be EOF

@return whether the input is a proper JSON text

\*/

bool accept(const bool strict = true)

{

json\_sax\_acceptor<BasicJsonType> sax\_acceptor;

return sax\_parse(&sax\_acceptor, strict);

}

template <typename SAX>

JSON\_HEDLEY\_NON\_NULL(2)

bool sax\_parse(SAX\* sax, const bool strict = true)

{

(void)detail::is\_sax\_static\_asserts<SAX, BasicJsonType> {};

const bool result = sax\_parse\_internal(sax);

// strict mode: next byte must be EOF

if (result and strict and (get\_token() != token\_type::end\_of\_input))

{

return sax->parse\_error(m\_lexer.get\_position(),

m\_lexer.get\_token\_string(),

parse\_error::create(101, m\_lexer.get\_position(),

exception\_message(token\_type::end\_of\_input, "value")));

}

return result;

}

private:

template <typename SAX>

JSON\_HEDLEY\_NON\_NULL(2)

bool sax\_parse\_internal(SAX\* sax)

{

// stack to remember the hierarchy of structured values we are parsing

// true = array; false = object

std::vector<bool> states;

// value to avoid a goto (see comment where set to true)

bool skip\_to\_state\_evaluation = false;

while (true)

{

if (not skip\_to\_state\_evaluation)

{

// invariant: get\_token() was called before each iteration

switch (last\_token)

{

case token\_type::begin\_object:

{

if (JSON\_HEDLEY\_UNLIKELY(not sax->start\_object(std::size\_t(-1))))

{

return false;

}

// closing } -> we are done

if (get\_token() == token\_type::end\_object)

{

if (JSON\_HEDLEY\_UNLIKELY(not sax->end\_object()))

{

return false;

}

break;

}

// parse key

if (JSON\_HEDLEY\_UNLIKELY(last\_token != token\_type::value\_string))

{

return sax->parse\_error(m\_lexer.get\_position(),

m\_lexer.get\_token\_string(),

parse\_error::create(101, m\_lexer.get\_position(),

exception\_message(token\_type::value\_string, "object key")));

}

if (JSON\_HEDLEY\_UNLIKELY(not sax->key(m\_lexer.get\_string())))

{

return false;

}

// parse separator (:)

if (JSON\_HEDLEY\_UNLIKELY(get\_token() != token\_type::name\_separator))

{

return sax->parse\_error(m\_lexer.get\_position(),

m\_lexer.get\_token\_string(),

parse\_error::create(101, m\_lexer.get\_position(),

exception\_message(token\_type::name\_separator, "object separator")));

}

// remember we are now inside an object

states.push\_back(false);

// parse values

get\_token();

continue;

}

case token\_type::begin\_array:

{

if (JSON\_HEDLEY\_UNLIKELY(not sax->start\_array(std::size\_t(-1))))

{

return false;

}

// closing ] -> we are done

if (get\_token() == token\_type::end\_array)

{

if (JSON\_HEDLEY\_UNLIKELY(not sax->end\_array()))

{

return false;

}

break;

}

// remember we are now inside an array

states.push\_back(true);

// parse values (no need to call get\_token)

continue;

}

case token\_type::value\_float:

{

const auto res = m\_lexer.get\_number\_float();

if (JSON\_HEDLEY\_UNLIKELY(not std::isfinite(res)))

{

return sax->parse\_error(m\_lexer.get\_position(),

m\_lexer.get\_token\_string(),

out\_of\_range::create(406, "number overflow parsing '" + m\_lexer.get\_token\_string() + "'"));

}

if (JSON\_HEDLEY\_UNLIKELY(not sax->number\_float(res, m\_lexer.get\_string())))

{

return false;

}

break;

}

case token\_type::literal\_false:

{

if (JSON\_HEDLEY\_UNLIKELY(not sax->boolean(false)))

{

return false;

}

break;

}

case token\_type::literal\_null:

{

if (JSON\_HEDLEY\_UNLIKELY(not sax->null()))

{

return false;

}

break;

}

case token\_type::literal\_true:

{

if (JSON\_HEDLEY\_UNLIKELY(not sax->boolean(true)))

{

return false;

}

break;

}

case token\_type::value\_integer:

{

if (JSON\_HEDLEY\_UNLIKELY(not sax->number\_integer(m\_lexer.get\_number\_integer())))

{

return false;

}

break;

}

case token\_type::value\_string:

{

if (JSON\_HEDLEY\_UNLIKELY(not sax->string(m\_lexer.get\_string())))

{

return false;

}

break;

}

case token\_type::value\_unsigned:

{

if (JSON\_HEDLEY\_UNLIKELY(not sax->number\_unsigned(m\_lexer.get\_number\_unsigned())))

{

return false;

}

break;

}

case token\_type::parse\_error:

{

// using "uninitialized" to avoid "expected" message

return sax->parse\_error(m\_lexer.get\_position(),

m\_lexer.get\_token\_string(),

parse\_error::create(101, m\_lexer.get\_position(),

exception\_message(token\_type::uninitialized, "value")));

}

default: // the last token was unexpected

{

return sax->parse\_error(m\_lexer.get\_position(),

m\_lexer.get\_token\_string(),

parse\_error::create(101, m\_lexer.get\_position(),

exception\_message(token\_type::literal\_or\_value, "value")));

}

}

}

else

{

skip\_to\_state\_evaluation = false;

}

// we reached this line after we successfully parsed a value

if (states.empty())

{

// empty stack: we reached the end of the hierarchy: done

return true;

}

if (states.back()) // array

{

// comma -> next value

if (get\_token() == token\_type::value\_separator)

{

// parse a new value

get\_token();

continue;

}

// closing ]

if (JSON\_HEDLEY\_LIKELY(last\_token == token\_type::end\_array))

{

if (JSON\_HEDLEY\_UNLIKELY(not sax->end\_array()))

{

return false;

}

// We are done with this array. Before we can parse a

// new value, we need to evaluate the new state first.

// By setting skip\_to\_state\_evaluation to false, we

// are effectively jumping to the beginning of this if.

assert(not states.empty());

states.pop\_back();

skip\_to\_state\_evaluation = true;

continue;

}

return sax->parse\_error(m\_lexer.get\_position(),

m\_lexer.get\_token\_string(),

parse\_error::create(101, m\_lexer.get\_position(),

exception\_message(token\_type::end\_array, "array")));

}

else // object

{

// comma -> next value

if (get\_token() == token\_type::value\_separator)

{

// parse key

if (JSON\_HEDLEY\_UNLIKELY(get\_token() != token\_type::value\_string))

{

return sax->parse\_error(m\_lexer.get\_position(),

m\_lexer.get\_token\_string(),

parse\_error::create(101, m\_lexer.get\_position(),

exception\_message(token\_type::value\_string, "object key")));

}

if (JSON\_HEDLEY\_UNLIKELY(not sax->key(m\_lexer.get\_string())))

{

return false;

}

// parse separator (:)

if (JSON\_HEDLEY\_UNLIKELY(get\_token() != token\_type::name\_separator))

{

return sax->parse\_error(m\_lexer.get\_position(),

m\_lexer.get\_token\_string(),

parse\_error::create(101, m\_lexer.get\_position(),

exception\_message(token\_type::name\_separator, "object separator")));

}

// parse values

get\_token();

continue;

}

// closing }

if (JSON\_HEDLEY\_LIKELY(last\_token == token\_type::end\_object))

{

if (JSON\_HEDLEY\_UNLIKELY(not sax->end\_object()))

{

return false;

}

// We are done with this object. Before we can parse a

// new value, we need to evaluate the new state first.

// By setting skip\_to\_state\_evaluation to false, we

// are effectively jumping to the beginning of this if.

assert(not states.empty());

states.pop\_back();

skip\_to\_state\_evaluation = true;

continue;

}

return sax->parse\_error(m\_lexer.get\_position(),

m\_lexer.get\_token\_string(),

parse\_error::create(101, m\_lexer.get\_position(),

exception\_message(token\_type::end\_object, "object")));

}

}

}

/// get next token from lexer

token\_type get\_token()

{

return last\_token = m\_lexer.scan();

}

std::string exception\_message(const token\_type expected, const std::string& context)

{

std::string error\_msg = "syntax error ";

if (not context.empty())

{

error\_msg += "while parsing " + context + " ";

}

error\_msg += "- ";

if (last\_token == token\_type::parse\_error)

{

error\_msg += std::string(m\_lexer.get\_error\_message()) + "; last read: '" +

m\_lexer.get\_token\_string() + "'";

}

else

{

error\_msg += "unexpected " + std::string(lexer\_t::token\_type\_name(last\_token));

}

if (expected != token\_type::uninitialized)

{

error\_msg += "; expected " + std::string(lexer\_t::token\_type\_name(expected));

}

return error\_msg;

}

private:

/// callback function

const parser\_callback\_t callback = nullptr;

/// the type of the last read token

token\_type last\_token = token\_type::uninitialized;

/// the lexer

lexer\_t m\_lexer;

/// whether to throw exceptions in case of errors

const bool allow\_exceptions = true;

};

} // namespace detail

} // namespace nlohmann

// #include <nlohmann/detail/iterators/internal\_iterator.hpp>

// #include <nlohmann/detail/iterators/primitive\_iterator.hpp>

#include <cstddef> // ptrdiff\_t

#include <limits> // numeric\_limits

namespace nlohmann

{

namespace detail

{

/\*

@brief an iterator for primitive JSON types

This class models an iterator for primitive JSON types (boolean, number,

string). It's only purpose is to allow the iterator/const\_iterator classes

to "iterate" over primitive values. Internally, the iterator is modeled by

a `difference\_type` variable. Value begin\_value (`0`) models the begin,

end\_value (`1`) models past the end.

\*/

class primitive\_iterator\_t

{

private:

using difference\_type = std::ptrdiff\_t;

static constexpr difference\_type begin\_value = 0;

static constexpr difference\_type end\_value = begin\_value + 1;

/// iterator as signed integer type

difference\_type m\_it = (std::numeric\_limits<std::ptrdiff\_t>::min)();

public:

constexpr difference\_type get\_value() const noexcept

{

return m\_it;

}

/// set iterator to a defined beginning

void set\_begin() noexcept

{

m\_it = begin\_value;

}

/// set iterator to a defined past the end

void set\_end() noexcept

{

m\_it = end\_value;

}

/// return whether the iterator can be dereferenced

constexpr bool is\_begin() const noexcept

{

return m\_it == begin\_value;

}

/// return whether the iterator is at end

constexpr bool is\_end() const noexcept

{

return m\_it == end\_value;

}

friend constexpr bool operator==(primitive\_iterator\_t lhs, primitive\_iterator\_t rhs) noexcept

{

return lhs.m\_it == rhs.m\_it;

}

friend constexpr bool operator<(primitive\_iterator\_t lhs, primitive\_iterator\_t rhs) noexcept

{

return lhs.m\_it < rhs.m\_it;

}

primitive\_iterator\_t operator+(difference\_type n) noexcept

{

auto result = \*this;

result += n;

return result;

}

friend constexpr difference\_type operator-(primitive\_iterator\_t lhs, primitive\_iterator\_t rhs) noexcept

{

return lhs.m\_it - rhs.m\_it;

}

primitive\_iterator\_t& operator++() noexcept

{

++m\_it;

return \*this;

}

primitive\_iterator\_t const operator++(int) noexcept

{

auto result = \*this;

++m\_it;

return result;

}

primitive\_iterator\_t& operator--() noexcept

{

--m\_it;

return \*this;

}

primitive\_iterator\_t const operator--(int) noexcept

{

auto result = \*this;

--m\_it;

return result;

}

primitive\_iterator\_t& operator+=(difference\_type n) noexcept

{

m\_it += n;

return \*this;

}

primitive\_iterator\_t& operator-=(difference\_type n) noexcept

{

m\_it -= n;

return \*this;

}

};

} // namespace detail

} // namespace nlohmann

namespace nlohmann

{

namespace detail

{

/\*!

@brief 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.

\*/

template<typename BasicJsonType> struct internal\_iterator

{

/// iterator for JSON objects

typename BasicJsonType::object\_t::iterator object\_iterator {};

/// iterator for JSON arrays

typename BasicJsonType::array\_t::iterator array\_iterator {};

/// generic iterator for all other types

primitive\_iterator\_t primitive\_iterator {};

};

} // namespace detail

} // namespace nlohmann

// #include <nlohmann/detail/iterators/iter\_impl.hpp>

#include <ciso646> // not

#include <iterator> // iterator, random\_access\_iterator\_tag, bidirectional\_iterator\_tag, advance, next

#include <type\_traits> // conditional, is\_const, remove\_const

// #include <nlohmann/detail/exceptions.hpp>

// #include <nlohmann/detail/iterators/internal\_iterator.hpp>

// #include <nlohmann/detail/iterators/primitive\_iterator.hpp>

// #include <nlohmann/detail/macro\_scope.hpp>

// #include <nlohmann/detail/meta/cpp\_future.hpp>

// #include <nlohmann/detail/meta/type\_traits.hpp>

// #include <nlohmann/detail/value\_t.hpp>

namespace nlohmann

{

namespace detail

{

// forward declare, to be able to friend it later on

template<typename IteratorType> class iteration\_proxy;

template<typename IteratorType> class iteration\_proxy\_value;

/\*!

@brief a template for a bidirectional iterator for the @ref basic\_json class

This class implements a both iterators (iterator and const\_iterator) for the

@ref basic\_json class.

@note An iterator is called \*initialized\* when a pointer to a JSON value has

been set (e.g., by a constructor or a copy assignment). If the iterator is

default-constructed, it is \*uninitialized\* and most methods are undefined.

\*\*The library uses assertions to detect calls on uninitialized iterators.\*\*

@requirement The class satisfies the following concept requirements:

-

[BidirectionalIterator](https://en.cppreference.com/w/cpp/named\_req/BidirectionalIterator):

The iterator that can be moved can be moved in both directions (i.e.

incremented and decremented).

@since version 1.0.0, simplified in version 2.0.9, change to bidirectional

iterators in version 3.0.0 (see https://github.com/nlohmann/json/issues/593)

\*/

template<typename BasicJsonType>

class iter\_impl

{

/// allow basic\_json to access private members

friend iter\_impl<typename std::conditional<std::is\_const<BasicJsonType>::value, typename std::remove\_const<BasicJsonType>::type, const BasicJsonType>::type>;

friend BasicJsonType;

friend iteration\_proxy<iter\_impl>;

friend iteration\_proxy\_value<iter\_impl>;

using object\_t = typename BasicJsonType::object\_t;

using array\_t = typename BasicJsonType::array\_t;

// make sure BasicJsonType is basic\_json or const basic\_json

static\_assert(is\_basic\_json<typename std::remove\_const<BasicJsonType>::type>::value,

"iter\_impl only accepts (const) basic\_json");

public:

/// The std::iterator class template (used as a base class to provide typedefs) is deprecated in C++17.

/// The C++ Standard has never required user-defined iterators to derive from std::iterator.

/// A user-defined iterator should provide publicly accessible typedefs named

/// iterator\_category, value\_type, difference\_type, pointer, and reference.

/// Note that value\_type is required to be non-const, even for constant iterators.

using iterator\_category = std::bidirectional\_iterator\_tag;

/// the type of the values when the iterator is dereferenced

using value\_type = typename BasicJsonType::value\_type;

/// a type to represent differences between iterators

using difference\_type = typename BasicJsonType::difference\_type;

/// defines a pointer to the type iterated over (value\_type)

using pointer = typename std::conditional<std::is\_const<BasicJsonType>::value,

typename BasicJsonType::const\_pointer,

typename BasicJsonType::pointer>::type;

/// defines a reference to the type iterated over (value\_type)

using reference =

typename std::conditional<std::is\_const<BasicJsonType>::value,

typename BasicJsonType::const\_reference,

typename BasicJsonType::reference>::type;

/// default constructor

iter\_impl() = default;

/\*!

@brief constructor for a given JSON instance

@param[in] object pointer to a JSON object for this iterator

@pre object != nullptr

@post The iterator is initialized; i.e. `m\_object != nullptr`.

\*/

explicit iter\_impl(pointer object) noexcept : m\_object(object)

{

assert(m\_object != nullptr);

switch (m\_object->m\_type)

{

case value\_t::object:

{

m\_it.object\_iterator = typename object\_t::iterator();

break;

}

case value\_t::array:

{

m\_it.array\_iterator = typename array\_t::iterator();

break;

}

default:

{

m\_it.primitive\_iterator = primitive\_iterator\_t();

break;

}

}

}

/\*!

@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.

\*/

/\*!

@brief const copy constructor

@param[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

\*/

iter\_impl(const iter\_impl<const BasicJsonType>& other) noexcept

: m\_object(other.m\_object), m\_it(other.m\_it)

{}

/\*!

@brief converting assignment

@param[in] other const iterator to copy from

@return const/non-const iterator

@note It is not checked whether @a other is initialized.

\*/

iter\_impl& operator=(const iter\_impl<const BasicJsonType>& other) noexcept

{

m\_object = other.m\_object;

m\_it = other.m\_it;

return \*this;

}

/\*!

@brief converting constructor

@param[in] other non-const iterator to copy from

@note It is not checked whether @a other is initialized.

\*/

iter\_impl(const iter\_impl<typename std::remove\_const<BasicJsonType>::type>& other) noexcept

: m\_object(other.m\_object), m\_it(other.m\_it)

{}

/\*!

@brief converting assignment

@param[in] other non-const iterator to copy from

@return const/non-const iterator

@note It is not checked whether @a other is initialized.

\*/

iter\_impl& operator=(const iter\_impl<typename std::remove\_const<BasicJsonType>::type>& other) noexcept

{

m\_object = other.m\_object;

m\_it = other.m\_it;

return \*this;

}

private:

/\*!

@brief set the iterator to the first value

@pre The iterator is initialized; i.e. `m\_object != nullptr`.

\*/

void set\_begin() noexcept

{

assert(m\_object != nullptr);

switch (m\_object->m\_type)

{

case value\_t::object:

{

m\_it.object\_iterator = m\_object->m\_value.object->begin();

break;

}

case value\_t::array:

{

m\_it.array\_iterator = m\_object->m\_value.array->begin();

break;

}

case value\_t::null:

{

// set to end so begin()==end() is true: null is empty

m\_it.primitive\_iterator.set\_end();

break;

}

default:

{

m\_it.primitive\_iterator.set\_begin();

break;

}

}

}

/\*!

@brief set the iterator past the last value

@pre The iterator is initialized; i.e. `m\_object != nullptr`.

\*/

void set\_end() noexcept

{

assert(m\_object != nullptr);

switch (m\_object->m\_type)

{

case value\_t::object:

{

m\_it.object\_iterator = m\_object->m\_value.object->end();

break;

}

case value\_t::array:

{

m\_it.array\_iterator = m\_object->m\_value.array->end();

break;

}

default:

{

m\_it.primitive\_iterator.set\_end();

break;

}

}

}

public:

/\*!

@brief return a reference to the value pointed to by the iterator

@pre The iterator is initialized; i.e. `m\_object != nullptr`.

\*/

reference operator\*() const

{

assert(m\_object != nullptr);

switch (m\_object->m\_type)

{

case value\_t::object:

{

assert(m\_it.object\_iterator != m\_object->m\_value.object->end());

return m\_it.object\_iterator->second;

}

case value\_t::array:

{

assert(m\_it.array\_iterator != m\_object->m\_value.array->end());

return \*m\_it.array\_iterator;

}

case value\_t::null:

JSON\_THROW(invalid\_iterator::create(214, "cannot get value"));

default:

{

if (JSON\_HEDLEY\_LIKELY(m\_it.primitive\_iterator.is\_begin()))

{

return \*m\_object;

}

JSON\_THROW(invalid\_iterator::create(214, "cannot get value"));

}

}

}

/\*!

@brief dereference the iterator

@pre The iterator is initialized; i.e. `m\_object != nullptr`.

\*/

pointer operator->() const

{

assert(m\_object != nullptr);

switch (m\_object->m\_type)

{

case value\_t::object:

{

assert(m\_it.object\_iterator != m\_object->m\_value.object->end());

return &(m\_it.object\_iterator->second);

}

case value\_t::array:

{

assert(m\_it.array\_iterator != m\_object->m\_value.array->end());

return &\*m\_it.array\_iterator;

}

default:

{

if (JSON\_HEDLEY\_LIKELY(m\_it.primitive\_iterator.is\_begin()))

{

return m\_object;

}

JSON\_THROW(invalid\_iterator::create(214, "cannot get value"));

}

}

}

/\*!

@brief post-increment (it++)

@pre The iterator is initialized; i.e. `m\_object != nullptr`.

\*/

iter\_impl const operator++(int)

{

auto result = \*this;

++(\*this);

return result;

}

/\*!

@brief pre-increment (++it)

@pre The iterator is initialized; i.e. `m\_object != nullptr`.

\*/

iter\_impl& operator++()

{

assert(m\_object != nullptr);

switch (m\_object->m\_type)

{

case value\_t::object:

{

std::advance(m\_it.object\_iterator, 1);

break;

}

case value\_t::array:

{

std::advance(m\_it.array\_iterator, 1);

break;

}

default:

{

++m\_it.primitive\_iterator;

break;

}

}

return \*this;

}

/\*!

@brief post-decrement (it--)

@pre The iterator is initialized; i.e. `m\_object != nullptr`.

\*/

iter\_impl const operator--(int)

{

auto result = \*this;

--(\*this);

return result;

}

/\*!

@brief pre-decrement (--it)

@pre The iterator is initialized; i.e. `m\_object != nullptr`.

\*/

iter\_impl& operator--()

{

assert(m\_object != nullptr);

switch (m\_object->m\_type)

{

case value\_t::object:

{

std::advance(m\_it.object\_iterator, -1);

break;

}

case value\_t::array:

{

std::advance(m\_it.array\_iterator, -1);

break;

}

default:

{

--m\_it.primitive\_iterator;

break;

}

}

return \*this;

}

/\*!

@brief comparison: equal

@pre The iterator is initialized; i.e. `m\_object != nullptr`.

\*/

bool operator==(const iter\_impl& other) const

{

// if objects are not the same, the comparison is undefined

if (JSON\_HEDLEY\_UNLIKELY(m\_object != other.m\_object))

{

JSON\_THROW(invalid\_iterator::create(212, "cannot compare iterators of different containers"));

}

assert(m\_object != nullptr);

switch (m\_object->m\_type)

{

case value\_t::object:

return (m\_it.object\_iterator == other.m\_it.object\_iterator);

case value\_t::array:

return (m\_it.array\_iterator == other.m\_it.array\_iterator);

default:

return (m\_it.primitive\_iterator == other.m\_it.primitive\_iterator);

}

}

/\*!

@brief comparison: not equal

@pre The iterator is initialized; i.e. `m\_object != nullptr`.

\*/

bool operator!=(const iter\_impl& other) const

{

return not operator==(other);

}

/\*!

@brief comparison: smaller

@pre The iterator is initialized; i.e. `m\_object != nullptr`.

\*/

bool operator<(const iter\_impl& other) const

{

// if objects are not the same, the comparison is undefined

if (JSON\_HEDLEY\_UNLIKELY(m\_object != other.m\_object))

{

JSON\_THROW(invalid\_iterator::create(212, "cannot compare iterators of different containers"));

}

assert(m\_object != nullptr);

switch (m\_object->m\_type)

{

case value\_t::object:

JSON\_THROW(invalid\_iterator::create(213, "cannot compare order of object iterators"));

case value\_t::array:

return (m\_it.array\_iterator < other.m\_it.array\_iterator);

default:

return (m\_it.primitive\_iterator < other.m\_it.primitive\_iterator);

}

}

/\*!

@brief comparison: less than or equal

@pre The iterator is initialized; i.e. `m\_object != nullptr`.

\*/

bool operator<=(const iter\_impl& other) const

{

return not other.operator < (\*this);

}

/\*!

@brief comparison: greater than

@pre The iterator is initialized; i.e. `m\_object != nullptr`.

\*/

bool operator>(const iter\_impl& other) const

{

return not operator<=(other);

}

/\*!

@brief comparison: greater than or equal

@pre The iterator is initialized; i.e. `m\_object != nullptr`.

\*/

bool operator>=(const iter\_impl& other) const

{

return not operator<(other);

}

/\*!

@brief add to iterator

@pre The iterator is initialized; i.e. `m\_object != nullptr`.

\*/

iter\_impl& operator+=(difference\_type i)

{

assert(m\_object != nullptr);

switch (m\_object->m\_type)

{

case value\_t::object:

JSON\_THROW(invalid\_iterator::create(209, "cannot use offsets with object iterators"));

case value\_t::array:

{

std::advance(m\_it.array\_iterator, i);

break;

}

default:

{

m\_it.primitive\_iterator += i;

break;

}

}

return \*this;

}

/\*!

@brief subtract from iterator

@pre The iterator is initialized; i.e. `m\_object != nullptr`.

\*/

iter\_impl& operator-=(difference\_type i)

{

return operator+=(-i);

}

/\*!

@brief add to iterator

@pre The iterator is initialized; i.e. `m\_object != nullptr`.

\*/

iter\_impl operator+(difference\_type i) const

{

auto result = \*this;

result += i;

return result;

}

/\*!

@brief addition of distance and iterator

@pre The iterator is initialized; i.e. `m\_object != nullptr`.

\*/

friend iter\_impl operator+(difference\_type i, const iter\_impl& it)

{

auto result = it;

result += i;

return result;

}

/\*!

@brief subtract from iterator

@pre The iterator is initialized; i.e. `m\_object != nullptr`.

\*/

iter\_impl operator-(difference\_type i) const

{

auto result = \*this;

result -= i;

return result;

}

/\*!

@brief return difference

@pre The iterator is initialized; i.e. `m\_object != nullptr`.

\*/

difference\_type operator-(const iter\_impl& other) const

{

assert(m\_object != nullptr);

switch (m\_object->m\_type)

{

case value\_t::object:

JSON\_THROW(invalid\_iterator::create(209, "cannot use offsets with object iterators"));

case value\_t::array:

return m\_it.array\_iterator - other.m\_it.array\_iterator;

default:

return m\_it.primitive\_iterator - other.m\_it.primitive\_iterator;

}

}

/\*!

@brief access to successor

@pre The iterator is initialized; i.e. `m\_object != nullptr`.

\*/

reference operator[](difference\_type n) const

{

assert(m\_object != nullptr);

switch (m\_object->m\_type)

{

case value\_t::object:

JSON\_THROW(invalid\_iterator::create(208, "cannot use operator[] for object iterators"));

case value\_t::array:

return \*std::next(m\_it.array\_iterator, n);

case value\_t::null:

JSON\_THROW(invalid\_iterator::create(214, "cannot get value"));

default:

{

if (JSON\_HEDLEY\_LIKELY(m\_it.primitive\_iterator.get\_value() == -n))

{

return \*m\_object;

}

JSON\_THROW(invalid\_iterator::create(214, "cannot get value"));

}

}

}

/\*!

@brief return the key of an object iterator

@pre The iterator is initialized; i.e. `m\_object != nullptr`.

\*/

const typename object\_t::key\_type& key() const

{

assert(m\_object != nullptr);

if (JSON\_HEDLEY\_LIKELY(m\_object->is\_object()))

{

return m\_it.object\_iterator->first;

}

JSON\_THROW(invalid\_iterator::create(207, "cannot use key() for non-object iterators"));

}

/\*!

@brief return the value of an iterator

@pre The iterator is initialized; i.e. `m\_object != nullptr`.

\*/

reference value() const

{

return operator\*();

}

private:

/// associated JSON instance

pointer m\_object = nullptr;

/// the actual iterator of the associated instance

internal\_iterator<typename std::remove\_const<BasicJsonType>::type> m\_it {};

};

} // namespace detail

} // namespace nlohmann

// #include <nlohmann/detail/iterators/iteration\_proxy.hpp>

// #include <nlohmann/detail/iterators/json\_reverse\_iterator.hpp>

#include <cstddef> // ptrdiff\_t

#include <iterator> // reverse\_iterator

#include <utility> // declval

namespace nlohmann

{

namespace detail

{

//////////////////////

// reverse\_iterator //

//////////////////////

/\*!

@brief a template for a reverse iterator class

@tparam Base the base iterator type to reverse. Valid types are @ref

iterator (to create @ref reverse\_iterator) and @ref const\_iterator (to

create @ref const\_reverse\_iterator).

@requirement The class satisfies the following concept requirements:

-

[BidirectionalIterator](https://en.cppreference.com/w/cpp/named\_req/BidirectionalIterator):

The iterator that can be moved can be moved in both directions (i.e.

incremented and decremented).

- [OutputIterator](https://en.cppreference.com/w/cpp/named\_req/OutputIterator):

It is possible to write to the pointed-to element (only if @a Base is

@ref iterator).

@since version 1.0.0

\*/

template<typename Base>

class json\_reverse\_iterator : public std::reverse\_iterator<Base>

{

public:

using difference\_type = std::ptrdiff\_t;

/// shortcut to the reverse iterator adapter

using base\_iterator = std::reverse\_iterator<Base>;

/// the reference type for the pointed-to element

using reference = typename Base::reference;

/// create reverse iterator from iterator

explicit json\_reverse\_iterator(const typename base\_iterator::iterator\_type& it) noexcept

: base\_iterator(it) {}

/// create reverse iterator from base class

explicit json\_reverse\_iterator(const base\_iterator& it) noexcept : base\_iterator(it) {}

/// post-increment (it++)

json\_reverse\_iterator const operator++(int)

{

return static\_cast<json\_reverse\_iterator>(base\_iterator::operator++(1));

}

/// pre-increment (++it)

json\_reverse\_iterator& operator++()

{

return static\_cast<json\_reverse\_iterator&>(base\_iterator::operator++());

}

/// post-decrement (it--)

json\_reverse\_iterator const operator--(int)

{

return static\_cast<json\_reverse\_iterator>(base\_iterator::operator--(1));

}

/// pre-decrement (--it)

json\_reverse\_iterator& operator--()

{

return static\_cast<json\_reverse\_iterator&>(base\_iterator::operator--());

}

/// add to iterator

json\_reverse\_iterator& operator+=(difference\_type i)

{

return static\_cast<json\_reverse\_iterator&>(base\_iterator::operator+=(i));

}

/// add to iterator

json\_reverse\_iterator operator+(difference\_type i) const

{

return static\_cast<json\_reverse\_iterator>(base\_iterator::operator+(i));

}

/// subtract from iterator

json\_reverse\_iterator operator-(difference\_type i) const

{

return static\_cast<json\_reverse\_iterator>(base\_iterator::operator-(i));

}

/// return difference

difference\_type operator-(const json\_reverse\_iterator& other) const

{

return base\_iterator(\*this) - base\_iterator(other);

}

/// access to successor

reference operator[](difference\_type n) const

{

return \*(this->operator+(n));

}

/// return the key of an object iterator

auto key() const -> decltype(std::declval<Base>().key())

{

auto it = --this->base();

return it.key();

}

/// return the value of an iterator

reference value() const

{

auto it = --this->base();

return it.operator \* ();

}

};

} // namespace detail

} // namespace nlohmann

// #include <nlohmann/detail/iterators/primitive\_iterator.hpp>

// #include <nlohmann/detail/json\_pointer.hpp>

#include <algorithm> // all\_of

#include <cassert> // assert

#include <cctype> // isdigit

#include <numeric> // accumulate

#include <string> // string

#include <utility> // move

#include <vector> // vector

// #include <nlohmann/detail/exceptions.hpp>

// #include <nlohmann/detail/macro\_scope.hpp>

// #include <nlohmann/detail/value\_t.hpp>

namespace nlohmann

{

template<typename BasicJsonType>

class json\_pointer

{

// allow basic\_json to access private members

NLOHMANN\_BASIC\_JSON\_TPL\_DECLARATION

friend class basic\_json;

public:

/\*!

@brief create JSON pointer

Create a JSON pointer according to the syntax described in

[Section 3 of RFC6901](https://tools.ietf.org/html/rfc6901#section-3).

@param[in] s string representing the JSON pointer; if omitted, the empty

string is assumed which references the whole JSON value

@throw parse\_error.107 if the given JSON pointer @a s is nonempty and does

not begin with a slash (`/`); see example below

@throw parse\_error.108 if a tilde (`~`) in the given JSON pointer @a s is

not followed by `0` (representing `~`) or `1` (representing `/`); see

example below

@liveexample{The example shows the construction several valid JSON pointers

as well as the exceptional behavior.,json\_pointer}

@since version 2.0.0

\*/

explicit json\_pointer(const std::string& s = "")

: reference\_tokens(split(s))

{}

/\*!

@brief return a string representation of the JSON pointer

@invariant For each JSON pointer `ptr`, it holds:

@code {.cpp}

ptr == json\_pointer(ptr.to\_string());

@endcode

@return a string representation of the JSON pointer

@liveexample{The example shows the result of `to\_string`.,json\_pointer\_\_to\_string}

@since version 2.0.0

\*/

std::string to\_string() const

{

return std::accumulate(reference\_tokens.begin(), reference\_tokens.end(),

std::string{},

[](const std::string & a, const std::string & b)

{

return a + "/" + escape(b);

});

}

/// @copydoc to\_string()

operator std::string() const

{

return to\_string();

}

/\*!

@brief append another JSON pointer at the end of this JSON pointer

@param[in] ptr JSON pointer to append

@return JSON pointer with @a ptr appended

@liveexample{The example shows the usage of `operator/=`.,json\_pointer\_\_operator\_add}

@complexity Linear in the length of @a ptr.

@sa @ref operator/=(std::string) to append a reference token

@sa @ref operator/=(std::size\_t) to append an array index

@sa @ref operator/(const json\_pointer&, const json\_pointer&) for a binary operator

@since version 3.6.0

\*/

json\_pointer& operator/=(const json\_pointer& ptr)

{

reference\_tokens.insert(reference\_tokens.end(),

ptr.reference\_tokens.begin(),

ptr.reference\_tokens.end());

return \*this;

}

/\*!

@brief append an unescaped reference token at the end of this JSON pointer

@param[in] token reference token to append

@return JSON pointer with @a token appended without escaping @a token

@liveexample{The example shows the usage of `operator/=`.,json\_pointer\_\_operator\_add}

@complexity Amortized constant.

@sa @ref operator/=(const json\_pointer&) to append a JSON pointer

@sa @ref operator/=(std::size\_t) to append an array index

@sa @ref operator/(const json\_pointer&, std::size\_t) for a binary operator

@since version 3.6.0

\*/

json\_pointer& operator/=(std::string token)

{

push\_back(std::move(token));

return \*this;

}

/\*!

@brief append an array index at the end of this JSON pointer

@param[in] array\_index array index to append

@return JSON pointer with @a array\_index appended

@liveexample{The example shows the usage of `operator/=`.,json\_pointer\_\_operator\_add}

@complexity Amortized constant.

@sa @ref operator/=(const json\_pointer&) to append a JSON pointer

@sa @ref operator/=(std::string) to append a reference token

@sa @ref operator/(const json\_pointer&, std::string) for a binary operator

@since version 3.6.0

\*/

json\_pointer& operator/=(std::size\_t array\_index)

{

return \*this /= std::to\_string(array\_index);

}

/\*!

@brief create a new JSON pointer by appending the right JSON pointer at the end of the left JSON pointer

@param[in] lhs JSON pointer

@param[in] rhs JSON pointer

@return a new JSON pointer with @a rhs appended to @a lhs

@liveexample{The example shows the usage of `operator/`.,json\_pointer\_\_operator\_add\_binary}

@complexity Linear in the length of @a lhs and @a rhs.

@sa @ref operator/=(const json\_pointer&) to append a JSON pointer

@since version 3.6.0

\*/

friend json\_pointer operator/(const json\_pointer& lhs,

const json\_pointer& rhs)

{

return json\_pointer(lhs) /= rhs;

}

/\*!

@brief create a new JSON pointer by appending the unescaped token at the end of the JSON pointer

@param[in] ptr JSON pointer

@param[in] token reference token

@return a new JSON pointer with unescaped @a token appended to @a ptr

@liveexample{The example shows the usage of `operator/`.,json\_pointer\_\_operator\_add\_binary}

@complexity Linear in the length of @a ptr.

@sa @ref operator/=(std::string) to append a reference token

@since version 3.6.0

\*/

friend json\_pointer operator/(const json\_pointer& ptr, std::string token)

{

return json\_pointer(ptr) /= std::move(token);

}

/\*!

@brief create a new JSON pointer by appending the array-index-token at the end of the JSON pointer

@param[in] ptr JSON pointer

@param[in] array\_index array index

@return a new JSON pointer with @a array\_index appended to @a ptr

@liveexample{The example shows the usage of `operator/`.,json\_pointer\_\_operator\_add\_binary}

@complexity Linear in the length of @a ptr.

@sa @ref operator/=(std::size\_t) to append an array index

@since version 3.6.0

\*/

friend json\_pointer operator/(const json\_pointer& ptr, std::size\_t array\_index)

{

return json\_pointer(ptr) /= array\_index;

}

/\*!

@brief returns the parent of this JSON pointer

@return parent of this JSON pointer; in case this JSON pointer is the root,

the root itself is returned

@complexity Linear in the length of the JSON pointer.

@liveexample{The example shows the result of `parent\_pointer` for different

JSON Pointers.,json\_pointer\_\_parent\_pointer}

@since version 3.6.0

\*/

json\_pointer parent\_pointer() const

{

if (empty())

{

return \*this;

}

json\_pointer res = \*this;

res.pop\_back();

return res;

}

/\*!

@brief remove last reference token

@pre not `empty()`

@liveexample{The example shows the usage of `pop\_back`.,json\_pointer\_\_pop\_back}

@complexity Constant.

@throw out\_of\_range.405 if JSON pointer has no parent

@since version 3.6.0

\*/

void pop\_back()

{

if (JSON\_HEDLEY\_UNLIKELY(empty()))

{

JSON\_THROW(detail::out\_of\_range::create(405, "JSON pointer has no parent"));

}

reference\_tokens.pop\_back();

}

/\*!

@brief return last reference token

@pre not `empty()`

@return last reference token

@liveexample{The example shows the usage of `back`.,json\_pointer\_\_back}

@complexity Constant.

@throw out\_of\_range.405 if JSON pointer has no parent

@since version 3.6.0

\*/

const std::string& back() const

{

if (JSON\_HEDLEY\_UNLIKELY(empty()))

{

JSON\_THROW(detail::out\_of\_range::create(405, "JSON pointer has no parent"));

}

return reference\_tokens.back();

}

/\*!

@brief append an unescaped token at the end of the reference pointer

@param[in] token token to add

@complexity Amortized constant.

@liveexample{The example shows the result of `push\_back` for different

JSON Pointers.,json\_pointer\_\_push\_back}

@since version 3.6.0

\*/

void push\_back(const std::string& token)

{

reference\_tokens.push\_back(token);

}

/// @copydoc push\_back(const std::string&)

void push\_back(std::string&& token)

{

reference\_tokens.push\_back(std::move(token));

}

/\*!

@brief return whether pointer points to the root document

@return true iff the JSON pointer points to the root document

@complexity Constant.

@exceptionsafety No-throw guarantee: this function never throws exceptions.

@liveexample{The example shows the result of `empty` for different JSON

Pointers.,json\_pointer\_\_empty}

@since version 3.6.0

\*/

bool empty() const noexcept

{

return reference\_tokens.empty();

}

private:

/\*!

@param[in] s reference token to be converted into an array index

@return integer representation of @a s

@throw out\_of\_range.404 if string @a s could not be converted to an integer

\*/

static int array\_index(const std::string& s)

{

std::size\_t processed\_chars = 0;

const int res = std::stoi(s, &processed\_chars);

// check if the string was completely read

if (JSON\_HEDLEY\_UNLIKELY(processed\_chars != s.size()))

{

JSON\_THROW(detail::out\_of\_range::create(404, "unresolved reference token '" + s + "'"));

}

return res;

}

json\_pointer top() const

{

if (JSON\_HEDLEY\_UNLIKELY(empty()))

{

JSON\_THROW(detail::out\_of\_range::create(405, "JSON pointer has no parent"));

}

json\_pointer result = \*this;

result.reference\_tokens = {reference\_tokens[0]};

return result;

}

/\*!

@brief create and return a reference to the pointed to value

@complexity Linear in the number of reference tokens.

@throw parse\_error.109 if array index is not a number

@throw type\_error.313 if value cannot be unflattened

\*/

BasicJsonType& get\_and\_create(BasicJsonType& j) const

{

using size\_type = typename BasicJsonType::size\_type;

auto result = &j;

// in case no reference tokens exist, return a reference to the JSON value

// j which will be overwritten by a primitive value

for (const auto& reference\_token : reference\_tokens)

{

switch (result->type())

{

case detail::value\_t::null:

{

if (reference\_token == "0")

{

// start a new array if reference token is 0

result = &result->operator[](0);

}

else

{

// start a new object otherwise

result = &result->operator[](reference\_token);

}

break;

}

case detail::value\_t::object:

{

// create an entry in the object

result = &result->operator[](reference\_token);

break;

}

case detail::value\_t::array:

{

// create an entry in the array

JSON\_TRY

{

result = &result->operator[](static\_cast<size\_type>(array\_index(reference\_token)));

}

JSON\_CATCH(std::invalid\_argument&)

{

JSON\_THROW(detail::parse\_error::create(109, 0, "array index '" + reference\_token + "' is not a number"));

}

break;

}

/\*

The following code is only reached if there exists a reference

token \_and\_ the current value is primitive. In this case, we have

an error situation, because primitive values may only occur as

single value; that is, with an empty list of reference tokens.

\*/

default:

JSON\_THROW(detail::type\_error::create(313, "invalid value to unflatten"));

}

}

return \*result;

}

/\*!

@brief return a reference to the pointed to value

@note This version does not throw if a value is not present, but tries to

create nested values instead. For instance, calling this function

with pointer `"/this/that"` on a null value is equivalent to calling

`operator[]("this").operator[]("that")` on that value, effectively

changing the null value to an object.

@param[in] ptr a JSON value

@return reference to the JSON value pointed to by the JSON pointer

@complexity Linear in the length of the JSON pointer.

@throw parse\_error.106 if an array index begins with '0'

@throw parse\_error.109 if an array index was not a number

@throw out\_of\_range.404 if the JSON pointer can not be resolved

\*/

BasicJsonType& get\_unchecked(BasicJsonType\* ptr) const

{

using size\_type = typename BasicJsonType::size\_type;

for (const auto& reference\_token : reference\_tokens)

{

// convert null values to arrays or objects before continuing

if (ptr->is\_null())

{

// check if reference token is a number

const bool nums =

std::all\_of(reference\_token.begin(), reference\_token.end(),

[](const unsigned char x)

{

return std::isdigit(x);

});

// change value to array for numbers or "-" or to object otherwise

\*ptr = (nums or reference\_token == "-")

? detail::value\_t::array

: detail::value\_t::object;

}

switch (ptr->type())

{

case detail::value\_t::object:

{

// use unchecked object access

ptr = &ptr->operator[](reference\_token);

break;

}

case detail::value\_t::array:

{

// error condition (cf. RFC 6901, Sect. 4)

if (JSON\_HEDLEY\_UNLIKELY(reference\_token.size() > 1 and reference\_token[0] == '0'))

{

JSON\_THROW(detail::parse\_error::create(106, 0,

"array index '" + reference\_token +

"' must not begin with '0'"));

}

if (reference\_token == "-")

{

// explicitly treat "-" as index beyond the end

ptr = &ptr->operator[](ptr->m\_value.array->size());

}

else

{

// convert array index to number; unchecked access

JSON\_TRY

{

ptr = &ptr->operator[](

static\_cast<size\_type>(array\_index(reference\_token)));

}

JSON\_CATCH(std::invalid\_argument&)

{

JSON\_THROW(detail::parse\_error::create(109, 0, "array index '" + reference\_token + "' is not a number"));

}

}

break;

}

default:

JSON\_THROW(detail::out\_of\_range::create(404, "unresolved reference token '" + reference\_token + "'"));

}

}

return \*ptr;

}

/\*!

@throw parse\_error.106 if an array index begins with '0'

@throw parse\_error.109 if an array index was not a number

@throw out\_of\_range.402 if the array index '-' is used

@throw out\_of\_range.404 if the JSON pointer can not be resolved

\*/

BasicJsonType& get\_checked(BasicJsonType\* ptr) const

{

using size\_type = typename BasicJsonType::size\_type;

for (const auto& reference\_token : reference\_tokens)

{

switch (ptr->type())

{

case detail::value\_t::object:

{

// note: at performs range check

ptr = &ptr->at(reference\_token);

break;

}

case detail::value\_t::array:

{

if (JSON\_HEDLEY\_UNLIKELY(reference\_token == "-"))

{

// "-" always fails the range check

JSON\_THROW(detail::out\_of\_range::create(402,

"array index '-' (" + std::to\_string(ptr->m\_value.array->size()) +

") is out of range"));

}

// error condition (cf. RFC 6901, Sect. 4)

if (JSON\_HEDLEY\_UNLIKELY(reference\_token.size() > 1 and reference\_token[0] == '0'))

{

JSON\_THROW(detail::parse\_error::create(106, 0,

"array index '" + reference\_token +

"' must not begin with '0'"));

}

// note: at performs range check

JSON\_TRY

{

ptr = &ptr->at(static\_cast<size\_type>(array\_index(reference\_token)));

}

JSON\_CATCH(std::invalid\_argument&)

{

JSON\_THROW(detail::parse\_error::create(109, 0, "array index '" + reference\_token + "' is not a number"));

}

break;

}

default:

JSON\_THROW(detail::out\_of\_range::create(404, "unresolved reference token '" + reference\_token + "'"));

}

}

return \*ptr;

}

/\*!

@brief return a const reference to the pointed to value

@param[in] ptr a JSON value

@return const reference to the JSON value pointed to by the JSON

pointer

@throw parse\_error.106 if an array index begins with '0'

@throw parse\_error.109 if an array index was not a number

@throw out\_of\_range.402 if the array index '-' is used

@throw out\_of\_range.404 if the JSON pointer can not be resolved

\*/

const BasicJsonType& get\_unchecked(const BasicJsonType\* ptr) const

{

using size\_type = typename BasicJsonType::size\_type;

for (const auto& reference\_token : reference\_tokens)

{

switch (ptr->type())

{

case detail::value\_t::object:

{

// use unchecked object access

ptr = &ptr->operator[](reference\_token);

break;

}

case detail::value\_t::array:

{

if (JSON\_HEDLEY\_UNLIKELY(reference\_token == "-"))

{

// "-" cannot be used for const access

JSON\_THROW(detail::out\_of\_range::create(402,

"array index '-' (" + std::to\_string(ptr->m\_value.array->size()) +

") is out of range"));

}

// error condition (cf. RFC 6901, Sect. 4)

if (JSON\_HEDLEY\_UNLIKELY(reference\_token.size() > 1 and reference\_token[0] == '0'))

{

JSON\_THROW(detail::parse\_error::create(106, 0,

"array index '" + reference\_token +

"' must not begin with '0'"));

}

// use unchecked array access

JSON\_TRY

{

ptr = &ptr->operator[](

static\_cast<size\_type>(array\_index(reference\_token)));

}

JSON\_CATCH(std::invalid\_argument&)

{

JSON\_THROW(detail::parse\_error::create(109, 0, "array index '" + reference\_token + "' is not a number"));

}

break;

}

default:

JSON\_THROW(detail::out\_of\_range::create(404, "unresolved reference token '" + reference\_token + "'"));

}

}

return \*ptr;

}

/\*!

@throw parse\_error.106 if an array index begins with '0'

@throw parse\_error.109 if an array index was not a number

@throw out\_of\_range.402 if the array index '-' is used

@throw out\_of\_range.404 if the JSON pointer can not be resolved

\*/

const BasicJsonType& get\_checked(const BasicJsonType\* ptr) const

{

using size\_type = typename BasicJsonType::size\_type;

for (const auto& reference\_token : reference\_tokens)

{

switch (ptr->type())

{

case detail::value\_t::object:

{

// note: at performs range check

ptr = &ptr->at(reference\_token);

break;

}

case detail::value\_t::array:

{

if (JSON\_HEDLEY\_UNLIKELY(reference\_token == "-"))

{

// "-" always fails the range check

JSON\_THROW(detail::out\_of\_range::create(402,

"array index '-' (" + std::to\_string(ptr->m\_value.array->size()) +

") is out of range"));

}

// error condition (cf. RFC 6901, Sect. 4)

if (JSON\_HEDLEY\_UNLIKELY(reference\_token.size() > 1 and reference\_token[0] == '0'))

{

JSON\_THROW(detail::parse\_error::create(106, 0,

"array index '" + reference\_token +

"' must not begin with '0'"));

}

// note: at performs range check

JSON\_TRY

{

ptr = &ptr->at(static\_cast<size\_type>(array\_index(reference\_token)));

}

JSON\_CATCH(std::invalid\_argument&)

{

JSON\_THROW(detail::parse\_error::create(109, 0, "array index '" + reference\_token + "' is not a number"));

}

break;

}

default:

JSON\_THROW(detail::out\_of\_range::create(404, "unresolved reference token '" + reference\_token + "'"));

}

}

return \*ptr;

}

/\*!

@throw parse\_error.106 if an array index begins with '0'

@throw parse\_error.109 if an array index was not a number

\*/

bool contains(const BasicJsonType\* ptr) const

{

using size\_type = typename BasicJsonType::size\_type;

for (const auto& reference\_token : reference\_tokens)

{

switch (ptr->type())

{

case detail::value\_t::object:

{

if (not ptr->contains(reference\_token))

{

// we did not find the key in the object

return false;

}

ptr = &ptr->operator[](reference\_token);

break;

}

case detail::value\_t::array:

{

if (JSON\_HEDLEY\_UNLIKELY(reference\_token == "-"))

{

// "-" always fails the range check

return false;

}

// error condition (cf. RFC 6901, Sect. 4)

if (JSON\_HEDLEY\_UNLIKELY(reference\_token.size() > 1 and reference\_token[0] == '0'))

{

JSON\_THROW(detail::parse\_error::create(106, 0,

"array index '" + reference\_token +

"' must not begin with '0'"));

}

JSON\_TRY

{

const auto idx = static\_cast<size\_type>(array\_index(reference\_token));

if (idx >= ptr->size())

{

// index out of range

return false;

}

ptr = &ptr->operator[](idx);

break;

}

JSON\_CATCH(std::invalid\_argument&)

{

JSON\_THROW(detail::parse\_error::create(109, 0, "array index '" + reference\_token + "' is not a number"));

}

break;

}

default:

{

// we do not expect primitive values if there is still a

// reference token to process

return false;

}

}

}

// no reference token left means we found a primitive value

return true;

}

/\*!

@brief split the string input to reference tokens

@note This function is only called by the json\_pointer constructor.

All exceptions below are documented there.

@throw parse\_error.107 if the pointer is not empty or begins with '/'

@throw parse\_error.108 if character '~' is not followed by '0' or '1'

\*/

static std::vector<std::string> split(const std::string& reference\_string)

{

std::vector<std::string> result;

// special case: empty reference string -> no reference tokens

if (reference\_string.empty())

{

return result;

}

// check if nonempty reference string begins with slash

if (JSON\_HEDLEY\_UNLIKELY(reference\_string[0] != '/'))

{

JSON\_THROW(detail::parse\_error::create(107, 1,

"JSON pointer must be empty or begin with '/' - was: '" +

reference\_string + "'"));

}

// extract the reference tokens:

// - slash: position of the last read slash (or end of string)

// - start: position after the previous slash

for (

// search for the first slash after the first character

std::size\_t slash = reference\_string.find\_first\_of('/', 1),

// set the beginning of the first reference token

start = 1;

// we can stop if start == 0 (if slash == std::string::npos)

start != 0;

// set the beginning of the next reference token

// (will eventually be 0 if slash == std::string::npos)

start = (slash == std::string::npos) ? 0 : slash + 1,

// find next slash

slash = reference\_string.find\_first\_of('/', start))

{

// use the text between the beginning of the reference token

// (start) and the last slash (slash).

auto reference\_token = reference\_string.substr(start, slash - start);

// check reference tokens are properly escaped

for (std::size\_t pos = reference\_token.find\_first\_of('~');

pos != std::string::npos;

pos = reference\_token.find\_first\_of('~', pos + 1))

{

assert(reference\_token[pos] == '~');

// ~ must be followed by 0 or 1

if (JSON\_HEDLEY\_UNLIKELY(pos == reference\_token.size() - 1 or

(reference\_token[pos + 1] != '0' and

reference\_token[pos + 1] != '1')))

{

JSON\_THROW(detail::parse\_error::create(108, 0, "escape character '~' must be followed with '0' or '1'"));

}

}

// finally, store the reference token

unescape(reference\_token);

result.push\_back(reference\_token);

}

return result;

}

/\*!

@brief replace all occurrences of a substring by another string

@param[in,out] s the string to manipulate; changed so that all

occurrences of @a f are replaced with @a t

@param[in] f the substring to replace with @a t

@param[in] t the string to replace @a f

@pre The search string @a f must not be empty. \*\*This precondition is

enforced with an assertion.\*\*

@since version 2.0.0

\*/

static void replace\_substring(std::string& s, const std::string& f,

const std::string& t)

{

assert(not f.empty());

for (auto pos = s.find(f); // find first occurrence of f

pos != std::string::npos; // make sure f was found

s.replace(pos, f.size(), t), // replace with t, and

pos = s.find(f, pos + t.size())) // find next occurrence of f

{}

}

/// escape "~" to "~0" and "/" to "~1"

static std::string escape(std::string s)

{

replace\_substring(s, "~", "~0");

replace\_substring(s, "/", "~1");

return s;

}

/// unescape "~1" to tilde and "~0" to slash (order is important!)

static void unescape(std::string& s)

{

replace\_substring(s, "~1", "/");

replace\_substring(s, "~0", "~");

}

/\*!

@param[in] reference\_string the reference string to the current value

@param[in] value the value to consider

@param[in,out] result the result object to insert values to

@note Empty objects or arrays are flattened to `null`.

\*/

static void flatten(const std::string& reference\_string,

const BasicJsonType& value,

BasicJsonType& result)

{

switch (value.type())

{

case detail::value\_t::array:

{

if (value.m\_value.array->empty())

{

// flatten empty array as null

result[reference\_string] = nullptr;

}

else

{

// iterate array and use index as reference string

for (std::size\_t i = 0; i < value.m\_value.array->size(); ++i)

{

flatten(reference\_string + "/" + std::to\_string(i),

value.m\_value.array->operator[](i), result);

}

}

break;

}

case detail::value\_t::object:

{

if (value.m\_value.object->empty())

{

// flatten empty object as null

result[reference\_string] = nullptr;

}

else

{

// iterate object and use keys as reference string

for (const auto& element : \*value.m\_value.object)

{

flatten(reference\_string + "/" + escape(element.first), element.second, result);

}

}

break;

}

default:

{

// add primitive value with its reference string

result[reference\_string] = value;

break;

}

}

}

/\*!

@param[in] value flattened JSON

@return unflattened JSON

@throw parse\_error.109 if array index is not a number

@throw type\_error.314 if value is not an object

@throw type\_error.315 if object values are not primitive

@throw type\_error.313 if value cannot be unflattened

\*/

static BasicJsonType

unflatten(const BasicJsonType& value)

{

if (JSON\_HEDLEY\_UNLIKELY(not value.is\_object()))

{

JSON\_THROW(detail::type\_error::create(314, "only objects can be unflattened"));

}

BasicJsonType result;

// iterate the JSON object values

for (const auto& element : \*value.m\_value.object)

{

if (JSON\_HEDLEY\_UNLIKELY(not element.second.is\_primitive()))

{

JSON\_THROW(detail::type\_error::create(315, "values in object must be primitive"));

}

// assign value to reference pointed to by JSON pointer; Note that if

// the JSON pointer is "" (i.e., points to the whole value), function

// get\_and\_create returns a reference to result itself. An assignment

// will then create a primitive value.

json\_pointer(element.first).get\_and\_create(result) = element.second;

}

return result;

}

/\*!

@brief compares two JSON pointers for equality

@param[in] lhs JSON pointer to compare

@param[in] rhs JSON pointer to compare

@return whether @a lhs is equal to @a rhs

@complexity Linear in the length of the JSON pointer

@exceptionsafety No-throw guarantee: this function never throws exceptions.

\*/

friend bool operator==(json\_pointer const& lhs,

json\_pointer const& rhs) noexcept

{

return lhs.reference\_tokens == rhs.reference\_tokens;

}

/\*!

@brief compares two JSON pointers for inequality

@param[in] lhs JSON pointer to compare

@param[in] rhs JSON pointer to compare

@return whether @a lhs is not equal @a rhs

@complexity Linear in the length of the JSON pointer

@exceptionsafety No-throw guarantee: this function never throws exceptions.

\*/

friend bool operator!=(json\_pointer const& lhs,

json\_pointer const& rhs) noexcept

{

return not (lhs == rhs);

}

/// the reference tokens

std::vector<std::string> reference\_tokens;

};

} // namespace nlohmann

// #include <nlohmann/detail/json\_ref.hpp>

#include <initializer\_list>

#include <utility>

// #include <nlohmann/detail/meta/type\_traits.hpp>

namespace nlohmann

{

namespace detail

{

template<typename BasicJsonType>

class json\_ref

{

public:

using value\_type = BasicJsonType;

json\_ref(value\_type&& value)

: owned\_value(std::move(value)), value\_ref(&owned\_value), is\_rvalue(true)

{}

json\_ref(const value\_type& value)

: value\_ref(const\_cast<value\_type\*>(&value)), is\_rvalue(false)

{}

json\_ref(std::initializer\_list<json\_ref> init)

: owned\_value(init), value\_ref(&owned\_value), is\_rvalue(true)

{}

template <

class... Args,

enable\_if\_t<std::is\_constructible<value\_type, Args...>::value, int> = 0 >

json\_ref(Args && ... args)

: owned\_value(std::forward<Args>(args)...), value\_ref(&owned\_value),

is\_rvalue(true) {}

// class should be movable only

json\_ref(json\_ref&&) = default;

json\_ref(const json\_ref&) = delete;

json\_ref& operator=(const json\_ref&) = delete;

json\_ref& operator=(json\_ref&&) = delete;

~json\_ref() = default;

value\_type moved\_or\_copied() const

{

if (is\_rvalue)

{

return std::move(\*value\_ref);

}

return \*value\_ref;

}

value\_type const& operator\*() const

{

return \*static\_cast<value\_type const\*>(value\_ref);

}

value\_type const\* operator->() const

{

return static\_cast<value\_type const\*>(value\_ref);

}

private:

mutable value\_type owned\_value = nullptr;

value\_type\* value\_ref = nullptr;

const bool is\_rvalue;

};

} // namespace detail

} // namespace nlohmann

// #include <nlohmann/detail/macro\_scope.hpp>

// #include <nlohmann/detail/meta/cpp\_future.hpp>

// #include <nlohmann/detail/meta/type\_traits.hpp>

// #include <nlohmann/detail/output/binary\_writer.hpp>

#include <algorithm> // reverse

#include <array> // array

#include <cstdint> // uint8\_t, uint16\_t, uint32\_t, uint64\_t

#include <cstring> // memcpy

#include <limits> // numeric\_limits

#include <string> // string

// #include <nlohmann/detail/input/binary\_reader.hpp>

// #include <nlohmann/detail/macro\_scope.hpp>

// #include <nlohmann/detail/output/output\_adapters.hpp>

#include <algorithm> // copy

#include <cstddef> // size\_t

#include <ios> // streamsize

#include <iterator> // back\_inserter

#include <memory> // shared\_ptr, make\_shared

#include <ostream> // basic\_ostream

#include <string> // basic\_string

#include <vector> // vector

// #include <nlohmann/detail/macro\_scope.hpp>

namespace nlohmann

{

namespace detail

{

/// abstract output adapter interface

template<typename CharType> struct output\_adapter\_protocol

{

virtual void write\_character(CharType c) = 0;

virtual void write\_characters(const CharType\* s, std::size\_t length) = 0;

virtual ~output\_adapter\_protocol() = default;

};

/// a type to simplify interfaces

template<typename CharType>

using output\_adapter\_t = std::shared\_ptr<output\_adapter\_protocol<CharType>>;

/// output adapter for byte vectors

template<typename CharType>

class output\_vector\_adapter : public output\_adapter\_protocol<CharType>

{

public:

explicit output\_vector\_adapter(std::vector<CharType>& vec) noexcept

: v(vec)

{}

void write\_character(CharType c) override

{

v.push\_back(c);

}

JSON\_HEDLEY\_NON\_NULL(2)

void write\_characters(const CharType\* s, std::size\_t length) override

{

std::copy(s, s + length, std::back\_inserter(v));

}

private:

std::vector<CharType>& v;

};

/// output adapter for output streams

template<typename CharType>

class output\_stream\_adapter : public output\_adapter\_protocol<CharType>

{

public:

explicit output\_stream\_adapter(std::basic\_ostream<CharType>& s) noexcept

: stream(s)

{}

void write\_character(CharType c) override

{

stream.put(c);

}

JSON\_HEDLEY\_NON\_NULL(2)

void write\_characters(const CharType\* s, std::size\_t length) override

{

stream.write(s, static\_cast<std::streamsize>(length));

}

private:

std::basic\_ostream<CharType>& stream;

};

/// output adapter for basic\_string

template<typename CharType, typename StringType = std::basic\_string<CharType>>

class output\_string\_adapter : public output\_adapter\_protocol<CharType>

{

public:

explicit output\_string\_adapter(StringType& s) noexcept

: str(s)

{}

void write\_character(CharType c) override

{

str.push\_back(c);

}

JSON\_HEDLEY\_NON\_NULL(2)

void write\_characters(const CharType\* s, std::size\_t length) override

{

str.append(s, length);

}

private:

StringType& str;

};

template<typename CharType, typename StringType = std::basic\_string<CharType>>

class output\_adapter

{

public:

output\_adapter(std::vector<CharType>& vec)

: oa(std::make\_shared<output\_vector\_adapter<CharType>>(vec)) {}

output\_adapter(std::basic\_ostream<CharType>& s)

: oa(std::make\_shared<output\_stream\_adapter<CharType>>(s)) {}

output\_adapter(StringType& s)

: oa(std::make\_shared<output\_string\_adapter<CharType, StringType>>(s)) {}

operator output\_adapter\_t<CharType>()

{

return oa;

}

private:

output\_adapter\_t<CharType> oa = nullptr;

};

} // namespace detail

} // namespace nlohmann

namespace nlohmann

{

namespace detail

{

///////////////////

// binary writer //

///////////////////

/\*!

@brief serialization to CBOR and MessagePack values

\*/

template<typename BasicJsonType, typename CharType>

class binary\_writer

{

using string\_t = typename BasicJsonType::string\_t;

public:

/\*!

@brief create a binary writer

@param[in] adapter output adapter to write to

\*/

explicit binary\_writer(output\_adapter\_t<CharType> adapter) : oa(adapter)

{

assert(oa);

}

/\*!

@param[in] j JSON value to serialize

@pre j.type() == value\_t::object

\*/

void write\_bson(const BasicJsonType& j)

{

switch (j.type())

{

case value\_t::object:

{

write\_bson\_object(\*j.m\_value.object);

break;

}

default:

{

JSON\_THROW(type\_error::create(317, "to serialize to BSON, top-level type must be object, but is " + std::string(j.type\_name())));

}

}

}

/\*!

@param[in] j JSON value to serialize

\*/

void write\_cbor(const BasicJsonType& j)

{

switch (j.type())

{

case value\_t::null:

{

oa->write\_character(to\_char\_type(0xF6));

break;

}

case value\_t::boolean:

{

oa->write\_character(j.m\_value.boolean

? to\_char\_type(0xF5)

: to\_char\_type(0xF4));

break;

}

case value\_t::number\_integer:

{

if (j.m\_value.number\_integer >= 0)

{

// CBOR does not differentiate between positive signed

// integers and unsigned integers. Therefore, we used the

// code from the value\_t::number\_unsigned case here.

if (j.m\_value.number\_integer <= 0x17)

{

write\_number(static\_cast<std::uint8\_t>(j.m\_value.number\_integer));

}

else if (j.m\_value.number\_integer <= (std::numeric\_limits<std::uint8\_t>::max)())

{

oa->write\_character(to\_char\_type(0x18));

write\_number(static\_cast<std::uint8\_t>(j.m\_value.number\_integer));

}

else if (j.m\_value.number\_integer <= (std::numeric\_limits<std::uint16\_t>::max)())

{

oa->write\_character(to\_char\_type(0x19));

write\_number(static\_cast<std::uint16\_t>(j.m\_value.number\_integer));

}

else if (j.m\_value.number\_integer <= (std::numeric\_limits<std::uint32\_t>::max)())

{

oa->write\_character(to\_char\_type(0x1A));

write\_number(static\_cast<std::uint32\_t>(j.m\_value.number\_integer));

}

else

{

oa->write\_character(to\_char\_type(0x1B));

write\_number(static\_cast<std::uint64\_t>(j.m\_value.number\_integer));

}

}

else

{

// The conversions below encode the sign in the first

// byte, and the value is converted to a positive number.

const auto positive\_number = -1 - j.m\_value.number\_integer;

if (j.m\_value.number\_integer >= -24)

{

write\_number(static\_cast<std::uint8\_t>(0x20 + positive\_number));

}

else if (positive\_number <= (std::numeric\_limits<std::uint8\_t>::max)())

{

oa->write\_character(to\_char\_type(0x38));

write\_number(static\_cast<std::uint8\_t>(positive\_number));

}

else if (positive\_number <= (std::numeric\_limits<std::uint16\_t>::max)())

{

oa->write\_character(to\_char\_type(0x39));

write\_number(static\_cast<std::uint16\_t>(positive\_number));

}

else if (positive\_number <= (std::numeric\_limits<std::uint32\_t>::max)())

{

oa->write\_character(to\_char\_type(0x3A));

write\_number(static\_cast<std::uint32\_t>(positive\_number));

}

else

{

oa->write\_character(to\_char\_type(0x3B));

write\_number(static\_cast<std::uint64\_t>(positive\_number));

}

}

break;

}

case value\_t::number\_unsigned:

{

if (j.m\_value.number\_unsigned <= 0x17)

{

write\_number(static\_cast<std::uint8\_t>(j.m\_value.number\_unsigned));

}

else if (j.m\_value.number\_unsigned <= (std::numeric\_limits<std::uint8\_t>::max)())

{

oa->write\_character(to\_char\_type(0x18));

write\_number(static\_cast<std::uint8\_t>(j.m\_value.number\_unsigned));

}

else if (j.m\_value.number\_unsigned <= (std::numeric\_limits<std::uint16\_t>::max)())

{

oa->write\_character(to\_char\_type(0x19));

write\_number(static\_cast<std::uint16\_t>(j.m\_value.number\_unsigned));

}

else if (j.m\_value.number\_unsigned <= (std::numeric\_limits<std::uint32\_t>::max)())

{

oa->write\_character(to\_char\_type(0x1A));

write\_number(static\_cast<std::uint32\_t>(j.m\_value.number\_unsigned));

}

else

{

oa->write\_character(to\_char\_type(0x1B));

write\_number(static\_cast<std::uint64\_t>(j.m\_value.number\_unsigned));

}

break;

}

case value\_t::number\_float:

{

oa->write\_character(get\_cbor\_float\_prefix(j.m\_value.number\_float));

write\_number(j.m\_value.number\_float);

break;

}

case value\_t::string:

{

// step 1: write control byte and the string length

const auto N = j.m\_value.string->size();

if (N <= 0x17)

{

write\_number(static\_cast<std::uint8\_t>(0x60 + N));

}

else if (N <= (std::numeric\_limits<std::uint8\_t>::max)())

{

oa->write\_character(to\_char\_type(0x78));

write\_number(static\_cast<std::uint8\_t>(N));

}

else if (N <= (std::numeric\_limits<std::uint16\_t>::max)())

{

oa->write\_character(to\_char\_type(0x79));

write\_number(static\_cast<std::uint16\_t>(N));

}

else if (N <= (std::numeric\_limits<std::uint32\_t>::max)())

{

oa->write\_character(to\_char\_type(0x7A));

write\_number(static\_cast<std::uint32\_t>(N));

}

// LCOV\_EXCL\_START

else if (N <= (std::numeric\_limits<std::uint64\_t>::max)())

{

oa->write\_character(to\_char\_type(0x7B));

write\_number(static\_cast<std::uint64\_t>(N));

}

// LCOV\_EXCL\_STOP

// step 2: write the string

oa->write\_characters(

reinterpret\_cast<const CharType\*>(j.m\_value.string->c\_str()),

j.m\_value.string->size());

break;

}

case value\_t::array:

{

// step 1: write control byte and the array size

const auto N = j.m\_value.array->size();

if (N <= 0x17)

{

write\_number(static\_cast<std::uint8\_t>(0x80 + N));

}

else if (N <= (std::numeric\_limits<std::uint8\_t>::max)())

{

oa->write\_character(to\_char\_type(0x98));

write\_number(static\_cast<std::uint8\_t>(N));

}

else if (N <= (std::numeric\_limits<std::uint16\_t>::max)())

{

oa->write\_character(to\_char\_type(0x99));

write\_number(static\_cast<std::uint16\_t>(N));

}

else if (N <= (std::numeric\_limits<std::uint32\_t>::max)())

{

oa->write\_character(to\_char\_type(0x9A));

write\_number(static\_cast<std::uint32\_t>(N));

}

// LCOV\_EXCL\_START

else if (N <= (std::numeric\_limits<std::uint64\_t>::max)())

{

oa->write\_character(to\_char\_type(0x9B));

write\_number(static\_cast<std::uint64\_t>(N));

}

// LCOV\_EXCL\_STOP

// step 2: write each element

for (const auto& el : \*j.m\_value.array)

{

write\_cbor(el);

}

break;

}

case value\_t::object:

{

// step 1: write control byte and the object size

const auto N = j.m\_value.object->size();

if (N <= 0x17)

{

write\_number(static\_cast<std::uint8\_t>(0xA0 + N));

}

else if (N <= (std::numeric\_limits<std::uint8\_t>::max)())

{

oa->write\_character(to\_char\_type(0xB8));

write\_number(static\_cast<std::uint8\_t>(N));

}

else if (N <= (std::numeric\_limits<std::uint16\_t>::max)())

{

oa->write\_character(to\_char\_type(0xB9));

write\_number(static\_cast<std::uint16\_t>(N));

}

else if (N <= (std::numeric\_limits<std::uint32\_t>::max)())

{

oa->write\_character(to\_char\_type(0xBA));

write\_number(static\_cast<std::uint32\_t>(N));

}

// LCOV\_EXCL\_START

else if (N <= (std::numeric\_limits<std::uint64\_t>::max)())

{

oa->write\_character(to\_char\_type(0xBB));

write\_number(static\_cast<std::uint64\_t>(N));

}

// LCOV\_EXCL\_STOP

// step 2: write each element

for (const auto& el : \*j.m\_value.object)

{

write\_cbor(el.first);

write\_cbor(el.second);

}

break;

}

default:

break;

}

}

/\*!

@param[in] j JSON value to serialize

\*/

void write\_msgpack(const BasicJsonType& j)

{

switch (j.type())

{

case value\_t::null: // nil

{

oa->write\_character(to\_char\_type(0xC0));

break;

}

case value\_t::boolean: // true and false

{

oa->write\_character(j.m\_value.boolean

? to\_char\_type(0xC3)

: to\_char\_type(0xC2));

break;

}

case value\_t::number\_integer:

{

if (j.m\_value.number\_integer >= 0)

{

// MessagePack does not differentiate between positive

// signed integers and unsigned integers. Therefore, we used

// the code from the value\_t::number\_unsigned case here.

if (j.m\_value.number\_unsigned < 128)

{

// positive fixnum

write\_number(static\_cast<std::uint8\_t>(j.m\_value.number\_integer));

}

else if (j.m\_value.number\_unsigned <= (std::numeric\_limits<std::uint8\_t>::max)())

{

// uint 8

oa->write\_character(to\_char\_type(0xCC));

write\_number(static\_cast<std::uint8\_t>(j.m\_value.number\_integer));

}

else if (j.m\_value.number\_unsigned <= (std::numeric\_limits<std::uint16\_t>::max)())

{

// uint 16

oa->write\_character(to\_char\_type(0xCD));

write\_number(static\_cast<std::uint16\_t>(j.m\_value.number\_integer));

}

else if (j.m\_value.number\_unsigned <= (std::numeric\_limits<std::uint32\_t>::max)())

{

// uint 32

oa->write\_character(to\_char\_type(0xCE));

write\_number(static\_cast<std::uint32\_t>(j.m\_value.number\_integer));

}

else if (j.m\_value.number\_unsigned <= (std::numeric\_limits<std::uint64\_t>::max)())

{

// uint 64

oa->write\_character(to\_char\_type(0xCF));

write\_number(static\_cast<std::uint64\_t>(j.m\_value.number\_integer));

}

}

else

{

if (j.m\_value.number\_integer >= -32)

{

// negative fixnum

write\_number(static\_cast<std::int8\_t>(j.m\_value.number\_integer));

}

else if (j.m\_value.number\_integer >= (std::numeric\_limits<std::int8\_t>::min)() and

j.m\_value.number\_integer <= (std::numeric\_limits<std::int8\_t>::max)())

{

// int 8

oa->write\_character(to\_char\_type(0xD0));

write\_number(static\_cast<std::int8\_t>(j.m\_value.number\_integer));

}

else if (j.m\_value.number\_integer >= (std::numeric\_limits<std::int16\_t>::min)() and

j.m\_value.number\_integer <= (std::numeric\_limits<std::int16\_t>::max)())

{

// int 16

oa->write\_character(to\_char\_type(0xD1));

write\_number(static\_cast<std::int16\_t>(j.m\_value.number\_integer));

}

else if (j.m\_value.number\_integer >= (std::numeric\_limits<std::int32\_t>::min)() and

j.m\_value.number\_integer <= (std::numeric\_limits<std::int32\_t>::max)())

{

// int 32

oa->write\_character(to\_char\_type(0xD2));

write\_number(static\_cast<std::int32\_t>(j.m\_value.number\_integer));

}

else if (j.m\_value.number\_integer >= (std::numeric\_limits<std::int64\_t>::min)() and

j.m\_value.number\_integer <= (std::numeric\_limits<std::int64\_t>::max)())

{

// int 64

oa->write\_character(to\_char\_type(0xD3));

write\_number(static\_cast<std::int64\_t>(j.m\_value.number\_integer));

}

}

break;

}

case value\_t::number\_unsigned:

{

if (j.m\_value.number\_unsigned < 128)

{

// positive fixnum

write\_number(static\_cast<std::uint8\_t>(j.m\_value.number\_integer));

}

else if (j.m\_value.number\_unsigned <= (std::numeric\_limits<std::uint8\_t>::max)())

{

// uint 8

oa->write\_character(to\_char\_type(0xCC));

write\_number(static\_cast<std::uint8\_t>(j.m\_value.number\_integer));

}

else if (j.m\_value.number\_unsigned <= (std::numeric\_limits<std::uint16\_t>::max)())

{

// uint 16

oa->write\_character(to\_char\_type(0xCD));

write\_number(static\_cast<std::uint16\_t>(j.m\_value.number\_integer));

}

else if (j.m\_value.number\_unsigned <= (std::numeric\_limits<std::uint32\_t>::max)())

{

// uint 32

oa->write\_character(to\_char\_type(0xCE));

write\_number(static\_cast<std::uint32\_t>(j.m\_value.number\_integer));

}

else if (j.m\_value.number\_unsigned <= (std::numeric\_limits<std::uint64\_t>::max)())

{

// uint 64

oa->write\_character(to\_char\_type(0xCF));

write\_number(static\_cast<std::uint64\_t>(j.m\_value.number\_integer));

}

break;

}

case value\_t::number\_float:

{

oa->write\_character(get\_msgpack\_float\_prefix(j.m\_value.number\_float));

write\_number(j.m\_value.number\_float);

break;

}

case value\_t::string:

{

// step 1: write control byte and the string length

const auto N = j.m\_value.string->size();

if (N <= 31)

{

// fixstr

write\_number(static\_cast<std::uint8\_t>(0xA0 | N));

}

else if (N <= (std::numeric\_limits<std::uint8\_t>::max)())

{

// str 8

oa->write\_character(to\_char\_type(0xD9));

write\_number(static\_cast<std::uint8\_t>(N));

}

else if (N <= (std::numeric\_limits<std::uint16\_t>::max)())

{

// str 16

oa->write\_character(to\_char\_type(0xDA));

write\_number(static\_cast<std::uint16\_t>(N));

}

else if (N <= (std::numeric\_limits<std::uint32\_t>::max)())

{

// str 32

oa->write\_character(to\_char\_type(0xDB));

write\_number(static\_cast<std::uint32\_t>(N));

}

// step 2: write the string

oa->write\_characters(

reinterpret\_cast<const CharType\*>(j.m\_value.string->c\_str()),

j.m\_value.string->size());

break;

}

case value\_t::array:

{

// step 1: write control byte and the array size

const auto N = j.m\_value.array->size();

if (N <= 15)

{

// fixarray

write\_number(static\_cast<std::uint8\_t>(0x90 | N));

}

else if (N <= (std::numeric\_limits<std::uint16\_t>::max)())

{

// array 16

oa->write\_character(to\_char\_type(0xDC));

write\_number(static\_cast<std::uint16\_t>(N));

}

else if (N <= (std::numeric\_limits<std::uint32\_t>::max)())

{

// array 32

oa->write\_character(to\_char\_type(0xDD));

write\_number(static\_cast<std::uint32\_t>(N));

}

// step 2: write each element

for (const auto& el : \*j.m\_value.array)

{

write\_msgpack(el);

}

break;

}

case value\_t::object:

{

// step 1: write control byte and the object size

const auto N = j.m\_value.object->size();

if (N <= 15)

{

// fixmap

write\_number(static\_cast<std::uint8\_t>(0x80 | (N & 0xF)));

}

else if (N <= (std::numeric\_limits<std::uint16\_t>::max)())

{

// map 16

oa->write\_character(to\_char\_type(0xDE));

write\_number(static\_cast<std::uint16\_t>(N));

}

else if (N <= (std::numeric\_limits<std::uint32\_t>::max)())

{

// map 32

oa->write\_character(to\_char\_type(0xDF));

write\_number(static\_cast<std::uint32\_t>(N));

}

// step 2: write each element

for (const auto& el : \*j.m\_value.object)

{

write\_msgpack(el.first);

write\_msgpack(el.second);

}

break;

}

default:

break;

}

}

/\*!

@param[in] j JSON value to serialize

@param[in] use\_count whether to use '#' prefixes (optimized format)

@param[in] use\_type whether to use '$' prefixes (optimized format)

@param[in] add\_prefix whether prefixes need to be used for this value

\*/

void write\_ubjson(const BasicJsonType& j, const bool use\_count,

const bool use\_type, const bool add\_prefix = true)

{

switch (j.type())

{

case value\_t::null:

{

if (add\_prefix)

{

oa->write\_character(to\_char\_type('Z'));

}

break;

}

case value\_t::boolean:

{

if (add\_prefix)

{

oa->write\_character(j.m\_value.boolean

? to\_char\_type('T')

: to\_char\_type('F'));

}

break;

}

case value\_t::number\_integer:

{

write\_number\_with\_ubjson\_prefix(j.m\_value.number\_integer, add\_prefix);

break;

}

case value\_t::number\_unsigned:

{

write\_number\_with\_ubjson\_prefix(j.m\_value.number\_unsigned, add\_prefix);

break;

}

case value\_t::number\_float:

{

write\_number\_with\_ubjson\_prefix(j.m\_value.number\_float, add\_prefix);

break;

}

case value\_t::string:

{

if (add\_prefix)

{

oa->write\_character(to\_char\_type('S'));

}

write\_number\_with\_ubjson\_prefix(j.m\_value.string->size(), true);

oa->write\_characters(

reinterpret\_cast<const CharType\*>(j.m\_value.string->c\_str()),

j.m\_value.string->size());

break;

}

case value\_t::array:

{

if (add\_prefix)

{

oa->write\_character(to\_char\_type('['));

}

bool prefix\_required = true;

if (use\_type and not j.m\_value.array->empty())

{

assert(use\_count);

const CharType first\_prefix = ubjson\_prefix(j.front());

const bool same\_prefix = std::all\_of(j.begin() + 1, j.end(),

[this, first\_prefix](const BasicJsonType & v)

{

return ubjson\_prefix(v) == first\_prefix;

});

if (same\_prefix)

{

prefix\_required = false;

oa->write\_character(to\_char\_type('$'));

oa->write\_character(first\_prefix);

}

}

if (use\_count)

{

oa->write\_character(to\_char\_type('#'));

write\_number\_with\_ubjson\_prefix(j.m\_value.array->size(), true);

}

for (const auto& el : \*j.m\_value.array)

{

write\_ubjson(el, use\_count, use\_type, prefix\_required);

}

if (not use\_count)

{

oa->write\_character(to\_char\_type(']'));

}

break;

}

case value\_t::object:

{

if (add\_prefix)

{

oa->write\_character(to\_char\_type('{'));

}

bool prefix\_required = true;

if (use\_type and not j.m\_value.object->empty())

{

assert(use\_count);

const CharType first\_prefix = ubjson\_prefix(j.front());

const bool same\_prefix = std::all\_of(j.begin(), j.end(),

[this, first\_prefix](const BasicJsonType & v)

{

return ubjson\_prefix(v) == first\_prefix;

});

if (same\_prefix)

{

prefix\_required = false;

oa->write\_character(to\_char\_type('$'));

oa->write\_character(first\_prefix);

}

}

if (use\_count)

{

oa->write\_character(to\_char\_type('#'));

write\_number\_with\_ubjson\_prefix(j.m\_value.object->size(), true);

}

for (const auto& el : \*j.m\_value.object)

{

write\_number\_with\_ubjson\_prefix(el.first.size(), true);

oa->write\_characters(

reinterpret\_cast<const CharType\*>(el.first.c\_str()),

el.first.size());

write\_ubjson(el.second, use\_count, use\_type, prefix\_required);

}

if (not use\_count)

{

oa->write\_character(to\_char\_type('}'));

}

break;

}

default:

break;

}

}

private:

//////////

// BSON //

//////////

/\*!

@return The size of a BSON document entry header, including the id marker

and the entry name size (and its null-terminator).

\*/

static std::size\_t calc\_bson\_entry\_header\_size(const string\_t& name)

{

const auto it = name.find(static\_cast<typename string\_t::value\_type>(0));

if (JSON\_HEDLEY\_UNLIKELY(it != BasicJsonType::string\_t::npos))

{

JSON\_THROW(out\_of\_range::create(409,

"BSON key cannot contain code point U+0000 (at byte " + std::to\_string(it) + ")"));

}

return /\*id\*/ 1ul + name.size() + /\*zero-terminator\*/1u;

}

/\*!

@brief Writes the given @a element\_type and @a name to the output adapter

\*/

void write\_bson\_entry\_header(const string\_t& name,

const std::uint8\_t element\_type)

{

oa->write\_character(to\_char\_type(element\_type)); // boolean

oa->write\_characters(

reinterpret\_cast<const CharType\*>(name.c\_str()),

name.size() + 1u);

}

/\*!

@brief Writes a BSON element with key @a name and boolean value @a value

\*/

void write\_bson\_boolean(const string\_t& name,

const bool value)

{

write\_bson\_entry\_header(name, 0x08);

oa->write\_character(value ? to\_char\_type(0x01) : to\_char\_type(0x00));

}

/\*!

@brief Writes a BSON element with key @a name and double value @a value

\*/

void write\_bson\_double(const string\_t& name,

const double value)

{

write\_bson\_entry\_header(name, 0x01);

write\_number<double, true>(value);

}

/\*!

@return The size of the BSON-encoded string in @a value

\*/

static std::size\_t calc\_bson\_string\_size(const string\_t& value)

{

return sizeof(std::int32\_t) + value.size() + 1ul;

}

/\*!

@brief Writes a BSON element with key @a name and string value @a value

\*/

void write\_bson\_string(const string\_t& name,

const string\_t& value)

{

write\_bson\_entry\_header(name, 0x02);

write\_number<std::int32\_t, true>(static\_cast<std::int32\_t>(value.size() + 1ul));

oa->write\_characters(

reinterpret\_cast<const CharType\*>(value.c\_str()),

value.size() + 1);

}

/\*!

@brief Writes a BSON element with key @a name and null value

\*/

void write\_bson\_null(const string\_t& name)

{

write\_bson\_entry\_header(name, 0x0A);

}

/\*!

@return The size of the BSON-encoded integer @a value

\*/

static std::size\_t calc\_bson\_integer\_size(const std::int64\_t value)

{

return (std::numeric\_limits<std::int32\_t>::min)() <= value and value <= (std::numeric\_limits<std::int32\_t>::max)()

? sizeof(std::int32\_t)

: sizeof(std::int64\_t);

}

/\*!

@brief Writes a BSON element with key @a name and integer @a value

\*/

void write\_bson\_integer(const string\_t& name,

const std::int64\_t value)

{

if ((std::numeric\_limits<std::int32\_t>::min)() <= value and value <= (std::numeric\_limits<std::int32\_t>::max)())

{

write\_bson\_entry\_header(name, 0x10); // int32

write\_number<std::int32\_t, true>(static\_cast<std::int32\_t>(value));

}

else

{

write\_bson\_entry\_header(name, 0x12); // int64

write\_number<std::int64\_t, true>(static\_cast<std::int64\_t>(value));

}

}

/\*!

@return The size of the BSON-encoded unsigned integer in @a j

\*/

static constexpr std::size\_t calc\_bson\_unsigned\_size(const std::uint64\_t value) noexcept

{

return (value <= static\_cast<std::uint64\_t>((std::numeric\_limits<std::int32\_t>::max)()))

? sizeof(std::int32\_t)

: sizeof(std::int64\_t);

}

/\*!

@brief Writes a BSON element with key @a name and unsigned @a value

\*/

void write\_bson\_unsigned(const string\_t& name,

const std::uint64\_t value)

{

if (value <= static\_cast<std::uint64\_t>((std::numeric\_limits<std::int32\_t>::max)()))

{

write\_bson\_entry\_header(name, 0x10 /\* int32 \*/);

write\_number<std::int32\_t, true>(static\_cast<std::int32\_t>(value));

}

else if (value <= static\_cast<std::uint64\_t>((std::numeric\_limits<std::int64\_t>::max)()))

{

write\_bson\_entry\_header(name, 0x12 /\* int64 \*/);

write\_number<std::int64\_t, true>(static\_cast<std::int64\_t>(value));

}

else

{

JSON\_THROW(out\_of\_range::create(407, "integer number " + std::to\_string(value) + " cannot be represented by BSON as it does not fit int64"));

}

}

/\*!

@brief Writes a BSON element with key @a name and object @a value

\*/

void write\_bson\_object\_entry(const string\_t& name,

const typename BasicJsonType::object\_t& value)

{

write\_bson\_entry\_header(name, 0x03); // object

write\_bson\_object(value);

}

/\*!

@return The size of the BSON-encoded array @a value

\*/

static std::size\_t calc\_bson\_array\_size(const typename BasicJsonType::array\_t& value)

{

std::size\_t array\_index = 0ul;

const std::size\_t embedded\_document\_size = std::accumulate(std::begin(value), std::end(value), 0ul, [&array\_index](std::size\_t result, const typename BasicJsonType::array\_t::value\_type & el)

{

return result + calc\_bson\_element\_size(std::to\_string(array\_index++), el);

});

return sizeof(std::int32\_t) + embedded\_document\_size + 1ul;

}

/\*!

@brief Writes a BSON element with key @a name and array @a value

\*/

void write\_bson\_array(const string\_t& name,

const typename BasicJsonType::array\_t& value)

{

write\_bson\_entry\_header(name, 0x04); // array

write\_number<std::int32\_t, true>(static\_cast<std::int32\_t>(calc\_bson\_array\_size(value)));

std::size\_t array\_index = 0ul;

for (const auto& el : value)

{

write\_bson\_element(std::to\_string(array\_index++), el);

}

oa->write\_character(to\_char\_type(0x00));

}

/\*!

@brief Calculates the size necessary to serialize the JSON value @a j with its @a name

@return The calculated size for the BSON document entry for @a j with the given @a name.

\*/

static std::size\_t calc\_bson\_element\_size(const string\_t& name,

const BasicJsonType& j)

{

const auto header\_size = calc\_bson\_entry\_header\_size(name);

switch (j.type())

{

case value\_t::object:

return header\_size + calc\_bson\_object\_size(\*j.m\_value.object);

case value\_t::array:

return header\_size + calc\_bson\_array\_size(\*j.m\_value.array);

case value\_t::boolean:

return header\_size + 1ul;

case value\_t::number\_float:

return header\_size + 8ul;

case value\_t::number\_integer:

return header\_size + calc\_bson\_integer\_size(j.m\_value.number\_integer);

case value\_t::number\_unsigned:

return header\_size + calc\_bson\_unsigned\_size(j.m\_value.number\_unsigned);

case value\_t::string:

return header\_size + calc\_bson\_string\_size(\*j.m\_value.string);

case value\_t::null:

return header\_size + 0ul;

// LCOV\_EXCL\_START

default:

assert(false);

return 0ul;

// LCOV\_EXCL\_STOP

}

}

/\*!

@brief Serializes the JSON value @a j to BSON and associates it with the

key @a name.

@param name The name to associate with the JSON entity @a j within the

current BSON document

@return The size of the BSON entry

\*/

void write\_bson\_element(const string\_t& name,

const BasicJsonType& j)

{

switch (j.type())

{

case value\_t::object:

return write\_bson\_object\_entry(name, \*j.m\_value.object);

case value\_t::array:

return write\_bson\_array(name, \*j.m\_value.array);

case value\_t::boolean:

return write\_bson\_boolean(name, j.m\_value.boolean);

case value\_t::number\_float:

return write\_bson\_double(name, j.m\_value.number\_float);

case value\_t::number\_integer:

return write\_bson\_integer(name, j.m\_value.number\_integer);

case value\_t::number\_unsigned:

return write\_bson\_unsigned(name, j.m\_value.number\_unsigned);

case value\_t::string:

return write\_bson\_string(name, \*j.m\_value.string);

case value\_t::null:

return write\_bson\_null(name);

// LCOV\_EXCL\_START

default:

assert(false);

return;

// LCOV\_EXCL\_STOP

}

}

/\*!

@brief Calculates the size of the BSON serialization of the given

JSON-object @a j.

@param[in] j JSON value to serialize

@pre j.type() == value\_t::object

\*/

static std::size\_t calc\_bson\_object\_size(const typename BasicJsonType::object\_t& value)

{

std::size\_t document\_size = std::accumulate(value.begin(), value.end(), 0ul,

[](size\_t result, const typename BasicJsonType::object\_t::value\_type & el)

{

return result += calc\_bson\_element\_size(el.first, el.second);

});

return sizeof(std::int32\_t) + document\_size + 1ul;

}

/\*!

@param[in] j JSON value to serialize

@pre j.type() == value\_t::object

\*/

void write\_bson\_object(const typename BasicJsonType::object\_t& value)

{

write\_number<std::int32\_t, true>(static\_cast<std::int32\_t>(calc\_bson\_object\_size(value)));

for (const auto& el : value)

{

write\_bson\_element(el.first, el.second);

}

oa->write\_character(to\_char\_type(0x00));

}

//////////

// CBOR //

//////////

static constexpr CharType get\_cbor\_float\_prefix(float /\*unused\*/)

{

return to\_char\_type(0xFA); // Single-Precision Float

}

static constexpr CharType get\_cbor\_float\_prefix(double /\*unused\*/)

{

return to\_char\_type(0xFB); // Double-Precision Float

}

/////////////

// MsgPack //

/////////////

static constexpr CharType get\_msgpack\_float\_prefix(float /\*unused\*/)

{

return to\_char\_type(0xCA); // float 32

}

static constexpr CharType get\_msgpack\_float\_prefix(double /\*unused\*/)

{

return to\_char\_type(0xCB); // float 64

}

////////////

// UBJSON //

////////////

// UBJSON: write number (floating point)

template<typename NumberType, typename std::enable\_if<

std::is\_floating\_point<NumberType>::value, int>::type = 0>

void write\_number\_with\_ubjson\_prefix(const NumberType n,

const bool add\_prefix)

{

if (add\_prefix)

{

oa->write\_character(get\_ubjson\_float\_prefix(n));

}

write\_number(n);

}

// UBJSON: write number (unsigned integer)

template<typename NumberType, typename std::enable\_if<

std::is\_unsigned<NumberType>::value, int>::type = 0>

void write\_number\_with\_ubjson\_prefix(const NumberType n,

const bool add\_prefix)

{

if (n <= static\_cast<std::uint64\_t>((std::numeric\_limits<std::int8\_t>::max)()))

{

if (add\_prefix)

{

oa->write\_character(to\_char\_type('i')); // int8

}

write\_number(static\_cast<std::uint8\_t>(n));

}

else if (n <= (std::numeric\_limits<std::uint8\_t>::max)())

{

if (add\_prefix)

{

oa->write\_character(to\_char\_type('U')); // uint8

}

write\_number(static\_cast<std::uint8\_t>(n));

}

else if (n <= static\_cast<std::uint64\_t>((std::numeric\_limits<std::int16\_t>::max)()))

{

if (add\_prefix)

{

oa->write\_character(to\_char\_type('I')); // int16

}

write\_number(static\_cast<std::int16\_t>(n));

}

else if (n <= static\_cast<std::uint64\_t>((std::numeric\_limits<std::int32\_t>::max)()))

{

if (add\_prefix)

{

oa->write\_character(to\_char\_type('l')); // int32

}

write\_number(static\_cast<std::int32\_t>(n));

}

else if (n <= static\_cast<std::uint64\_t>((std::numeric\_limits<std::int64\_t>::max)()))

{

if (add\_prefix)

{

oa->write\_character(to\_char\_type('L')); // int64

}

write\_number(static\_cast<std::int64\_t>(n));

}

else

{

JSON\_THROW(out\_of\_range::create(407, "integer number " + std::to\_string(n) + " cannot be represented by UBJSON as it does not fit int64"));

}

}

// UBJSON: write number (signed integer)

template<typename NumberType, typename std::enable\_if<

std::is\_signed<NumberType>::value and

not std::is\_floating\_point<NumberType>::value, int>::type = 0>

void write\_number\_with\_ubjson\_prefix(const NumberType n,

const bool add\_prefix)

{

if ((std::numeric\_limits<std::int8\_t>::min)() <= n and n <= (std::numeric\_limits<std::int8\_t>::max)())

{

if (add\_prefix)

{

oa->write\_character(to\_char\_type('i')); // int8

}

write\_number(static\_cast<std::int8\_t>(n));

}

else if (static\_cast<std::int64\_t>((std::numeric\_limits<std::uint8\_t>::min)()) <= n and n <= static\_cast<std::int64\_t>((std::numeric\_limits<std::uint8\_t>::max)()))

{

if (add\_prefix)

{

oa->write\_character(to\_char\_type('U')); // uint8

}

write\_number(static\_cast<std::uint8\_t>(n));

}

else if ((std::numeric\_limits<std::int16\_t>::min)() <= n and n <= (std::numeric\_limits<std::int16\_t>::max)())

{

if (add\_prefix)

{

oa->write\_character(to\_char\_type('I')); // int16

}

write\_number(static\_cast<std::int16\_t>(n));

}

else if ((std::numeric\_limits<std::int32\_t>::min)() <= n and n <= (std::numeric\_limits<std::int32\_t>::max)())

{

if (add\_prefix)

{

oa->write\_character(to\_char\_type('l')); // int32

}

write\_number(static\_cast<std::int32\_t>(n));

}

else if ((std::numeric\_limits<std::int64\_t>::min)() <= n and n <= (std::numeric\_limits<std::int64\_t>::max)())

{

if (add\_prefix)

{

oa->write\_character(to\_char\_type('L')); // int64

}

write\_number(static\_cast<std::int64\_t>(n));

}

// LCOV\_EXCL\_START

else

{

JSON\_THROW(out\_of\_range::create(407, "integer number " + std::to\_string(n) + " cannot be represented by UBJSON as it does not fit int64"));

}

// LCOV\_EXCL\_STOP

}

/\*!

@brief determine the type prefix of container values

@note This function does not need to be 100% accurate when it comes to

integer limits. In case a number exceeds the limits of int64\_t,

this will be detected by a later call to function

write\_number\_with\_ubjson\_prefix. Therefore, we return 'L' for any

value that does not fit the previous limits.

\*/

CharType ubjson\_prefix(const BasicJsonType& j) const noexcept

{

switch (j.type())

{

case value\_t::null:

return 'Z';

case value\_t::boolean:

return j.m\_value.boolean ? 'T' : 'F';

case value\_t::number\_integer:

{

if ((std::numeric\_limits<std::int8\_t>::min)() <= j.m\_value.number\_integer and j.m\_value.number\_integer <= (std::numeric\_limits<std::int8\_t>::max)())

{

return 'i';

}

if ((std::numeric\_limits<std::uint8\_t>::min)() <= j.m\_value.number\_integer and j.m\_value.number\_integer <= (std::numeric\_limits<std::uint8\_t>::max)())

{

return 'U';

}

if ((std::numeric\_limits<std::int16\_t>::min)() <= j.m\_value.number\_integer and j.m\_value.number\_integer <= (std::numeric\_limits<std::int16\_t>::max)())

{

return 'I';

}

if ((std::numeric\_limits<std::int32\_t>::min)() <= j.m\_value.number\_integer and j.m\_value.number\_integer <= (std::numeric\_limits<std::int32\_t>::max)())

{

return 'l';

}

// no check and assume int64\_t (see note above)

return 'L';

}

case value\_t::number\_unsigned:

{

if (j.m\_value.number\_unsigned <= static\_cast<std::uint64\_t>((std::numeric\_limits<std::int8\_t>::max)()))

{

return 'i';

}

if (j.m\_value.number\_unsigned <= static\_cast<std::uint64\_t>((std::numeric\_limits<std::uint8\_t>::max)()))

{

return 'U';

}

if (j.m\_value.number\_unsigned <= static\_cast<std::uint64\_t>((std::numeric\_limits<std::int16\_t>::max)()))

{

return 'I';

}

if (j.m\_value.number\_unsigned <= static\_cast<std::uint64\_t>((std::numeric\_limits<std::int32\_t>::max)()))

{

return 'l';

}

// no check and assume int64\_t (see note above)

return 'L';

}

case value\_t::number\_float:

return get\_ubjson\_float\_prefix(j.m\_value.number\_float);

case value\_t::string:

return 'S';

case value\_t::array:

return '[';

case value\_t::object:

return '{';

default: // discarded values

return 'N';

}

}

static constexpr CharType get\_ubjson\_float\_prefix(float /\*unused\*/)

{

return 'd'; // float 32

}

static constexpr CharType get\_ubjson\_float\_prefix(double /\*unused\*/)

{

return 'D'; // float 64

}

///////////////////////

// Utility functions //

///////////////////////

/\*

@brief write a number to output input

@param[in] n number of type @a NumberType

@tparam NumberType the type of the number

@tparam OutputIsLittleEndian Set to true if output data is

required to be little endian

@note This function needs to respect the system's endianess, because bytes

in CBOR, MessagePack, and UBJSON are stored in network order (big

endian) and therefore need reordering on little endian systems.

\*/

template<typename NumberType, bool OutputIsLittleEndian = false>

void write\_number(const NumberType n)

{

// step 1: write number to array of length NumberType

std::array<CharType, sizeof(NumberType)> vec;

std::memcpy(vec.data(), &n, sizeof(NumberType));

// step 2: write array to output (with possible reordering)

if (is\_little\_endian != OutputIsLittleEndian)

{

// reverse byte order prior to conversion if necessary

std::reverse(vec.begin(), vec.end());

}

oa->write\_characters(vec.data(), sizeof(NumberType));

}

public:

// The following to\_char\_type functions are implement the conversion

// between uint8\_t and CharType. In case CharType is not unsigned,

// such a conversion is required to allow values greater than 128.

// See <https://github.com/nlohmann/json/issues/1286> for a discussion.

template < typename C = CharType,

enable\_if\_t < std::is\_signed<C>::value and std::is\_signed<char>::value > \* = nullptr >

static constexpr CharType to\_char\_type(std::uint8\_t x) noexcept

{

return \*reinterpret\_cast<char\*>(&x);

}

template < typename C = CharType,

enable\_if\_t < std::is\_signed<C>::value and std::is\_unsigned<char>::value > \* = nullptr >

static CharType to\_char\_type(std::uint8\_t x) noexcept

{

static\_assert(sizeof(std::uint8\_t) == sizeof(CharType), "size of CharType must be equal to std::uint8\_t");

static\_assert(std::is\_pod<CharType>::value, "CharType must be POD");

CharType result;

std::memcpy(&result, &x, sizeof(x));

return result;

}

template<typename C = CharType,

enable\_if\_t<std::is\_unsigned<C>::value>\* = nullptr>

static constexpr CharType to\_char\_type(std::uint8\_t x) noexcept

{

return x;

}

template < typename InputCharType, typename C = CharType,

enable\_if\_t <

std::is\_signed<C>::value and

std::is\_signed<char>::value and

std::is\_same<char, typename std::remove\_cv<InputCharType>::type>::value

> \* = nullptr >

static constexpr CharType to\_char\_type(InputCharType x) noexcept

{

return x;

}

private:

/// whether we can assume little endianess

const bool is\_little\_endian = binary\_reader<BasicJsonType>::little\_endianess();

/// the output

output\_adapter\_t<CharType> oa = nullptr;

};

} // namespace detail

} // namespace nlohmann

// #include <nlohmann/detail/output/output\_adapters.hpp>

// #include <nlohmann/detail/output/serializer.hpp>

#include <algorithm> // reverse, remove, fill, find, none\_of

#include <array> // array

#include <cassert> // assert

#include <ciso646> // and, or

#include <clocale> // localeconv, lconv

#include <cmath> // labs, isfinite, isnan, signbit

#include <cstddef> // size\_t, ptrdiff\_t

#include <cstdint> // uint8\_t

#include <cstdio> // snprintf

#include <limits> // numeric\_limits

#include <string> // string

#include <type\_traits> // is\_same

#include <utility> // move

// #include <nlohmann/detail/conversions/to\_chars.hpp>

#include <array> // array

#include <cassert> // assert

#include <ciso646> // or, and, not

#include <cmath> // signbit, isfinite

#include <cstdint> // intN\_t, uintN\_t

#include <cstring> // memcpy, memmove

#include <limits> // numeric\_limits

#include <type\_traits> // conditional

// #include <nlohmann/detail/macro\_scope.hpp>

namespace nlohmann

{

namespace detail

{

/\*!

@brief 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

\*/

namespace dtoa\_impl

{

template <typename Target, typename Source>

Target reinterpret\_bits(const Source source)

{

static\_assert(sizeof(Target) == sizeof(Source), "size mismatch");

Target target;

std::memcpy(&target, &source, sizeof(Source));

return target;

}

struct diyfp // f \* 2^e

{

static constexpr int kPrecision = 64; // = q

std::uint64\_t f = 0;

int e = 0;

constexpr diyfp(std::uint64\_t f\_, int e\_) noexcept : f(f\_), e(e\_) {}

/\*!

@brief returns x - y

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

\*/

static diyfp sub(const diyfp& x, const diyfp& y) noexcept

{

assert(x.e == y.e);

assert(x.f >= y.f);

return {x.f - y.f, x.e};

}

/\*!

@brief returns x \* y

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

\*/

static diyfp mul(const diyfp& x, const diyfp& y) noexcept

{

static\_assert(kPrecision == 64, "internal error");

// Computes:

// f = round((x.f \* y.f) / 2^q)

// e = x.e + y.e + q

// Emulate the 64-bit \* 64-bit multiplication:

//

// p = u \* v

// = (u\_lo + 2^32 u\_hi) (v\_lo + 2^32 v\_hi)

// = (u\_lo v\_lo ) + 2^32 ((u\_lo v\_hi ) + (u\_hi v\_lo )) + 2^64 (u\_hi v\_hi )

// = (p0 ) + 2^32 ((p1 ) + (p2 )) + 2^64 (p3 )

// = (p0\_lo + 2^32 p0\_hi) + 2^32 ((p1\_lo + 2^32 p1\_hi) + (p2\_lo + 2^32 p2\_hi)) + 2^64 (p3 )

// = (p0\_lo ) + 2^32 (p0\_hi + p1\_lo + p2\_lo ) + 2^64 (p1\_hi + p2\_hi + p3)

// = (p0\_lo ) + 2^32 (Q ) + 2^64 (H )

// = (p0\_lo ) + 2^32 (Q\_lo + 2^32 Q\_hi ) + 2^64 (H )

//

// (Since Q might be larger than 2^32 - 1)

//

// = (p0\_lo + 2^32 Q\_lo) + 2^64 (Q\_hi + H)

//

// (Q\_hi + H does not overflow a 64-bit int)

//

// = p\_lo + 2^64 p\_hi

const std::uint64\_t u\_lo = x.f & 0xFFFFFFFFu;

const std::uint64\_t u\_hi = x.f >> 32u;

const std::uint64\_t v\_lo = y.f & 0xFFFFFFFFu;

const std::uint64\_t v\_hi = y.f >> 32u;

const std::uint64\_t p0 = u\_lo \* v\_lo;

const std::uint64\_t p1 = u\_lo \* v\_hi;

const std::uint64\_t p2 = u\_hi \* v\_lo;

const std::uint64\_t p3 = u\_hi \* v\_hi;

const std::uint64\_t p0\_hi = p0 >> 32u;

const std::uint64\_t p1\_lo = p1 & 0xFFFFFFFFu;

const std::uint64\_t p1\_hi = p1 >> 32u;

const std::uint64\_t p2\_lo = p2 & 0xFFFFFFFFu;

const std::uint64\_t p2\_hi = p2 >> 32u;

std::uint64\_t Q = p0\_hi + p1\_lo + p2\_lo;

// The full product might now be computed as

//

// p\_hi = p3 + p2\_hi + p1\_hi + (Q >> 32)

// p\_lo = p0\_lo + (Q << 32)

//

// But in this particular case here, the full p\_lo is not required.

// Effectively we only need to add the highest bit in p\_lo to p\_hi (and

// Q\_hi + 1 does not overflow).

Q += std::uint64\_t{1} << (64u - 32u - 1u); // round, ties up

const std::uint64\_t h = p3 + p2\_hi + p1\_hi + (Q >> 32u);

return {h, x.e + y.e + 64};

}

/\*!

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

@pre x.f != 0

\*/

static diyfp normalize(diyfp x) noexcept

{

assert(x.f != 0);

while ((x.f >> 63u) == 0)

{

x.f <<= 1u;

x.e--;

}

return x;

}

/\*!

@brief normalize x such that the result has the exponent E

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

\*/

static diyfp normalize\_to(const diyfp& x, const int target\_exponent) noexcept

{

const int delta = x.e - target\_exponent;

assert(delta >= 0);

assert(((x.f << delta) >> delta) == x.f);

return {x.f << delta, target\_exponent};

}

};

struct boundaries

{

diyfp w;

diyfp minus;

diyfp plus;

};

/\*!

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

boundaries.

@pre value must be finite and positive

\*/

template <typename FloatType>

boundaries compute\_boundaries(FloatType value)

{

assert(std::isfinite(value));

assert(value > 0);

// Convert the IEEE representation into a diyfp.

//

// If v is denormal:

// value = 0.F \* 2^(1 - bias) = ( F) \* 2^(1 - bias - (p-1))

// If v is normalized:

// value = 1.F \* 2^(E - bias) = (2^(p-1) + F) \* 2^(E - bias - (p-1))

static\_assert(std::numeric\_limits<FloatType>::is\_iec559,

"internal error: dtoa\_short requires an IEEE-754 floating-point implementation");

constexpr int kPrecision = std::numeric\_limits<FloatType>::digits; // = p (includes the hidden bit)

constexpr int kBias = std::numeric\_limits<FloatType>::max\_exponent - 1 + (kPrecision - 1);

constexpr int kMinExp = 1 - kBias;

constexpr std::uint64\_t kHiddenBit = std::uint64\_t{1} << (kPrecision - 1); // = 2^(p-1)

using bits\_type = typename std::conditional<kPrecision == 24, std::uint32\_t, std::uint64\_t >::type;

const std::uint64\_t bits = reinterpret\_bits<bits\_type>(value);

const std::uint64\_t E = bits >> (kPrecision - 1);

const std::uint64\_t F = bits & (kHiddenBit - 1);

const bool is\_denormal = E == 0;

const diyfp v = is\_denormal

? diyfp(F, kMinExp)

: diyfp(F + kHiddenBit, static\_cast<int>(E) - kBias);

// Compute the boundaries m- and m+ of the floating-point value

// v = f \* 2^e.

//

// Determine v- and v+, the floating-point predecessor and successor if v,

// respectively.

//

// v- = v - 2^e if f != 2^(p-1) or e == e\_min (A)

// = v - 2^(e-1) if f == 2^(p-1) and e > e\_min (B)

//

// v+ = v + 2^e

//

// Let m- = (v- + v) / 2 and m+ = (v + v+) / 2. All real numbers \_strictly\_

// between m- and m+ round to v, regardless of how the input rounding

// algorithm breaks ties.

//

// ---+-------------+-------------+-------------+-------------+--- (A)

// v- m- v m+ v+

//

// -----------------+------+------+-------------+-------------+--- (B)

// v- m- v m+ v+

const bool lower\_boundary\_is\_closer = F == 0 and E > 1;

const diyfp m\_plus = diyfp(2 \* v.f + 1, v.e - 1);

const diyfp m\_minus = lower\_boundary\_is\_closer

? diyfp(4 \* v.f - 1, v.e - 2) // (B)

: diyfp(2 \* v.f - 1, v.e - 1); // (A)

// Determine the normalized w+ = m+.

const diyfp w\_plus = diyfp::normalize(m\_plus);

// Determine w- = m- such that e\_(w-) = e\_(w+).

const diyfp w\_minus = diyfp::normalize\_to(m\_minus, w\_plus.e);

return {diyfp::normalize(v), w\_minus, w\_plus};

}

// Given normalized diyfp w, Grisu needs to find a (normalized) cached

// power-of-ten c, such that the exponent of the product c \* w = f \* 2^e lies

// within a certain range [alpha, gamma] (Definition 3.2 from [1])

//

// alpha <= e = e\_c + e\_w + q <= gamma

//

// or

//

// f\_c \* f\_w \* 2^alpha <= f\_c 2^(e\_c) \* f\_w 2^(e\_w) \* 2^q

// <= f\_c \* f\_w \* 2^gamma

//

// Since c and w are normalized, i.e. 2^(q-1) <= f < 2^q, this implies

//

// 2^(q-1) \* 2^(q-1) \* 2^alpha <= c \* w \* 2^q < 2^q \* 2^q \* 2^gamma

//

// or

//

// 2^(q - 2 + alpha) <= c \* w < 2^(q + gamma)

//

// The choice of (alpha,gamma) determines the size of the table and the form of

// the digit generation procedure. Using (alpha,gamma)=(-60,-32) works out well

// in practice:

//

// The idea is to cut the number c \* w = f \* 2^e into two parts, which can be

// processed independently: An integral part p1, and a fractional part p2:

//

// f \* 2^e = ( (f div 2^-e) \* 2^-e + (f mod 2^-e) ) \* 2^e

// = (f div 2^-e) + (f mod 2^-e) \* 2^e

// = p1 + p2 \* 2^e

//

// The conversion of p1 into decimal form requires a series of divisions and

// modulos by (a power of) 10. These operations are faster for 32-bit than for

// 64-bit integers, so p1 should ideally fit into a 32-bit integer. This can be

// achieved by choosing

//

// -e >= 32 or e <= -32 := gamma

//

// In order to convert the fractional part

//

// p2 \* 2^e = p2 / 2^-e = d[-1] / 10^1 + d[-2] / 10^2 + ...

//

// into decimal form, the fraction is repeatedly multiplied by 10 and the digits

// d[-i] are extracted in order:

//

// (10 \* p2) div 2^-e = d[-1]

// (10 \* p2) mod 2^-e = d[-2] / 10^1 + ...

//

// The multiplication by 10 must not overflow. It is sufficient to choose

//

// 10 \* p2 < 16 \* p2 = 2^4 \* p2 <= 2^64.

//

// Since p2 = f mod 2^-e < 2^-e,

//

// -e <= 60 or e >= -60 := alpha

constexpr int kAlpha = -60;

constexpr int kGamma = -32;

struct cached\_power // c = f \* 2^e ~= 10^k

{

std::uint64\_t f;

int e;

int k;

};

/\*!

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.

\*/

inline cached\_power get\_cached\_power\_for\_binary\_exponent(int e)

{

// Now

//

// alpha <= e\_c + e + q <= gamma (1)

// ==> f\_c \* 2^alpha <= c \* 2^e \* 2^q

//

// and since the c's are normalized, 2^(q-1) <= f\_c,

//

// ==> 2^(q - 1 + alpha) <= c \* 2^(e + q)

// ==> 2^(alpha - e - 1) <= c

//

// If c were an exact power of ten, i.e. c = 10^k, one may determine k as

//

// k = ceil( log\_10( 2^(alpha - e - 1) ) )

// = ceil( (alpha - e - 1) \* log\_10(2) )

//

// From the paper:

// "In theory the result of the procedure could be wrong since c is rounded,

// and the computation itself is approximated [...]. In practice, however,

// this simple function is sufficient."

//

// For IEEE double precision floating-point numbers converted into

// normalized diyfp's w = f \* 2^e, with q = 64,

//

// e >= -1022 (min IEEE exponent)

// -52 (p - 1)

// -52 (p - 1, possibly normalize denormal IEEE numbers)

// -11 (normalize the diyfp)

// = -1137

//

// and

//

// e <= +1023 (max IEEE exponent)

// -52 (p - 1)

// -11 (normalize the diyfp)

// = 960

//

// This binary exponent range [-1137,960] results in a decimal exponent

// range [-307,324]. One does not need to store a cached power for each

// k in this range. For each such k it suffices to find a cached power

// such that the exponent of the product lies in [alpha,gamma].

// This implies that the difference of the decimal exponents of adjacent

// table entries must be less than or equal to

//

// floor( (gamma - alpha) \* log\_10(2) ) = 8.

//

// (A smaller distance gamma-alpha would require a larger table.)

// NB:

// Actually this function returns c, such that -60 <= e\_c + e + 64 <= -34.

constexpr int kCachedPowersMinDecExp = -300;

constexpr int kCachedPowersDecStep = 8;

static constexpr std::array<cached\_power, 79> kCachedPowers =

{

{

{ 0xAB70FE17C79AC6CA, -1060, -300 },

{ 0xFF77B1FCBEBCDC4F, -1034, -292 },

{ 0xBE5691EF416BD60C, -1007, -284 },

{ 0x8DD01FAD907FFC3C, -980, -276 },

{ 0xD3515C2831559A83, -954, -268 },

{ 0x9D71AC8FADA6C9B5, -927, -260 },

{ 0xEA9C227723EE8BCB, -901, -252 },

{ 0xAECC49914078536D, -874, -244 },

{ 0x823C12795DB6CE57, -847, -236 },

{ 0xC21094364DFB5637, -821, -228 },

{ 0x9096EA6F3848984F, -794, -220 },

{ 0xD77485CB25823AC7, -768, -212 },

{ 0xA086CFCD97BF97F4, -741, -204 },

{ 0xEF340A98172AACE5, -715, -196 },

{ 0xB23867FB2A35B28E, -688, -188 },

{ 0x84C8D4DFD2C63F3B, -661, -180 },

{ 0xC5DD44271AD3CDBA, -635, -172 },

{ 0x936B9FCEBB25C996, -608, -164 },

{ 0xDBAC6C247D62A584, -582, -156 },

{ 0xA3AB66580D5FDAF6, -555, -148 },

{ 0xF3E2F893DEC3F126, -529, -140 },

{ 0xB5B5ADA8AAFF80B8, -502, -132 },

{ 0x87625F056C7C4A8B, -475, -124 },

{ 0xC9BCFF6034C13053, -449, -116 },

{ 0x964E858C91BA2655, -422, -108 },

{ 0xDFF9772470297EBD, -396, -100 },

{ 0xA6DFBD9FB8E5B88F, -369, -92 },

{ 0xF8A95FCF88747D94, -343, -84 },

{ 0xB94470938FA89BCF, -316, -76 },

{ 0x8A08F0F8BF0F156B, -289, -68 },

{ 0xCDB02555653131B6, -263, -60 },

{ 0x993FE2C6D07B7FAC, -236, -52 },

{ 0xE45C10C42A2B3B06, -210, -44 },

{ 0xAA242499697392D3, -183, -36 },

{ 0xFD87B5F28300CA0E, -157, -28 },

{ 0xBCE5086492111AEB, -130, -20 },

{ 0x8CBCCC096F5088CC, -103, -12 },

{ 0xD1B71758E219652C, -77, -4 },

{ 0x9C40000000000000, -50, 4 },

{ 0xE8D4A51000000000, -24, 12 },

{ 0xAD78EBC5AC620000, 3, 20 },

{ 0x813F3978F8940984, 30, 28 },

{ 0xC097CE7BC90715B3, 56, 36 },

{ 0x8F7E32CE7BEA5C70, 83, 44 },

{ 0xD5D238A4ABE98068, 109, 52 },

{ 0x9F4F2726179A2245, 136, 60 },

{ 0xED63A231D4C4FB27, 162, 68 },

{ 0xB0DE65388CC8ADA8, 189, 76 },

{ 0x83C7088E1AAB65DB, 216, 84 },

{ 0xC45D1DF942711D9A, 242, 92 },

{ 0x924D692CA61BE758, 269, 100 },

{ 0xDA01EE641A708DEA, 295, 108 },

{ 0xA26DA3999AEF774A, 322, 116 },

{ 0xF209787BB47D6B85, 348, 124 },

{ 0xB454E4A179DD1877, 375, 132 },

{ 0x865B86925B9BC5C2, 402, 140 },

{ 0xC83553C5C8965D3D, 428, 148 },

{ 0x952AB45CFA97A0B3, 455, 156 },

{ 0xDE469FBD99A05FE3, 481, 164 },

{ 0xA59BC234DB398C25, 508, 172 },

{ 0xF6C69A72A3989F5C, 534, 180 },

{ 0xB7DCBF5354E9BECE, 561, 188 },

{ 0x88FCF317F22241E2, 588, 196 },

{ 0xCC20CE9BD35C78A5, 614, 204 },

{ 0x98165AF37B2153DF, 641, 212 },

{ 0xE2A0B5DC971F303A, 667, 220 },

{ 0xA8D9D1535CE3B396, 694, 228 },

{ 0xFB9B7CD9A4A7443C, 720, 236 },

{ 0xBB764C4CA7A44410, 747, 244 },

{ 0x8BAB8EEFB6409C1A, 774, 252 },

{ 0xD01FEF10A657842C, 800, 260 },

{ 0x9B10A4E5E9913129, 827, 268 },

{ 0xE7109BFBA19C0C9D, 853, 276 },

{ 0xAC2820D9623BF429, 880, 284 },

{ 0x80444B5E7AA7CF85, 907, 292 },

{ 0xBF21E44003ACDD2D, 933, 300 },

{ 0x8E679C2F5E44FF8F, 960, 308 },

{ 0xD433179D9C8CB841, 986, 316 },

{ 0x9E19DB92B4E31BA9, 1013, 324 },

}

};

// This computation gives exactly the same results for k as

// k = ceil((kAlpha - e - 1) \* 0.30102999566398114)

// for |e| <= 1500, but doesn't require floating-point operations.

// NB: log\_10(2) ~= 78913 / 2^18

assert(e >= -1500);

assert(e <= 1500);

const int f = kAlpha - e - 1;

const int k = (f \* 78913) / (1 << 18) + static\_cast<int>(f > 0);

const int index = (-kCachedPowersMinDecExp + k + (kCachedPowersDecStep - 1)) / kCachedPowersDecStep;

assert(index >= 0);

assert(static\_cast<std::size\_t>(index) < kCachedPowers.size());

const cached\_power cached = kCachedPowers[static\_cast<std::size\_t>(index)];

assert(kAlpha <= cached.e + e + 64);

assert(kGamma >= cached.e + e + 64);

return cached;

}

/\*!

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

For n == 0, returns 1 and sets pow10 := 1.

\*/

inline int find\_largest\_pow10(const std::uint32\_t n, std::uint32\_t& pow10)

{

// LCOV\_EXCL\_START

if (n >= 1000000000)

{

pow10 = 1000000000;

return 10;

}

// LCOV\_EXCL\_STOP

else if (n >= 100000000)

{

pow10 = 100000000;

return 9;

}

else if (n >= 10000000)

{

pow10 = 10000000;

return 8;

}

else if (n >= 1000000)

{

pow10 = 1000000;

return 7;

}

else if (n >= 100000)

{

pow10 = 100000;

return 6;

}

else if (n >= 10000)

{

pow10 = 10000;

return 5;

}

else if (n >= 1000)

{

pow10 = 1000;

return 4;

}

else if (n >= 100)

{

pow10 = 100;

return 3;

}

else if (n >= 10)

{

pow10 = 10;

return 2;

}

else

{

pow10 = 1;

return 1;

}

}

inline void grisu2\_round(char\* buf, int len, std::uint64\_t dist, std::uint64\_t delta,

std::uint64\_t rest, std::uint64\_t ten\_k)

{

assert(len >= 1);

assert(dist <= delta);

assert(rest <= delta);

assert(ten\_k > 0);

// <--------------------------- delta ---->

// <---- dist --------->

// --------------[------------------+-------------------]--------------

// M- w M+

//

// ten\_k

// <------>

// <---- rest ---->

// --------------[------------------+----+--------------]--------------

// w V

// = buf \* 10^k

//

// ten\_k represents a unit-in-the-last-place in the decimal representation

// stored in buf.

// Decrement buf by ten\_k while this takes buf closer to w.

// The tests are written in this order to avoid overflow in unsigned

// integer arithmetic.

while (rest < dist

and delta - rest >= ten\_k

and (rest + ten\_k < dist or dist - rest > rest + ten\_k - dist))

{

assert(buf[len - 1] != '0');

buf[len - 1]--;

rest += ten\_k;

}

}

/\*!

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.

\*/

inline void grisu2\_digit\_gen(char\* buffer, int& length, int& decimal\_exponent,

diyfp M\_minus, diyfp w, diyfp M\_plus)

{

static\_assert(kAlpha >= -60, "internal error");

static\_assert(kGamma <= -32, "internal error");

// Generates the digits (and the exponent) of a decimal floating-point

// number V = buffer \* 10^decimal\_exponent in the range [M-, M+]. The diyfp's

// w, M- and M+ share the same exponent e, which satisfies alpha <= e <= gamma.

//

// <--------------------------- delta ---->

// <---- dist --------->

// --------------[------------------+-------------------]--------------

// M- w M+

//

// Grisu2 generates the digits of M+ from left to right and stops as soon as

// V is in [M-,M+].

assert(M\_plus.e >= kAlpha);

assert(M\_plus.e <= kGamma);

std::uint64\_t delta = diyfp::sub(M\_plus, M\_minus).f; // (significand of (M+ - M-), implicit exponent is e)

std::uint64\_t dist = diyfp::sub(M\_plus, w ).f; // (significand of (M+ - w ), implicit exponent is e)

// Split M+ = f \* 2^e into two parts p1 and p2 (note: e < 0):

//

// M+ = f \* 2^e

// = ((f div 2^-e) \* 2^-e + (f mod 2^-e)) \* 2^e

// = ((p1 ) \* 2^-e + (p2 )) \* 2^e

// = p1 + p2 \* 2^e

const diyfp one(std::uint64\_t{1} << -M\_plus.e, M\_plus.e);

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.)

std::uint64\_t p2 = M\_plus.f & (one.f - 1); // p2 = f mod 2^-e

// 1)

//

// Generate the digits of the integral part p1 = d[n-1]...d[1]d[0]

assert(p1 > 0);

std::uint32\_t pow10;

const int k = find\_largest\_pow10(p1, pow10);

// 10^(k-1) <= p1 < 10^k, pow10 = 10^(k-1)

//

// p1 = (p1 div 10^(k-1)) \* 10^(k-1) + (p1 mod 10^(k-1))

// = (d[k-1] ) \* 10^(k-1) + (p1 mod 10^(k-1))

//

// M+ = p1 + p2 \* 2^e

// = d[k-1] \* 10^(k-1) + (p1 mod 10^(k-1)) + p2 \* 2^e

// = d[k-1] \* 10^(k-1) + ((p1 mod 10^(k-1)) \* 2^-e + p2) \* 2^e

// = d[k-1] \* 10^(k-1) + ( rest) \* 2^e

//

// Now generate the digits d[n] of p1 from left to right (n = k-1,...,0)

//

// p1 = d[k-1]...d[n] \* 10^n + d[n-1]...d[0]

//

// but stop as soon as

//

// rest \* 2^e = (d[n-1]...d[0] \* 2^-e + p2) \* 2^e <= delta \* 2^e

int n = k;

while (n > 0)

{

// Invariants:

// M+ = buffer \* 10^n + (p1 + p2 \* 2^e) (buffer = 0 for n = k)

// pow10 = 10^(n-1) <= p1 < 10^n

//

const std::uint32\_t d = p1 / pow10; // d = p1 div 10^(n-1)

const std::uint32\_t r = p1 % pow10; // r = p1 mod 10^(n-1)

//

// M+ = buffer \* 10^n + (d \* 10^(n-1) + r) + p2 \* 2^e

// = (buffer \* 10 + d) \* 10^(n-1) + (r + p2 \* 2^e)

//

assert(d <= 9);

buffer[length++] = static\_cast<char>('0' + d); // buffer := buffer \* 10 + d

//

// M+ = buffer \* 10^(n-1) + (r + p2 \* 2^e)

//

p1 = r;

n--;

//

// M+ = buffer \* 10^n + (p1 + p2 \* 2^e)

// pow10 = 10^n

//

// Now check if enough digits have been generated.

// Compute

//

// p1 + p2 \* 2^e = (p1 \* 2^-e + p2) \* 2^e = rest \* 2^e

//

// Note:

// Since rest and delta share the same exponent e, it suffices to

// compare the significands.

const std::uint64\_t rest = (std::uint64\_t{p1} << -one.e) + p2;

if (rest <= delta)

{

// V = buffer \* 10^n, with M- <= V <= M+.

decimal\_exponent += n;

// We may now just stop. But instead look if the buffer could be

// decremented to bring V closer to w.

//

// pow10 = 10^n is now 1 ulp in the decimal representation V.

// The rounding procedure works with diyfp's with an implicit

// exponent of e.

//

// 10^n = (10^n \* 2^-e) \* 2^e = ulp \* 2^e

//

const std::uint64\_t ten\_n = std::uint64\_t{pow10} << -one.e;

grisu2\_round(buffer, length, dist, delta, rest, ten\_n);

return;

}

pow10 /= 10;

//

// pow10 = 10^(n-1) <= p1 < 10^n

// Invariants restored.

}

// 2)

//

// The digits of the integral part have been generated:

//

// M+ = d[k-1]...d[1]d[0] + p2 \* 2^e

// = buffer + p2 \* 2^e

//

// Now generate the digits of the fractional part p2 \* 2^e.

//

// Note:

// No decimal point is generated: the exponent is adjusted instead.

//

// p2 actually represents the fraction

//

// p2 \* 2^e

// = p2 / 2^-e

// = d[-1] / 10^1 + d[-2] / 10^2 + ...

//

// Now generate the digits d[-m] of p1 from left to right (m = 1,2,...)

//

// p2 \* 2^e = d[-1]d[-2]...d[-m] \* 10^-m

// + 10^-m \* (d[-m-1] / 10^1 + d[-m-2] / 10^2 + ...)

//

// using

//

// 10^m \* p2 = ((10^m \* p2) div 2^-e) \* 2^-e + ((10^m \* p2) mod 2^-e)

// = ( d) \* 2^-e + ( r)

//

// or

// 10^m \* p2 \* 2^e = d + r \* 2^e

//

// i.e.

//

// M+ = buffer + p2 \* 2^e

// = buffer + 10^-m \* (d + r \* 2^e)

// = (buffer \* 10^m + d) \* 10^-m + 10^-m \* r \* 2^e

//

// and stop as soon as 10^-m \* r \* 2^e <= delta \* 2^e

assert(p2 > delta);

int m = 0;

for (;;)

{

// Invariant:

// M+ = buffer \* 10^-m + 10^-m \* (d[-m-1] / 10 + d[-m-2] / 10^2 + ...) \* 2^e

// = buffer \* 10^-m + 10^-m \* (p2 ) \* 2^e

// = buffer \* 10^-m + 10^-m \* (1/10 \* (10 \* p2) ) \* 2^e

// = buffer \* 10^-m + 10^-m \* (1/10 \* ((10\*p2 div 2^-e) \* 2^-e + (10\*p2 mod 2^-e)) \* 2^e

//

assert(p2 <= (std::numeric\_limits<std::uint64\_t>::max)() / 10);

p2 \*= 10;

const std::uint64\_t d = p2 >> -one.e; // d = (10 \* p2) div 2^-e

const std::uint64\_t r = p2 & (one.f - 1); // r = (10 \* p2) mod 2^-e

//

// M+ = buffer \* 10^-m + 10^-m \* (1/10 \* (d \* 2^-e + r) \* 2^e

// = buffer \* 10^-m + 10^-m \* (1/10 \* (d + r \* 2^e))

// = (buffer \* 10 + d) \* 10^(-m-1) + 10^(-m-1) \* r \* 2^e

//

assert(d <= 9);

buffer[length++] = static\_cast<char>('0' + d); // buffer := buffer \* 10 + d

//

// M+ = buffer \* 10^(-m-1) + 10^(-m-1) \* r \* 2^e

//

p2 = r;

m++;

//

// M+ = buffer \* 10^-m + 10^-m \* p2 \* 2^e

// Invariant restored.

// Check if enough digits have been generated.

//

// 10^-m \* p2 \* 2^e <= delta \* 2^e

// p2 \* 2^e <= 10^m \* delta \* 2^e

// p2 <= 10^m \* delta

delta \*= 10;

dist \*= 10;

if (p2 <= delta)

{

break;

}

}

// V = buffer \* 10^-m, with M- <= V <= M+.

decimal\_exponent -= m;

// 1 ulp in the decimal representation is now 10^-m.

// Since delta and dist are now scaled by 10^m, we need to do the

// same with ulp in order to keep the units in sync.

//

// 10^m \* 10^-m = 1 = 2^-e \* 2^e = ten\_m \* 2^e

//

const std::uint64\_t ten\_m = one.f;

grisu2\_round(buffer, length, dist, delta, p2, ten\_m);

// By construction this algorithm generates the shortest possible decimal

// number (Loitsch, Theorem 6.2) which rounds back to w.

// For an input number of precision p, at least

//

// N = 1 + ceil(p \* log\_10(2))

//

// decimal digits are sufficient to identify all binary floating-point

// numbers (Matula, "In-and-Out conversions").

// This implies that the algorithm does not produce more than N decimal

// digits.

//

// N = 17 for p = 53 (IEEE double precision)

// N = 9 for p = 24 (IEEE single precision)

}

/\*!

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.

\*/

JSON\_HEDLEY\_NON\_NULL(1)

inline void grisu2(char\* buf, int& len, int& decimal\_exponent,

diyfp m\_minus, diyfp v, diyfp m\_plus)

{

assert(m\_plus.e == m\_minus.e);

assert(m\_plus.e == v.e);

// --------(-----------------------+-----------------------)-------- (A)

// m- v m+

//

// --------------------(-----------+-----------------------)-------- (B)

// m- v m+

//

// First scale v (and m- and m+) such that the exponent is in the range

// [alpha, gamma].

const cached\_power cached = get\_cached\_power\_for\_binary\_exponent(m\_plus.e);

const diyfp c\_minus\_k(cached.f, cached.e); // = c ~= 10^-k

// The exponent of the products is = v.e + c\_minus\_k.e + q and is in the range [alpha,gamma]

const diyfp w = diyfp::mul(v, c\_minus\_k);

const diyfp w\_minus = diyfp::mul(m\_minus, c\_minus\_k);

const diyfp w\_plus = diyfp::mul(m\_plus, c\_minus\_k);

// ----(---+---)---------------(---+---)---------------(---+---)----

// w- w w+

// = c\*m- = c\*v = c\*m+

//

// diyfp::mul rounds its result and c\_minus\_k is approximated too. w, w- and

// w+ are now off by a small amount.

// In fact:

//

// w - v \* 10^k < 1 ulp

//

// To account for this inaccuracy, add resp. subtract 1 ulp.

//

// --------+---[---------------(---+---)---------------]---+--------

// w- M- w M+ w+

//

// Now any number in [M-, M+] (bounds included) will round to w when input,

// regardless of how the input rounding algorithm breaks ties.

//

// And digit\_gen generates the shortest possible such number in [M-, M+].

// Note that this does not mean that Grisu2 always generates the shortest

// possible number in the interval (m-, m+).

const diyfp M\_minus(w\_minus.f + 1, w\_minus.e);

const diyfp M\_plus (w\_plus.f - 1, w\_plus.e );

decimal\_exponent = -cached.k; // = -(-k) = k

grisu2\_digit\_gen(buf, len, decimal\_exponent, M\_minus, w, M\_plus);

}

/\*!

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.

\*/

template <typename FloatType>

JSON\_HEDLEY\_NON\_NULL(1)

void grisu2(char\* buf, int& len, int& decimal\_exponent, FloatType value)

{

static\_assert(diyfp::kPrecision >= std::numeric\_limits<FloatType>::digits + 3,

"internal error: not enough precision");

assert(std::isfinite(value));

assert(value > 0);

// If the neighbors (and boundaries) of 'value' are always computed for double-precision

// numbers, all float's can be recovered using strtod (and strtof). However, the resulting

// decimal representations are not exactly "short".

//

// The documentation for 'std::to\_chars' (https://en.cppreference.com/w/cpp/utility/to\_chars)

// says "value is converted to a string as if by std::sprintf in the default ("C") locale"

// and since sprintf promotes float's to double's, I think this is exactly what 'std::to\_chars'

// does.

// On the other hand, the documentation for 'std::to\_chars' requires that "parsing the

// representation using the corresponding std::from\_chars function recovers value exactly". That

// indicates that single precision floating-point numbers should be recovered using

// 'std::strtof'.

//

// NB: If the neighbors are computed for single-precision numbers, there is a single float

// (7.0385307e-26f) which can't be recovered using strtod. The resulting double precision

// value is off by 1 ulp.

#if 0

const boundaries w = compute\_boundaries(static\_cast<double>(value));

#else

const boundaries w = compute\_boundaries(value);

#endif

grisu2(buf, len, decimal\_exponent, w.minus, w.w, w.plus);

}

/\*!

@brief appends a decimal representation of e to buf

@return a pointer to the element following the exponent.

@pre -1000 < e < 1000

\*/

JSON\_HEDLEY\_NON\_NULL(1)

JSON\_HEDLEY\_RETURNS\_NON\_NULL

inline char\* append\_exponent(char\* buf, int e)

{

assert(e > -1000);

assert(e < 1000);

if (e < 0)

{

e = -e;

\*buf++ = '-';

}

else

{

\*buf++ = '+';

}

auto k = static\_cast<std::uint32\_t>(e);

if (k < 10)

{

// Always print at least two digits in the exponent.

// This is for compatibility with printf("%g").

\*buf++ = '0';

\*buf++ = static\_cast<char>('0' + k);

}

else if (k < 100)

{

\*buf++ = static\_cast<char>('0' + k / 10);

k %= 10;

\*buf++ = static\_cast<char>('0' + k);

}

else

{

\*buf++ = static\_cast<char>('0' + k / 100);

k %= 100;

\*buf++ = static\_cast<char>('0' + k / 10);

k %= 10;

\*buf++ = static\_cast<char>('0' + k);

}

return buf;

}

/\*!

@brief 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.

@pre min\_exp < 0

@pre max\_exp > 0

\*/

JSON\_HEDLEY\_NON\_NULL(1)

JSON\_HEDLEY\_RETURNS\_NON\_NULL

inline char\* format\_buffer(char\* buf, int len, int decimal\_exponent,

int min\_exp, int max\_exp)

{

assert(min\_exp < 0);

assert(max\_exp > 0);

const int k = len;

const int n = len + decimal\_exponent;

// v = buf \* 10^(n-k)

// k is the length of the buffer (number of decimal digits)

// n is the position of the decimal point relative to the start of the buffer.

if (k <= n and n <= max\_exp)

{

// digits[000]

// len <= max\_exp + 2

std::memset(buf + k, '0', static\_cast<size\_t>(n - k));

// Make it look like a floating-point number (#362, #378)

buf[n + 0] = '.';

buf[n + 1] = '0';

return buf + (n + 2);

}

if (0 < n and n <= max\_exp)

{

// dig.its

// len <= max\_digits10 + 1

assert(k > n);

std::memmove(buf + (n + 1), buf + n, static\_cast<size\_t>(k - n));

buf[n] = '.';

return buf + (k + 1);

}

if (min\_exp < n and n <= 0)

{

// 0.[000]digits

// len <= 2 + (-min\_exp - 1) + max\_digits10

std::memmove(buf + (2 + -n), buf, static\_cast<size\_t>(k));

buf[0] = '0';

buf[1] = '.';

std::memset(buf + 2, '0', static\_cast<size\_t>(-n));

return buf + (2 + (-n) + k);

}

if (k == 1)

{

// dE+123

// len <= 1 + 5

buf += 1;

}

else

{

// d.igitsE+123

// len <= max\_digits10 + 1 + 5

std::memmove(buf + 2, buf + 1, static\_cast<size\_t>(k - 1));

buf[1] = '.';

buf += 1 + k;

}

\*buf++ = 'e';

return append\_exponent(buf, n - 1);

}

} // namespace dtoa\_impl

/\*!

@brief 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.

@note The buffer must be large enough.

@note The result is NOT null-terminated.

\*/

template <typename FloatType>

JSON\_HEDLEY\_NON\_NULL(1, 2)

JSON\_HEDLEY\_RETURNS\_NON\_NULL

char\* to\_chars(char\* first, const char\* last, FloatType value)

{

static\_cast<void>(last); // maybe unused - fix warning

assert(std::isfinite(value));

// Use signbit(value) instead of (value < 0) since signbit works for -0.

if (std::signbit(value))

{

value = -value;

\*first++ = '-';

}

if (value == 0) // +-0

{

\*first++ = '0';

// Make it look like a floating-point number (#362, #378)

\*first++ = '.';

\*first++ = '0';

return first;

}

assert(last - first >= std::numeric\_limits<FloatType>::max\_digits10);

// Compute v = buffer \* 10^decimal\_exponent.

// The decimal digits are stored in the buffer, which needs to be interpreted

// as an unsigned decimal integer.

// len is the length of the buffer, i.e. the number of decimal digits.

int len = 0;

int decimal\_exponent = 0;

dtoa\_impl::grisu2(first, len, decimal\_exponent, value);

assert(len <= std::numeric\_limits<FloatType>::max\_digits10);

// Format the buffer like printf("%.\*g", prec, value)

constexpr int kMinExp = -4;

// Use digits10 here to increase compatibility with version 2.

constexpr int kMaxExp = std::numeric\_limits<FloatType>::digits10;

assert(last - first >= kMaxExp + 2);

assert(last - first >= 2 + (-kMinExp - 1) + std::numeric\_limits<FloatType>::max\_digits10);

assert(last - first >= std::numeric\_limits<FloatType>::max\_digits10 + 6);

return dtoa\_impl::format\_buffer(first, len, decimal\_exponent, kMinExp, kMaxExp);

}

} // namespace detail

} // namespace nlohmann

// #include <nlohmann/detail/exceptions.hpp>

// #include <nlohmann/detail/macro\_scope.hpp>

// #include <nlohmann/detail/meta/cpp\_future.hpp>

// #include <nlohmann/detail/output/binary\_writer.hpp>

// #include <nlohmann/detail/output/output\_adapters.hpp>

// #include <nlohmann/detail/value\_t.hpp>

namespace nlohmann

{

namespace detail

{

///////////////////

// serialization //

///////////////////

/// how to treat decoding errors

enum class error\_handler\_t

{

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

};

template<typename BasicJsonType>

class serializer

{

using string\_t = typename BasicJsonType::string\_t;

using number\_float\_t = typename BasicJsonType::number\_float\_t;

using number\_integer\_t = typename BasicJsonType::number\_integer\_t;

using number\_unsigned\_t = typename BasicJsonType::number\_unsigned\_t;

static constexpr std::uint8\_t UTF8\_ACCEPT = 0;

static constexpr std::uint8\_t UTF8\_REJECT = 1;

public:

/\*!

@param[in] s output stream to serialize to

@param[in] ichar indentation character to use

@param[in] error\_handler\_ how to react on decoding errors

\*/

serializer(output\_adapter\_t<char> s, const char ichar,

error\_handler\_t error\_handler\_ = error\_handler\_t::strict)

: o(std::move(s))

, loc(std::localeconv())

, thousands\_sep(loc->thousands\_sep == nullptr ? '\0' : \* (loc->thousands\_sep))

, decimal\_point(loc->decimal\_point == nullptr ? '\0' : \* (loc->decimal\_point))

, indent\_char(ichar)

, indent\_string(512, indent\_char)

, error\_handler(error\_handler\_)

{}

// delete because of pointer members

serializer(const serializer&) = delete;

serializer& operator=(const serializer&) = delete;

serializer(serializer&&) = delete;

serializer& operator=(serializer&&) = delete;

~serializer() = default;

/\*!

@brief 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

@param[in] val value to serialize

@param[in] pretty\_print whether the output shall be pretty-printed

@param[in] indent\_step the indent level

@param[in] current\_indent the current indent level (only used internally)

\*/

void dump(const BasicJsonType& val, const bool pretty\_print,

const bool ensure\_ascii,

const unsigned int indent\_step,

const unsigned int current\_indent = 0)

{

switch (val.m\_type)

{

case value\_t::object:

{

if (val.m\_value.object->empty())

{

o->write\_characters("{}", 2);

return;

}

if (pretty\_print)

{

o->write\_characters("{\n", 2);

// variable to hold indentation for recursive calls

const auto new\_indent = current\_indent + indent\_step;

if (JSON\_HEDLEY\_UNLIKELY(indent\_string.size() < new\_indent))

{

indent\_string.resize(indent\_string.size() \* 2, ' ');

}

// first n-1 elements

auto i = val.m\_value.object->cbegin();

for (std::size\_t cnt = 0; cnt < val.m\_value.object->size() - 1; ++cnt, ++i)

{

o->write\_characters(indent\_string.c\_str(), new\_indent);

o->write\_character('\"');

dump\_escaped(i->first, ensure\_ascii);

o->write\_characters("\": ", 3);

dump(i->second, true, ensure\_ascii, indent\_step, new\_indent);

o->write\_characters(",\n", 2);

}

// last element

assert(i != val.m\_value.object->cend());

assert(std::next(i) == val.m\_value.object->cend());

o->write\_characters(indent\_string.c\_str(), new\_indent);

o->write\_character('\"');

dump\_escaped(i->first, ensure\_ascii);

o->write\_characters("\": ", 3);

dump(i->second, true, ensure\_ascii, indent\_step, new\_indent);

o->write\_character('\n');

o->write\_characters(indent\_string.c\_str(), current\_indent);

o->write\_character('}');

}

else

{

o->write\_character('{');

// first n-1 elements

auto i = val.m\_value.object->cbegin();

for (std::size\_t cnt = 0; cnt < val.m\_value.object->size() - 1; ++cnt, ++i)

{

o->write\_character('\"');

dump\_escaped(i->first, ensure\_ascii);

o->write\_characters("\":", 2);

dump(i->second, false, ensure\_ascii, indent\_step, current\_indent);

o->write\_character(',');

}

// last element

assert(i != val.m\_value.object->cend());

assert(std::next(i) == val.m\_value.object->cend());

o->write\_character('\"');

dump\_escaped(i->first, ensure\_ascii);

o->write\_characters("\":", 2);

dump(i->second, false, ensure\_ascii, indent\_step, current\_indent);

o->write\_character('}');

}

return;

}

case value\_t::array:

{

if (val.m\_value.array->empty())

{

o->write\_characters("[]", 2);

return;

}

if (pretty\_print)

{

o->write\_characters("[\n", 2);

// variable to hold indentation for recursive calls

const auto new\_indent = current\_indent + indent\_step;

if (JSON\_HEDLEY\_UNLIKELY(indent\_string.size() < new\_indent))

{

indent\_string.resize(indent\_string.size() \* 2, ' ');

}

// first n-1 elements

for (auto i = val.m\_value.array->cbegin();

i != val.m\_value.array->cend() - 1; ++i)

{

o->write\_characters(indent\_string.c\_str(), new\_indent);

dump(\*i, true, ensure\_ascii, indent\_step, new\_indent);

o->write\_characters(",\n", 2);

}

// last element

assert(not val.m\_value.array->empty());

o->write\_characters(indent\_string.c\_str(), new\_indent);

dump(val.m\_value.array->back(), true, ensure\_ascii, indent\_step, new\_indent);

o->write\_character('\n');

o->write\_characters(indent\_string.c\_str(), current\_indent);

o->write\_character(']');

}

else

{

o->write\_character('[');

// first n-1 elements

for (auto i = val.m\_value.array->cbegin();

i != val.m\_value.array->cend() - 1; ++i)

{

dump(\*i, false, ensure\_ascii, indent\_step, current\_indent);

o->write\_character(',');

}

// last element

assert(not val.m\_value.array->empty());

dump(val.m\_value.array->back(), false, ensure\_ascii, indent\_step, current\_indent);

o->write\_character(']');

}

return;

}

case value\_t::string:

{

o->write\_character('\"');

dump\_escaped(\*val.m\_value.string, ensure\_ascii);

o->write\_character('\"');

return;

}

case value\_t::boolean:

{

if (val.m\_value.boolean)

{

o->write\_characters("true", 4);

}

else

{

o->write\_characters("false", 5);

}

return;

}

case value\_t::number\_integer:

{

dump\_integer(val.m\_value.number\_integer);

return;

}

case value\_t::number\_unsigned:

{

dump\_integer(val.m\_value.number\_unsigned);

return;

}

case value\_t::number\_float:

{

dump\_float(val.m\_value.number\_float);

return;

}

case value\_t::discarded:

{

o->write\_characters("<discarded>", 11);

return;

}

case value\_t::null:

{

o->write\_characters("null", 4);

return;

}

default: // LCOV\_EXCL\_LINE

assert(false); // LCOV\_EXCL\_LINE

}

}

private:

/\*!

@brief dump escaped string

Escape a string by replacing certain special characters by a sequence of an

escape character (backslash) and another character and other control

characters by a sequence of "\u" followed by a four-digit hex

representation. The escaped string is written to output stream @a o.

@param[in] s the string to escape

@param[in] ensure\_ascii whether to escape non-ASCII characters with

\uXXXX sequences

@complexity Linear in the length of string @a s.

\*/

void dump\_escaped(const string\_t& s, const bool ensure\_ascii)

{

std::uint32\_t codepoint;

std::uint8\_t state = UTF8\_ACCEPT;

std::size\_t bytes = 0; // number of bytes written to string\_buffer

// number of bytes written at the point of the last valid byte

std::size\_t bytes\_after\_last\_accept = 0;

std::size\_t undumped\_chars = 0;

for (std::size\_t i = 0; i < s.size(); ++i)

{

const auto byte = static\_cast<uint8\_t>(s[i]);

switch (decode(state, codepoint, byte))

{

case UTF8\_ACCEPT: // decode found a new code point

{

switch (codepoint)

{

case 0x08: // backspace

{

string\_buffer[bytes++] = '\\';

string\_buffer[bytes++] = 'b';

break;

}

case 0x09: // horizontal tab

{

string\_buffer[bytes++] = '\\';

string\_buffer[bytes++] = 't';

break;

}

case 0x0A: // newline

{

string\_buffer[bytes++] = '\\';

string\_buffer[bytes++] = 'n';

break;

}

case 0x0C: // formfeed

{

string\_buffer[bytes++] = '\\';

string\_buffer[bytes++] = 'f';

break;

}

case 0x0D: // carriage return

{

string\_buffer[bytes++] = '\\';

string\_buffer[bytes++] = 'r';

break;

}

case 0x22: // quotation mark

{

string\_buffer[bytes++] = '\\';

string\_buffer[bytes++] = '\"';

break;

}

case 0x5C: // reverse solidus

{

string\_buffer[bytes++] = '\\';

string\_buffer[bytes++] = '\\';

break;

}

default:

{

// escape control characters (0x00..0x1F) or, if

// ensure\_ascii parameter is used, non-ASCII characters

if ((codepoint <= 0x1F) or (ensure\_ascii and (codepoint >= 0x7F)))

{

if (codepoint <= 0xFFFF)

{

(std::snprintf)(string\_buffer.data() + bytes, 7, "\\u%04x",

static\_cast<std::uint16\_t>(codepoint));

bytes += 6;

}

else

{

(std::snprintf)(string\_buffer.data() + bytes, 13, "\\u%04x\\u%04x",

static\_cast<std::uint16\_t>(0xD7C0u + (codepoint >> 10u)),

static\_cast<std::uint16\_t>(0xDC00u + (codepoint & 0x3FFu)));

bytes += 12;

}

}

else

{

// copy byte to buffer (all previous bytes

// been copied have in default case above)

string\_buffer[bytes++] = s[i];

}

break;

}

}

// write buffer and reset index; there must be 13 bytes

// left, as this is the maximal number of bytes to be

// written ("\uxxxx\uxxxx\0") for one code point

if (string\_buffer.size() - bytes < 13)

{

o->write\_characters(string\_buffer.data(), bytes);

bytes = 0;

}

// remember the byte position of this accept

bytes\_after\_last\_accept = bytes;

undumped\_chars = 0;

break;

}

case UTF8\_REJECT: // decode found invalid UTF-8 byte

{

switch (error\_handler)

{

case error\_handler\_t::strict:

{

std::string sn(3, '\0');

(std::snprintf)(&sn[0], sn.size(), "%.2X", byte);

JSON\_THROW(type\_error::create(316, "invalid UTF-8 byte at index " + std::to\_string(i) + ": 0x" + sn));

}

case error\_handler\_t::ignore:

case error\_handler\_t::replace:

{

// in case we saw this character the first time, we

// would like to read it again, because the byte

// may be OK for itself, but just not OK for the

// previous sequence

if (undumped\_chars > 0)

{

--i;

}

// reset length buffer to the last accepted index;

// thus removing/ignoring the invalid characters

bytes = bytes\_after\_last\_accept;

if (error\_handler == error\_handler\_t::replace)

{

// add a replacement character

if (ensure\_ascii)

{

string\_buffer[bytes++] = '\\';

string\_buffer[bytes++] = 'u';

string\_buffer[bytes++] = 'f';

string\_buffer[bytes++] = 'f';

string\_buffer[bytes++] = 'f';

string\_buffer[bytes++] = 'd';

}

else

{

string\_buffer[bytes++] = detail::binary\_writer<BasicJsonType, char>::to\_char\_type('\xEF');

string\_buffer[bytes++] = detail::binary\_writer<BasicJsonType, char>::to\_char\_type('\xBF');

string\_buffer[bytes++] = detail::binary\_writer<BasicJsonType, char>::to\_char\_type('\xBD');

}

// write buffer and reset index; there must be 13 bytes

// left, as this is the maximal number of bytes to be

// written ("\uxxxx\uxxxx\0") for one code point

if (string\_buffer.size() - bytes < 13)

{

o->write\_characters(string\_buffer.data(), bytes);

bytes = 0;

}

bytes\_after\_last\_accept = bytes;

}

undumped\_chars = 0;

// continue processing the string

state = UTF8\_ACCEPT;

break;

}

default: // LCOV\_EXCL\_LINE

assert(false); // LCOV\_EXCL\_LINE

}

break;

}

default: // decode found yet incomplete multi-byte code point

{

if (not ensure\_ascii)

{

// code point will not be escaped - copy byte to buffer

string\_buffer[bytes++] = s[i];

}

++undumped\_chars;

break;

}

}

}

// we finished processing the string

if (JSON\_HEDLEY\_LIKELY(state == UTF8\_ACCEPT))

{

// write buffer

if (bytes > 0)

{

o->write\_characters(string\_buffer.data(), bytes);

}

}

else

{

// we finish reading, but do not accept: string was incomplete

switch (error\_handler)

{

case error\_handler\_t::strict:

{

std::string sn(3, '\0');

(std::snprintf)(&sn[0], sn.size(), "%.2X", static\_cast<std::uint8\_t>(s.back()));

JSON\_THROW(type\_error::create(316, "incomplete UTF-8 string; last byte: 0x" + sn));

}

case error\_handler\_t::ignore:

{

// write all accepted bytes

o->write\_characters(string\_buffer.data(), bytes\_after\_last\_accept);

break;

}

case error\_handler\_t::replace:

{

// write all accepted bytes

o->write\_characters(string\_buffer.data(), bytes\_after\_last\_accept);

// add a replacement character

if (ensure\_ascii)

{

o->write\_characters("\\ufffd", 6);

}

else

{

o->write\_characters("\xEF\xBF\xBD", 3);

}

break;

}

default: // LCOV\_EXCL\_LINE

assert(false); // LCOV\_EXCL\_LINE

}

}

}

/\*!

@brief count digits

Count the number of decimal (base 10) digits for an input unsigned integer.

@param[in] x unsigned integer number to count its digits

@return number of decimal digits

\*/

inline unsigned int count\_digits(number\_unsigned\_t x) noexcept

{

unsigned int n\_digits = 1;

for (;;)

{

if (x < 10)

{

return n\_digits;

}

if (x < 100)

{

return n\_digits + 1;

}

if (x < 1000)

{

return n\_digits + 2;

}

if (x < 10000)

{

return n\_digits + 3;

}

x = x / 10000u;

n\_digits += 4;

}

}

/\*!

@brief dump an integer

Dump a given integer to output stream @a o. Works internally with

@a number\_buffer.

@param[in] x integer number (signed or unsigned) to dump

@tparam NumberType either @a number\_integer\_t or @a number\_unsigned\_t

\*/

template<typename NumberType, detail::enable\_if\_t<

std::is\_same<NumberType, number\_unsigned\_t>::value or

std::is\_same<NumberType, number\_integer\_t>::value,

int> = 0>

void dump\_integer(NumberType x)

{

static constexpr std::array<std::array<char, 2>, 100> digits\_to\_99

{

{

{{'0', '0'}}, {{'0', '1'}}, {{'0', '2'}}, {{'0', '3'}}, {{'0', '4'}}, {{'0', '5'}}, {{'0', '6'}}, {{'0', '7'}}, {{'0', '8'}}, {{'0', '9'}},

{{'1', '0'}}, {{'1', '1'}}, {{'1', '2'}}, {{'1', '3'}}, {{'1', '4'}}, {{'1', '5'}}, {{'1', '6'}}, {{'1', '7'}}, {{'1', '8'}}, {{'1', '9'}},

{{'2', '0'}}, {{'2', '1'}}, {{'2', '2'}}, {{'2', '3'}}, {{'2', '4'}}, {{'2', '5'}}, {{'2', '6'}}, {{'2', '7'}}, {{'2', '8'}}, {{'2', '9'}},

{{'3', '0'}}, {{'3', '1'}}, {{'3', '2'}}, {{'3', '3'}}, {{'3', '4'}}, {{'3', '5'}}, {{'3', '6'}}, {{'3', '7'}}, {{'3', '8'}}, {{'3', '9'}},

{{'4', '0'}}, {{'4', '1'}}, {{'4', '2'}}, {{'4', '3'}}, {{'4', '4'}}, {{'4', '5'}}, {{'4', '6'}}, {{'4', '7'}}, {{'4', '8'}}, {{'4', '9'}},

{{'5', '0'}}, {{'5', '1'}}, {{'5', '2'}}, {{'5', '3'}}, {{'5', '4'}}, {{'5', '5'}}, {{'5', '6'}}, {{'5', '7'}}, {{'5', '8'}}, {{'5', '9'}},

{{'6', '0'}}, {{'6', '1'}}, {{'6', '2'}}, {{'6', '3'}}, {{'6', '4'}}, {{'6', '5'}}, {{'6', '6'}}, {{'6', '7'}}, {{'6', '8'}}, {{'6', '9'}},

{{'7', '0'}}, {{'7', '1'}}, {{'7', '2'}}, {{'7', '3'}}, {{'7', '4'}}, {{'7', '5'}}, {{'7', '6'}}, {{'7', '7'}}, {{'7', '8'}}, {{'7', '9'}},

{{'8', '0'}}, {{'8', '1'}}, {{'8', '2'}}, {{'8', '3'}}, {{'8', '4'}}, {{'8', '5'}}, {{'8', '6'}}, {{'8', '7'}}, {{'8', '8'}}, {{'8', '9'}},

{{'9', '0'}}, {{'9', '1'}}, {{'9', '2'}}, {{'9', '3'}}, {{'9', '4'}}, {{'9', '5'}}, {{'9', '6'}}, {{'9', '7'}}, {{'9', '8'}}, {{'9', '9'}},

}

};

// special case for "0"

if (x == 0)

{

o->write\_character('0');

return;

}

// use a pointer to fill the buffer

auto buffer\_ptr = number\_buffer.begin();

const bool is\_negative = std::is\_same<NumberType, number\_integer\_t>::value and not(x >= 0); // see issue #755

number\_unsigned\_t abs\_value;

unsigned int n\_chars;

if (is\_negative)

{

\*buffer\_ptr = '-';

abs\_value = remove\_sign(x);

// account one more byte for the minus sign

n\_chars = 1 + count\_digits(abs\_value);

}

else

{

abs\_value = static\_cast<number\_unsigned\_t>(x);

n\_chars = count\_digits(abs\_value);

}

// spare 1 byte for '\0'

assert(n\_chars < number\_buffer.size() - 1);

// jump to the end to generate the string from backward

// so we later avoid reversing the result

buffer\_ptr += n\_chars;

// Fast int2ascii implementation inspired by "Fastware" talk by Andrei Alexandrescu

// See: https://www.youtube.com/watch?v=o4-CwDo2zpg

while (abs\_value >= 100)

{

const auto digits\_index = static\_cast<unsigned>((abs\_value % 100));

abs\_value /= 100;

\*(--buffer\_ptr) = digits\_to\_99[digits\_index][1];

\*(--buffer\_ptr) = digits\_to\_99[digits\_index][0];

}

if (abs\_value >= 10)

{

const auto digits\_index = static\_cast<unsigned>(abs\_value);

\*(--buffer\_ptr) = digits\_to\_99[digits\_index][1];

\*(--buffer\_ptr) = digits\_to\_99[digits\_index][0];

}

else

{

\*(--buffer\_ptr) = static\_cast<char>('0' + abs\_value);

}

o->write\_characters(number\_buffer.data(), n\_chars);

}

/\*!

@brief dump a floating-point number

Dump a given floating-point number to output stream @a o. Works internally

with @a number\_buffer.

@param[in] x floating-point number to dump

\*/

void dump\_float(number\_float\_t x)

{

// NaN / inf

if (not std::isfinite(x))

{

o->write\_characters("null", 4);

return;

}

// If number\_float\_t is an IEEE-754 single or double precision number,

// use the Grisu2 algorithm to produce short numbers which are

// guaranteed to round-trip, using strtof and strtod, resp.

//

// NB: The test below works if <long double> == <double>.

static constexpr bool is\_ieee\_single\_or\_double

= (std::numeric\_limits<number\_float\_t>::is\_iec559 and std::numeric\_limits<number\_float\_t>::digits == 24 and std::numeric\_limits<number\_float\_t>::max\_exponent == 128) or

(std::numeric\_limits<number\_float\_t>::is\_iec559 and std::numeric\_limits<number\_float\_t>::digits == 53 and std::numeric\_limits<number\_float\_t>::max\_exponent == 1024);

dump\_float(x, std::integral\_constant<bool, is\_ieee\_single\_or\_double>());

}

void dump\_float(number\_float\_t x, std::true\_type /\*is\_ieee\_single\_or\_double\*/)

{

char\* begin = number\_buffer.data();

char\* end = ::nlohmann::detail::to\_chars(begin, begin + number\_buffer.size(), x);

o->write\_characters(begin, static\_cast<size\_t>(end - begin));

}

void dump\_float(number\_float\_t x, std::false\_type /\*is\_ieee\_single\_or\_double\*/)

{

// get number of digits for a float -> text -> float round-trip

static constexpr auto d = std::numeric\_limits<number\_float\_t>::max\_digits10;

// the actual conversion

std::ptrdiff\_t len = (std::snprintf)(number\_buffer.data(), number\_buffer.size(), "%.\*g", d, x);

// negative value indicates an error

assert(len > 0);

// check if buffer was large enough

assert(static\_cast<std::size\_t>(len) < number\_buffer.size());

// erase thousands separator

if (thousands\_sep != '\0')

{

const auto end = std::remove(number\_buffer.begin(),

number\_buffer.begin() + len, thousands\_sep);

std::fill(end, number\_buffer.end(), '\0');

assert((end - number\_buffer.begin()) <= len);

len = (end - number\_buffer.begin());

}

// convert decimal point to '.'

if (decimal\_point != '\0' and decimal\_point != '.')

{

const auto dec\_pos = std::find(number\_buffer.begin(), number\_buffer.end(), decimal\_point);

if (dec\_pos != number\_buffer.end())

{

\*dec\_pos = '.';

}

}

o->write\_characters(number\_buffer.data(), static\_cast<std::size\_t>(len));

// determine if need to append ".0"

const bool value\_is\_int\_like =

std::none\_of(number\_buffer.begin(), number\_buffer.begin() + len + 1,

[](char c)

{

return c == '.' or c == 'e';

});

if (value\_is\_int\_like)

{

o->write\_characters(".0", 2);

}

}

/\*!

@brief check whether a string is UTF-8 encoded

The function checks each byte of a string whether it is UTF-8 encoded. The

result of the check is stored in the @a state parameter. The function must

be called initially with state 0 (accept). State 1 means the string must

be rejected, because the current byte is not allowed. If the string is

completely processed, but the state is non-zero, the string ended

prematurely; that is, the last byte indicated more bytes should have

followed.

@param[in,out] state the state of the decoding

@param[in,out] codep codepoint (valid only if resulting state is UTF8\_ACCEPT)

@param[in] byte next byte to decode

@return new state

@note The function has been edited: a std::array is used.

@copyright Copyright (c) 2008-2009 Bjoern Hoehrmann <bjoern@hoehrmann.de>

@sa http://bjoern.hoehrmann.de/utf-8/decoder/dfa/

\*/

static std::uint8\_t decode(std::uint8\_t& state, std::uint32\_t& codep, const std::uint8\_t byte) noexcept

{

static const std::array<std::uint8\_t, 400> utf8d =

{

{

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

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

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

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

1, 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, // 80..9F

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

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, // C0..DF

0xA, 0x3, 0x3, 0x3, 0x3, 0x3, 0x3, 0x3, 0x3, 0x3, 0x3, 0x3, 0x3, 0x4, 0x3, 0x3, // E0..EF

0xB, 0x6, 0x6, 0x6, 0x5, 0x8, 0x8, 0x8, 0x8, 0x8, 0x8, 0x8, 0x8, 0x8, 0x8, 0x8, // F0..FF

0x0, 0x1, 0x2, 0x3, 0x5, 0x8, 0x7, 0x1, 0x1, 0x1, 0x4, 0x6, 0x1, 0x1, 0x1, 0x1, // s0..s0

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

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

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

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

}

};

const std::uint8\_t type = utf8d[byte];

codep = (state != UTF8\_ACCEPT)

? (byte & 0x3fu) | (codep << 6u)

: (0xFFu >> type) & (byte);

state = utf8d[256u + state \* 16u + type];

return state;

}

/\*

\* Overload to make the compiler happy while it is instantiating

\* dump\_integer for number\_unsigned\_t.

\* Must never be called.

\*/

number\_unsigned\_t remove\_sign(number\_unsigned\_t x)

{

assert(false); // LCOV\_EXCL\_LINE

return x; // LCOV\_EXCL\_LINE

}

/\*

\* Helper function for dump\_integer

\*

\* This function takes a negative signed integer and returns its absolute

\* value as unsigned integer. The plus/minus shuffling is necessary as we can

\* not directly remove the sign of an arbitrary signed integer as the

\* absolute values of INT\_MIN and INT\_MAX are usually not the same. See

\* #1708 for details.

\*/

inline number\_unsigned\_t remove\_sign(number\_integer\_t x) noexcept

{

assert(x < 0 and x < (std::numeric\_limits<number\_integer\_t>::max)());

return static\_cast<number\_unsigned\_t>(-(x + 1)) + 1;

}

private:

/// the output of the serializer

output\_adapter\_t<char> o = nullptr;

/// a (hopefully) large enough character buffer

std::array<char, 64> number\_buffer{{}};

/// 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;

};

} // namespace detail

} // namespace nlohmann

// #include <nlohmann/detail/value\_t.hpp>

// #include <nlohmann/json\_fwd.hpp>

/\*!

@brief namespace for Niels Lohmann

@see https://github.com/nlohmann

@since version 1.0.0

\*/

namespace nlohmann

{

/\*!

@brief a class to store JSON values

@tparam ObjectType type for JSON objects (`std::map` by default; will be used

in @ref object\_t)

@tparam ArrayType type for JSON arrays (`std::vector` by default; will be used

in @ref array\_t)

@tparam StringType type for JSON strings and object keys (`std::string` by

default; will be used in @ref string\_t)

@tparam BooleanType type for JSON booleans (`bool` by default; will be used

in @ref boolean\_t)

@tparam NumberIntegerType type for JSON integer numbers (`int64\_t` by

default; will be used in @ref number\_integer\_t)

@tparam NumberUnsignedType type for JSON unsigned integer numbers (@c

`uint64\_t` by default; will be used in @ref number\_unsigned\_t)

@tparam NumberFloatType type for JSON floating-point numbers (`double` by

default; will be used in @ref number\_float\_t)

@tparam AllocatorType type of the allocator to use (`std::allocator` by

default)

@tparam JSONSerializer the serializer to resolve internal calls to `to\_json()`

and `from\_json()` (@ref adl\_serializer by default)

@requirement The class satisfies the following concept requirements:

- Basic

- [DefaultConstructible](https://en.cppreference.com/w/cpp/named\_req/DefaultConstructible):

JSON values can be default constructed. The result will be a JSON null

value.

- [MoveConstructible](https://en.cppreference.com/w/cpp/named\_req/MoveConstructible):

A JSON value can be constructed from an rvalue argument.

- [CopyConstructible](https://en.cppreference.com/w/cpp/named\_req/CopyConstructible):

A JSON value can be copy-constructed from an lvalue expression.

- [MoveAssignable](https://en.cppreference.com/w/cpp/named\_req/MoveAssignable):

A JSON value van be assigned from an rvalue argument.

- [CopyAssignable](https://en.cppreference.com/w/cpp/named\_req/CopyAssignable):

A JSON value can be copy-assigned from an lvalue expression.

- [Destructible](https://en.cppreference.com/w/cpp/named\_req/Destructible):

JSON values can be destructed.

- Layout

- [StandardLayoutType](https://en.cppreference.com/w/cpp/named\_req/StandardLayoutType):

JSON values have

[standard layout](https://en.cppreference.com/w/cpp/language/data\_members#Standard\_layout):

All non-static data members are private and standard layout types, the

class has no virtual functions or (virtual) base classes.

- Library-wide

- [EqualityComparable](https://en.cppreference.com/w/cpp/named\_req/EqualityComparable):

JSON values can be compared with `==`, see @ref

operator==(const\_reference,const\_reference).

- [LessThanComparable](https://en.cppreference.com/w/cpp/named\_req/LessThanComparable):

JSON values can be compared with `<`, see @ref

operator<(const\_reference,const\_reference).

- [Swappable](https://en.cppreference.com/w/cpp/named\_req/Swappable):

Any JSON lvalue or rvalue of can be swapped with any lvalue or rvalue of

other compatible types, using unqualified function call @ref swap().

- [NullablePointer](https://en.cppreference.com/w/cpp/named\_req/NullablePointer):

JSON values can be compared against `std::nullptr\_t` objects which are used

to model the `null` value.

- Container

- [Container](https://en.cppreference.com/w/cpp/named\_req/Container):

JSON values can be used like STL containers and provide iterator access.

- [ReversibleContainer](https://en.cppreference.com/w/cpp/named\_req/ReversibleContainer);

JSON values can be used like STL containers and provide reverse iterator

access.

@invariant The member variables @a m\_value and @a m\_type have the following

relationship:

- If `m\_type == value\_t::object`, then `m\_value.object != nullptr`.

- If `m\_type == value\_t::array`, then `m\_value.array != nullptr`.

- If `m\_type == value\_t::string`, then `m\_value.string != nullptr`.

The invariants are checked by member function assert\_invariant().

@internal

@note ObjectType trick from http://stackoverflow.com/a/9860911

@endinternal

@see [RFC 7159: The JavaScript Object Notation (JSON) Data Interchange

Format](http://rfc7159.net/rfc7159)

@since version 1.0.0

@nosubgrouping

\*/

NLOHMANN\_BASIC\_JSON\_TPL\_DECLARATION

class basic\_json

{

private:

template<detail::value\_t> friend struct detail::external\_constructor;

friend ::nlohmann::json\_pointer<basic\_json>;

friend ::nlohmann::detail::parser<basic\_json>;

friend ::nlohmann::detail::serializer<basic\_json>;

template<typename BasicJsonType>

friend class ::nlohmann::detail::iter\_impl;

template<typename BasicJsonType, typename CharType>

friend class ::nlohmann::detail::binary\_writer;

template<typename BasicJsonType, typename SAX>

friend class ::nlohmann::detail::binary\_reader;

template<typename BasicJsonType>

friend class ::nlohmann::detail::json\_sax\_dom\_parser;

template<typename BasicJsonType>

friend class ::nlohmann::detail::json\_sax\_dom\_callback\_parser;

/// workaround type for MSVC

using basic\_json\_t = NLOHMANN\_BASIC\_JSON\_TPL;

// convenience aliases for types residing in namespace detail;

using lexer = ::nlohmann::detail::lexer<basic\_json>;

using parser = ::nlohmann::detail::parser<basic\_json>;

using primitive\_iterator\_t = ::nlohmann::detail::primitive\_iterator\_t;

template<typename BasicJsonType>

using internal\_iterator = ::nlohmann::detail::internal\_iterator<BasicJsonType>;

template<typename BasicJsonType>

using iter\_impl = ::nlohmann::detail::iter\_impl<BasicJsonType>;

template<typename Iterator>

using iteration\_proxy = ::nlohmann::detail::iteration\_proxy<Iterator>;

template<typename Base> using json\_reverse\_iterator = ::nlohmann::detail::json\_reverse\_iterator<Base>;

template<typename CharType>

using output\_adapter\_t = ::nlohmann::detail::output\_adapter\_t<CharType>;

using binary\_reader = ::nlohmann::detail::binary\_reader<basic\_json>;

template<typename CharType> using binary\_writer = ::nlohmann::detail::binary\_writer<basic\_json, CharType>;

using serializer = ::nlohmann::detail::serializer<basic\_json>;

public:

using value\_t = detail::value\_t;

/// JSON Pointer, see @ref nlohmann::json\_pointer

using json\_pointer = ::nlohmann::json\_pointer<basic\_json>;

template<typename T, typename SFINAE>

using json\_serializer = JSONSerializer<T, SFINAE>;

/// how to treat decoding errors

using error\_handler\_t = detail::error\_handler\_t;

/// helper type for initializer lists of basic\_json values

using initializer\_list\_t = std::initializer\_list<detail::json\_ref<basic\_json>>;

using input\_format\_t = detail::input\_format\_t;

/// SAX interface type, see @ref nlohmann::json\_sax

using json\_sax\_t = json\_sax<basic\_json>;

////////////////

// exceptions //

////////////////

/// @name exceptions

/// Classes to implement user-defined exceptions.

/// @{

/// @copydoc detail::exception

using exception = detail::exception;

/// @copydoc detail::parse\_error

using parse\_error = detail::parse\_error;

/// @copydoc detail::invalid\_iterator

using invalid\_iterator = detail::invalid\_iterator;

/// @copydoc detail::type\_error

using type\_error = detail::type\_error;

/// @copydoc detail::out\_of\_range

using out\_of\_range = detail::out\_of\_range;

/// @copydoc detail::other\_error

using other\_error = detail::other\_error;

/// @}

/////////////////////

// container types //

/////////////////////

/// @name container types

/// The canonic container types to use @ref basic\_json like any other STL

/// container.

/// @{

/// the type of elements in a basic\_json container

using value\_type = basic\_json;

/// the type of an element reference

using reference = value\_type&;

/// the type of an element const reference

using const\_reference = const value\_type&;

/// a type to represent differences between iterators

using difference\_type = std::ptrdiff\_t;

/// a type to represent container sizes

using size\_type = std::size\_t;

/// the allocator type

using allocator\_type = AllocatorType<basic\_json>;

/// the type of an element pointer

using pointer = typename std::allocator\_traits<allocator\_type>::pointer;

/// the type of an element const pointer

using const\_pointer = typename std::allocator\_traits<allocator\_type>::const\_pointer;

/// an iterator for a basic\_json container

using iterator = iter\_impl<basic\_json>;

/// a const iterator for a basic\_json container

using const\_iterator = iter\_impl<const basic\_json>;

/// a reverse iterator for a basic\_json container

using reverse\_iterator = json\_reverse\_iterator<typename basic\_json::iterator>;

/// a const reverse iterator for a basic\_json container

using const\_reverse\_iterator = json\_reverse\_iterator<typename basic\_json::const\_iterator>;

/// @}

/\*!

@brief returns the allocator associated with the container

\*/

static allocator\_type get\_allocator()

{

return allocator\_type();

}

/\*!

@brief returns version information on the library

This function returns a JSON object with information about the library,

including the version number and information on the platform and compiler.

@return JSON object holding version information

key | description

----------- | ---------------

`compiler` | Information on the used compiler. It is an object with the following keys: `c++` (the used C++ standard), `family` (the compiler family; possible values are `clang`, `icc`, `gcc`, `ilecpp`, `msvc`, `pgcpp`, `sunpro`, and `unknown`), and `version` (the compiler version).

`copyright` | The copyright line for the library as string.

`name` | The name of the library as string.

`platform` | The used platform as string. Possible values are `win32`, `linux`, `apple`, `unix`, and `unknown`.

`url` | The URL of the project as string.

`version` | The version of the library. It is an object with the following keys: `major`, `minor`, and `patch` as defined by [Semantic Versioning](http://semver.org), and `string` (the version string).

@liveexample{The following code shows an example output of the `meta()`

function.,meta}

@exceptionsafety Strong guarantee: if an exception is thrown, there are no

changes to any JSON value.

@complexity Constant.

@since 2.1.0

\*/

JSON\_HEDLEY\_WARN\_UNUSED\_RESULT

static basic\_json meta()

{

basic\_json result;

result["copyright"] = "(C) 2013-2017 Niels Lohmann";

result["name"] = "JSON for Modern C++";

result["url"] = "https://github.com/nlohmann/json";

result["version"]["string"] =

std::to\_string(NLOHMANN\_JSON\_VERSION\_MAJOR) + "." +

std::to\_string(NLOHMANN\_JSON\_VERSION\_MINOR) + "." +

std::to\_string(NLOHMANN\_JSON\_VERSION\_PATCH);

result["version"]["major"] = NLOHMANN\_JSON\_VERSION\_MAJOR;

result["version"]["minor"] = NLOHMANN\_JSON\_VERSION\_MINOR;

result["version"]["patch"] = NLOHMANN\_JSON\_VERSION\_PATCH;

#ifdef \_WIN32

result["platform"] = "win32";

#elif defined \_\_linux\_\_

result["platform"] = "linux";

#elif defined \_\_APPLE\_\_

result["platform"] = "apple";

#elif defined \_\_unix\_\_

result["platform"] = "unix";

#else

result["platform"] = "unknown";

#endif

#if defined(\_\_ICC) || defined(\_\_INTEL\_COMPILER)

result["compiler"] = {{"family", "icc"}, {"version", \_\_INTEL\_COMPILER}};

#elif defined(\_\_clang\_\_)

result["compiler"] = {{"family", "clang"}, {"version", \_\_clang\_version\_\_}};

#elif defined(\_\_GNUC\_\_) || defined(\_\_GNUG\_\_)

result["compiler"] = {{"family", "gcc"}, {"version", std::to\_string(\_\_GNUC\_\_) + "." + std::to\_string(\_\_GNUC\_MINOR\_\_) + "." + std::to\_string(\_\_GNUC\_PATCHLEVEL\_\_)}};

#elif defined(\_\_HP\_cc) || defined(\_\_HP\_aCC)

result["compiler"] = "hp"

#elif defined(\_\_IBMCPP\_\_)

result["compiler"] = {{"family", "ilecpp"}, {"version", \_\_IBMCPP\_\_}};

#elif defined(\_MSC\_VER)

result["compiler"] = {{"family", "msvc"}, {"version", \_MSC\_VER}};

#elif defined(\_\_PGI)

result["compiler"] = {{"family", "pgcpp"}, {"version", \_\_PGI}};

#elif defined(\_\_SUNPRO\_CC)

result["compiler"] = {{"family", "sunpro"}, {"version", \_\_SUNPRO\_CC}};

#else

result["compiler"] = {{"family", "unknown"}, {"version", "unknown"}};

#endif

#ifdef \_\_cplusplus

result["compiler"]["c++"] = std::to\_string(\_\_cplusplus);

#else

result["compiler"]["c++"] = "unknown";

#endif

return result;

}

///////////////////////////

// JSON value data types //

///////////////////////////

/// @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.

/// @{

#if defined(JSON\_HAS\_CPP\_14)

// Use transparent comparator if possible, combined with perfect forwarding

// on find() and count() calls prevents unnecessary string construction.

using object\_comparator\_t = std::less<>;

#else

using object\_comparator\_t = std::less<StringType>;

#endif

/\*!

@brief a type for an object

[RFC 7159](http://rfc7159.net/rfc7159) describes JSON objects as follows:

> An object is an unordered collection of zero or more name/value pairs,

> where a name is a string and a value is a string, number, boolean, null,

> object, or array.

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

described below.

@tparam ObjectType the container to store objects (e.g., `std::map` or

`std::unordered\_map`)

@tparam StringType the type of the keys or names (e.g., `std::string`).

The comparison function `std::less<StringType>` is used to order elements

inside the container.

@tparam AllocatorType the allocator to use for objects (e.g.,

`std::allocator`)

#### Default type

With the default values for @a ObjectType (`std::map`), @a StringType

(`std::string`), and @a AllocatorType (`std::allocator`), the default

value for @a object\_t is:

@code {.cpp}

std::map<

std::string, // key\_type

basic\_json, // value\_type

std::less<std::string>, // key\_compare

std::allocator<std::pair<const std::string, basic\_json>> // allocator\_type

>

@endcode

#### Behavior

The choice of @a object\_t influences the behavior of the JSON class. With

the default type, objects have the following behavior:

- When all names are unique, objects will be interoperable in the sense

that all software implementations receiving that object will agree on

the name-value mappings.

- When the names within an object are not unique, it is unspecified which

one of the values for a given key will be chosen. For instance,

`{"key": 2, "key": 1}` could be equal to either `{"key": 1}` or

`{"key": 2}`.

- Internally, name/value pairs are stored in lexicographical order of the

names. Objects will also be serialized (see @ref dump) in this order.

For instance, `{"b": 1, "a": 2}` and `{"a": 2, "b": 1}` will be stored

and serialized as `{"a": 2, "b": 1}`.

- When comparing objects, the order of the name/value pairs is irrelevant.

This makes objects interoperable in the sense that they will not be

affected by these differences. For instance, `{"b": 1, "a": 2}` and

`{"a": 2, "b": 1}` will be treated as equal.

#### Limits

[RFC 7159](http://rfc7159.net/rfc7159) specifies:

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

In this class, the object's limit of nesting is not explicitly constrained.

However, a maximum depth of nesting may be introduced by the compiler or

runtime environment. A theoretical limit can be queried by calling the

@ref max\_size function of a JSON object.

#### Storage

Objects are stored as pointers in a @ref basic\_json type. That is, for any

access to object values, a pointer of type `object\_t\*` must be

dereferenced.

@sa @ref array\_t -- type for an array value

@since version 1.0.0

@note The order name/value pairs are added to the object is \*not\*

preserved by the library. Therefore, iterating an object may return

name/value pairs in a different order than they were originally stored. In

fact, keys will be traversed in alphabetical order as `std::map` with

`std::less` is used by default. Please note this behavior conforms to [RFC

7159](http://rfc7159.net/rfc7159), because any order implements the

specified "unordered" nature of JSON objects.

\*/

using object\_t = ObjectType<StringType,

basic\_json,

object\_comparator\_t,

AllocatorType<std::pair<const StringType,

basic\_json>>>;

/\*!

@brief a type for an array

[RFC 7159](http://rfc7159.net/rfc7159) describes JSON arrays as follows:

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

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

explained below.

@tparam ArrayType container type to store arrays (e.g., `std::vector` or

`std::list`)

@tparam AllocatorType allocator to use for arrays (e.g., `std::allocator`)

#### Default type

With the default values for @a ArrayType (`std::vector`) and @a

AllocatorType (`std::allocator`), the default value for @a array\_t is:

@code {.cpp}

std::vector<

basic\_json, // value\_type

std::allocator<basic\_json> // allocator\_type

>

@endcode

#### Limits

[RFC 7159](http://rfc7159.net/rfc7159) specifies:

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

In this class, the array's limit of nesting is not explicitly constrained.

However, a maximum depth of nesting may be introduced by the compiler or

runtime environment. A theoretical limit can be queried by calling the

@ref max\_size function of a JSON array.

#### Storage

Arrays are stored as pointers in a @ref basic\_json type. That is, for any

access to array values, a pointer of type `array\_t\*` must be dereferenced.

@sa @ref object\_t -- type for an object value

@since version 1.0.0

\*/

using array\_t = ArrayType<basic\_json, AllocatorType<basic\_json>>;

/\*!

@brief a type for a string

[RFC 7159](http://rfc7159.net/rfc7159) describes JSON strings as follows:

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

To store objects in C++, a type is defined by the template parameter

described below. Unicode values are split by the JSON class into

byte-sized characters during deserialization.

@tparam StringType the container to store strings (e.g., `std::string`).

Note this container is used for keys/names in objects, see @ref object\_t.

#### Default type

With the default values for @a StringType (`std::string`), the default

value for @a string\_t is:

@code {.cpp}

std::string

@endcode

#### Encoding

Strings are stored in UTF-8 encoding. Therefore, functions like

`std::string::size()` or `std::string::length()` return the number of

bytes in the string rather than the number of characters or glyphs.

#### String comparison

[RFC 7159](http://rfc7159.net/rfc7159) states:

> Software implementations are typically required to test names of object

> members for equality. Implementations that transform the textual

> representation into sequences of Unicode code units and then perform the

> comparison numerically, code unit by code unit, are interoperable in the

> sense that implementations will agree in all cases on equality or

> inequality of two strings. For example, implementations that compare

> strings with escaped characters unconverted may incorrectly find that

> `"a\\b"` and `"a\u005Cb"` are not equal.

This implementation is interoperable as it does compare strings code unit

by code unit.

#### Storage

String values are stored as pointers in a @ref basic\_json type. That is,

for any access to string values, a pointer of type `string\_t\*` must be

dereferenced.

@since version 1.0.0

\*/

using string\_t = StringType;

/\*!

@brief a type for a boolean

[RFC 7159](http://rfc7159.net/rfc7159) implicitly describes a boolean as a

type which differentiates the two literals `true` and `false`.

To store objects in C++, a type is defined by the template parameter @a

BooleanType which chooses the type to use.

#### Default type

With the default values for @a BooleanType (`bool`), the default value for

@a boolean\_t is:

@code {.cpp}

bool

@endcode

#### Storage

Boolean values are stored directly inside a @ref basic\_json type.

@since version 1.0.0

\*/

using boolean\_t = BooleanType;

/\*!

@brief a type for a number (integer)

[RFC 7159](http://rfc7159.net/rfc7159) describes numbers as follows:

> The representation of numbers is similar to that used in most

> programming languages. A number is represented in base 10 using decimal

> digits. It contains an integer component that may be prefixed with an

> optional minus sign, which may be followed by a fraction part and/or an

> exponent part. Leading zeros are not allowed. (...) Numeric values that

> cannot be represented in the grammar below (such as Infinity and NaN)

> are not permitted.

This description includes both integer and floating-point numbers.

However, C++ allows more precise storage if it is known whether the number

is a signed integer, an unsigned integer or a floating-point number.

Therefore, three different types, @ref number\_integer\_t, @ref

number\_unsigned\_t and @ref number\_float\_t are used.

To store integer numbers in C++, a type is defined by the template

parameter @a NumberIntegerType which chooses the type to use.

#### Default type

With the default values for @a NumberIntegerType (`int64\_t`), the default

value for @a number\_integer\_t is:

@code {.cpp}

int64\_t

@endcode

#### Default behavior

- The restrictions about leading zeros is not enforced in C++. Instead,

leading zeros in integer literals lead to an interpretation as octal

number. Internally, the value will be stored as decimal number. For

instance, the C++ integer literal `010` will be serialized to `8`.

During deserialization, leading zeros yield an error.

- Not-a-number (NaN) values will be serialized to `null`.

#### Limits

[RFC 7159](http://rfc7159.net/rfc7159) specifies:

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

When the default type is used, the maximal integer number that can be

stored is `9223372036854775807` (INT64\_MAX) and the minimal integer number

that can be stored is `-9223372036854775808` (INT64\_MIN). Integer numbers

that are out of range will yield over/underflow when used in a

constructor. During deserialization, too large or small integer numbers

will be automatically be stored as @ref number\_unsigned\_t or @ref

number\_float\_t.

[RFC 7159](http://rfc7159.net/rfc7159) further states:

> Note that when such software is used, numbers that are integers and are

> in the range \f$[-2^{53}+1, 2^{53}-1]\f$ are interoperable in the sense

> that implementations will agree exactly on their numeric values.

As this range is a subrange of the exactly supported range [INT64\_MIN,

INT64\_MAX], this class's integer type is interoperable.

#### Storage

Integer number values are stored directly inside a @ref basic\_json type.

@sa @ref number\_float\_t -- type for number values (floating-point)

@sa @ref number\_unsigned\_t -- type for number values (unsigned integer)

@since version 1.0.0

\*/

using number\_integer\_t = NumberIntegerType;

/\*!

@brief a type for a number (unsigned)

[RFC 7159](http://rfc7159.net/rfc7159) describes numbers as follows:

> The representation of numbers is similar to that used in most

> programming languages. A number is represented in base 10 using decimal

> digits. It contains an integer component that may be prefixed with an

> optional minus sign, which may be followed by a fraction part and/or an

> exponent part. Leading zeros are not allowed. (...) Numeric values that

> cannot be represented in the grammar below (such as Infinity and NaN)

> are not permitted.

This description includes both integer and floating-point numbers.

However, C++ allows more precise storage if it is known whether the number

is a signed integer, an unsigned integer or a floating-point number.

Therefore, three different types, @ref number\_integer\_t, @ref

number\_unsigned\_t and @ref number\_float\_t are used.

To store unsigned integer numbers in C++, a type is defined by the

template parameter @a NumberUnsignedType which chooses the type to use.

#### Default type

With the default values for @a NumberUnsignedType (`uint64\_t`), the

default value for @a number\_unsigned\_t is:

@code {.cpp}

uint64\_t

@endcode

#### Default behavior

- The restrictions about leading zeros is not enforced in C++. Instead,

leading zeros in integer literals lead to an interpretation as octal

number. Internally, the value will be stored as decimal number. For

instance, the C++ integer literal `010` will be serialized to `8`.

During deserialization, leading zeros yield an error.

- Not-a-number (NaN) values will be serialized to `null`.

#### Limits

[RFC 7159](http://rfc7159.net/rfc7159) specifies:

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

When the default type is used, the maximal integer number that can be

stored is `18446744073709551615` (UINT64\_MAX) and the minimal integer

number that can be stored is `0`. Integer numbers that are out of range

will yield over/underflow when used in a constructor. During

deserialization, too large or small integer numbers will be automatically

be stored as @ref number\_integer\_t or @ref number\_float\_t.

[RFC 7159](http://rfc7159.net/rfc7159) further states:

> Note that when such software is used, numbers that are integers and are

> in the range \f$[-2^{53}+1, 2^{53}-1]\f$ are interoperable in the sense

> that implementations will agree exactly on their numeric values.

As this range is a subrange (when considered in conjunction with the

number\_integer\_t type) of the exactly supported range [0, UINT64\_MAX],

this class's integer type is interoperable.

#### Storage

Integer number values are stored directly inside a @ref basic\_json type.

@sa @ref number\_float\_t -- type for number values (floating-point)

@sa @ref number\_integer\_t -- type for number values (integer)

@since version 2.0.0

\*/

using number\_unsigned\_t = NumberUnsignedType;

/\*!

@brief a type for a number (floating-point)

[RFC 7159](http://rfc7159.net/rfc7159) describes numbers as follows:

> The representation of numbers is similar to that used in most

> programming languages. A number is represented in base 10 using decimal

> digits. It contains an integer component that may be prefixed with an

> optional minus sign, which may be followed by a fraction part and/or an

> exponent part. Leading zeros are not allowed. (...) Numeric values that

> cannot be represented in the grammar below (such as Infinity and NaN)

> are not permitted.

This description includes both integer and floating-point numbers.

However, C++ allows more precise storage if it is known whether the number

is a signed integer, an unsigned integer or a floating-point number.

Therefore, three different types, @ref number\_integer\_t, @ref

number\_unsigned\_t and @ref number\_float\_t are used.

To store floating-point numbers in C++, a type is defined by the template

parameter @a NumberFloatType which chooses the type to use.

#### Default type

With the default values for @a NumberFloatType (`double`), the default

value for @a number\_float\_t is:

@code {.cpp}

double

@endcode

#### Default behavior

- The restrictions about leading zeros is not enforced in C++. Instead,

leading zeros in floating-point literals will be ignored. Internally,

the value will be stored as decimal number. For instance, the C++

floating-point literal `01.2` will be serialized to `1.2`. During

deserialization, leading zeros yield an error.

- Not-a-number (NaN) values will be serialized to `null`.

#### Limits

[RFC 7159](http://rfc7159.net/rfc7159) states:

> This specification allows implementations to set limits on the range and

> precision of numbers accepted. Since software that implements IEEE

> 754-2008 binary64 (double precision) numbers is generally available and

> widely used, good interoperability can be achieved by implementations

> that expect no more precision or range than these provide, in the sense

> that implementations will approximate JSON numbers within the expected

> precision.

This implementation does exactly follow this approach, as it uses double

precision floating-point numbers. Note values smaller than

`-1.79769313486232e+308` and values greater than `1.79769313486232e+308`

will be stored as NaN internally and be serialized to `null`.

#### Storage

Floating-point number values are stored directly inside a @ref basic\_json

type.

@sa @ref number\_integer\_t -- type for number values (integer)

@sa @ref number\_unsigned\_t -- type for number values (unsigned integer)

@since version 1.0.0

\*/

using number\_float\_t = NumberFloatType;

/// @}

private:

/// helper for exception-safe object creation

template<typename T, typename... Args>

JSON\_HEDLEY\_RETURNS\_NON\_NULL

static T\* create(Args&& ... args)

{

AllocatorType<T> alloc;

using AllocatorTraits = std::allocator\_traits<AllocatorType<T>>;

auto deleter = [&](T \* object)

{

AllocatorTraits::deallocate(alloc, object, 1);

};

std::unique\_ptr<T, decltype(deleter)> object(AllocatorTraits::allocate(alloc, 1), deleter);

AllocatorTraits::construct(alloc, object.get(), std::forward<Args>(args)...);

assert(object != nullptr);

return object.release();

}

////////////////////////

// JSON value storage //

////////////////////////

/\*!

@brief a JSON value

The actual storage for a JSON value of the @ref basic\_json class. This

union combines the different storage types for the JSON value types

defined in @ref value\_t.

JSON type | value\_t type | used type

--------- | --------------- | ------------------------

object | object | pointer to @ref object\_t

array | array | pointer to @ref array\_t

string | string | pointer to @ref string\_t

boolean | boolean | @ref boolean\_t

number | number\_integer | @ref number\_integer\_t

number | number\_unsigned | @ref number\_unsigned\_t

number | number\_float | @ref number\_float\_t

null | null | \*no value is stored\*

@note Variable-length types (objects, arrays, and strings) are stored as

pointers. The size of the union should not exceed 64 bits if the default

value types are used.

@since version 1.0.0

\*/

union json\_value

{

/// object (stored with pointer to save storage)

object\_t\* object;

/// array (stored with pointer to save storage)

array\_t\* array;

/// string (stored with pointer to save storage)

string\_t\* string;

/// boolean

boolean\_t boolean;

/// number (integer)

number\_integer\_t number\_integer;

/// number (unsigned integer)

number\_unsigned\_t number\_unsigned;

/// number (floating-point)

number\_float\_t number\_float;

/// default constructor (for null values)

json\_value() = default;

/// constructor for booleans

json\_value(boolean\_t v) noexcept : boolean(v) {}

/// constructor for numbers (integer)

json\_value(number\_integer\_t v) noexcept : number\_integer(v) {}

/// constructor for numbers (unsigned)

json\_value(number\_unsigned\_t v) noexcept : number\_unsigned(v) {}

/// constructor for numbers (floating-point)

json\_value(number\_float\_t v) noexcept : number\_float(v) {}

/// constructor for empty values of a given type

json\_value(value\_t t)

{

switch (t)

{

case value\_t::object:

{

object = create<object\_t>();

break;

}

case value\_t::array:

{

array = create<array\_t>();

break;

}

case value\_t::string:

{

string = create<string\_t>("");

break;

}

case value\_t::boolean:

{

boolean = boolean\_t(false);

break;

}

case value\_t::number\_integer:

{

number\_integer = number\_integer\_t(0);

break;

}

case value\_t::number\_unsigned:

{

number\_unsigned = number\_unsigned\_t(0);

break;

}

case value\_t::number\_float:

{

number\_float = number\_float\_t(0.0);

break;

}

case value\_t::null:

{

object = nullptr; // silence warning, see #821

break;

}

default:

{

object = nullptr; // silence warning, see #821

if (JSON\_HEDLEY\_UNLIKELY(t == value\_t::null))

{

JSON\_THROW(other\_error::create(500, "961c151d2e87f2686a955a9be24d316f1362bf21 3.7.3")); // LCOV\_EXCL\_LINE

}

break;

}

}

}

/// constructor for strings

json\_value(const string\_t& value)

{

string = create<string\_t>(value);

}

/// constructor for rvalue strings

json\_value(string\_t&& value)

{

string = create<string\_t>(std::move(value));

}

/// constructor for objects

json\_value(const object\_t& value)

{

object = create<object\_t>(value);

}

/// constructor for rvalue objects

json\_value(object\_t&& value)

{

object = create<object\_t>(std::move(value));

}

/// constructor for arrays

json\_value(const array\_t& value)

{

array = create<array\_t>(value);

}

/// constructor for rvalue arrays

json\_value(array\_t&& value)

{

array = create<array\_t>(std::move(value));

}

void destroy(value\_t t) noexcept

{

// flatten the current json\_value to a heap-allocated stack

std::vector<basic\_json> stack;

// move the top-level items to stack

if (t == value\_t::array)

{

stack.reserve(array->size());

std::move(array->begin(), array->end(), std::back\_inserter(stack));

}

else if (t == value\_t::object)

{

stack.reserve(object->size());

for (auto&& it : \*object)

{

stack.push\_back(std::move(it.second));

}

}

while (not stack.empty())

{

// move the last item to local variable to be processed

basic\_json current\_item(std::move(stack.back()));

stack.pop\_back();

// if current\_item is array/object, move

// its children to the stack to be processed later

if (current\_item.is\_array())

{

std::move(current\_item.m\_value.array->begin(), current\_item.m\_value.array->end(),

std::back\_inserter(stack));

current\_item.m\_value.array->clear();

}

else if (current\_item.is\_object())

{

for (auto&& it : \*current\_item.m\_value.object)

{

stack.push\_back(std::move(it.second));

}

current\_item.m\_value.object->clear();

}

// it's now safe that current\_item get destructed

// since it doesn't have any children

}

switch (t)

{

case value\_t::object:

{

AllocatorType<object\_t> alloc;

std::allocator\_traits<decltype(alloc)>::destroy(alloc, object);

std::allocator\_traits<decltype(alloc)>::deallocate(alloc, object, 1);

break;

}

case value\_t::array:

{

AllocatorType<array\_t> alloc;

std::allocator\_traits<decltype(alloc)>::destroy(alloc, array);

std::allocator\_traits<decltype(alloc)>::deallocate(alloc, array, 1);

break;

}

case value\_t::string:

{

AllocatorType<string\_t> alloc;

std::allocator\_traits<decltype(alloc)>::destroy(alloc, string);

std::allocator\_traits<decltype(alloc)>::deallocate(alloc, string, 1);

break;

}

default:

{

break;

}

}

}

};

/\*!

@brief checks the class invariants

This function asserts the class invariants. It needs to be called at the

end of every constructor to make sure that created objects respect the

invariant. Furthermore, it has to be called each time the type of a JSON

value is changed, because the invariant expresses a relationship between

@a m\_type and @a m\_value.

\*/

void assert\_invariant() const noexcept

{

assert(m\_type != value\_t::object or m\_value.object != nullptr);

assert(m\_type != value\_t::array or m\_value.array != nullptr);

assert(m\_type != value\_t::string or m\_value.string != nullptr);

}

public:

//////////////////////////

// JSON parser callback //

//////////////////////////

/\*!

@brief parser event types

The parser callback distinguishes the following events:

- `object\_start`: the parser read `{` and started to process a JSON object

- `key`: the parser read a key of a value in an object

- `object\_end`: the parser read `}` and finished processing a JSON object

- `array\_start`: the parser read `[` and started to process a JSON array

- `array\_end`: the parser read `]` and finished processing a JSON array

- `value`: the parser finished reading a JSON value

@image html callback\_events.png "Example when certain parse events are triggered"

@sa @ref parser\_callback\_t for more information and examples

\*/

using parse\_event\_t = typename parser::parse\_event\_t;

/\*!

@brief per-element parser callback type

With a parser callback function, the result of parsing a JSON text can be

influenced. When passed to @ref parse, it is called on certain events

(passed as @ref parse\_event\_t via parameter @a event) with a set recursion

depth @a depth and context JSON value @a parsed. The return value of the

callback function is a boolean indicating whether the element that emitted

the callback shall be kept or not.

We distinguish six scenarios (determined by the event type) in which the

callback function can be called. The following table describes the values

of the parameters @a depth, @a event, and @a parsed.

parameter @a event | description | parameter @a depth | parameter @a parsed

------------------ | ----------- | ------------------ | -------------------

parse\_event\_t::object\_start | the parser read `{` and started to process a JSON object | depth of the parent of the JSON object | a JSON value with type discarded

parse\_event\_t::key | the parser read a key of a value in an object | depth of the currently parsed JSON object | a JSON string containing the key

parse\_event\_t::object\_end | the parser read `}` and finished processing a JSON object | depth of the parent of the JSON object | the parsed JSON object

parse\_event\_t::array\_start | the parser read `[` and started to process a JSON array | depth of the parent of the JSON array | a JSON value with type discarded

parse\_event\_t::array\_end | the parser read `]` and finished processing a JSON array | depth of the parent of the JSON array | the parsed JSON array

parse\_event\_t::value | the parser finished reading a JSON value | depth of the value | the parsed JSON value

@image html callback\_events.png "Example when certain parse events are triggered"

Discarding a value (i.e., returning `false`) has different effects

depending on the context in which function was called:

- Discarded values in structured types are skipped. That is, the parser

will behave as if the discarded value was never read.

- In case a value outside a structured type is skipped, it is replaced

with `null`. This case happens if the top-level element is skipped.

@param[in] depth the depth of the recursion during parsing

@param[in] event an event of type parse\_event\_t indicating the context in

the callback function has been called

@param[in,out] parsed the current intermediate parse result; note that

writing to this value has no effect for parse\_event\_t::key events

@return Whether the JSON value which called the function during parsing

should be kept (`true`) or not (`false`). In the latter case, it is either

skipped completely or replaced by an empty discarded object.

@sa @ref parse for examples

@since version 1.0.0

\*/

using parser\_callback\_t = typename parser::parser\_callback\_t;

//////////////////

// constructors //

//////////////////

/// @name constructors and destructors

/// Constructors of class @ref basic\_json, copy/move constructor, copy

/// assignment, static functions creating objects, and the destructor.

/// @{

/\*!

@brief create an empty value with a given type

Create an empty JSON value with a given type. The value will be default

initialized with an empty value which depends on the type:

Value type | initial value

----------- | -------------

null | `null`

boolean | `false`

string | `""`

number | `0`

object | `{}`

array | `[]`

@param[in] v the type of the value to create

@complexity Constant.

@exceptionsafety Strong guarantee: if an exception is thrown, there are no

changes to any JSON value.

@liveexample{The following code shows the constructor for different @ref

value\_t values,basic\_json\_\_value\_t}

@sa @ref clear() -- restores the postcondition of this constructor

@since version 1.0.0

\*/

basic\_json(const value\_t v)

: m\_type(v), m\_value(v)

{

assert\_invariant();

}

/\*!

@brief create a null object

Create a `null` JSON value. It either takes a null pointer as parameter

(explicitly creating `null`) or no parameter (implicitly creating `null`).

The passed null pointer itself is not read -- it is only used to choose

the right constructor.

@complexity Constant.

@exceptionsafety No-throw guarantee: this constructor never throws

exceptions.

@liveexample{The following code shows the constructor with and without a

null pointer parameter.,basic\_json\_\_nullptr\_t}

@since version 1.0.0

\*/

basic\_json(std::nullptr\_t = nullptr) noexcept

: basic\_json(value\_t::null)

{

assert\_invariant();

}

/\*!

@brief create a JSON value

This is a "catch all" constructor for all compatible JSON types; that is,

types for which a `to\_json()` method exists. The constructor forwards the

parameter @a val to that method (to `json\_serializer<U>::to\_json` method

with `U = uncvref\_t<CompatibleType>`, to be exact).

Template type @a CompatibleType includes, but is not limited to, the

following types:

- \*\*arrays\*\*: @ref array\_t and all kinds of compatible containers such as

`std::vector`, `std::deque`, `std::list`, `std::forward\_list`,

`std::array`, `std::valarray`, `std::set`, `std::unordered\_set`,

`std::multiset`, and `std::unordered\_multiset` with a `value\_type` from

which a @ref basic\_json value can be constructed.

- \*\*objects\*\*: @ref object\_t and all kinds of compatible associative

containers such as `std::map`, `std::unordered\_map`, `std::multimap`,

and `std::unordered\_multimap` with a `key\_type` compatible to

@ref string\_t and a `value\_type` from which a @ref basic\_json value can

be constructed.

- \*\*strings\*\*: @ref string\_t, string literals, and all compatible string

containers can be used.

- \*\*numbers\*\*: @ref number\_integer\_t, @ref number\_unsigned\_t,

@ref number\_float\_t, and all convertible number types such as `int`,

`size\_t`, `int64\_t`, `float` or `double` can be used.

- \*\*boolean\*\*: @ref boolean\_t / `bool` can be used.

See the examples below.

@tparam CompatibleType a type such that:

- @a CompatibleType is not derived from `std::istream`,

- @a CompatibleType is not @ref basic\_json (to avoid hijacking copy/move

constructors),

- @a CompatibleType is not a different @ref basic\_json type (i.e. with different template arguments)

- @a CompatibleType is not a @ref basic\_json nested type (e.g.,

@ref json\_pointer, @ref iterator, etc ...)

- @ref @ref json\_serializer<U> has a

`to\_json(basic\_json\_t&, CompatibleType&&)` method

@tparam U = `uncvref\_t<CompatibleType>`

@param[in] val the value to be forwarded to the respective constructor

@complexity Usually linear in the size of the passed @a val, also

depending on the implementation of the called `to\_json()`

method.

@exceptionsafety Depends on the called constructor. For types directly

supported by the library (i.e., all types for which no `to\_json()` function

was provided), strong guarantee holds: if an exception is thrown, there are

no changes to any JSON value.

@liveexample{The following code shows the constructor with several

compatible types.,basic\_json\_\_CompatibleType}

@since version 2.1.0

\*/

template <typename CompatibleType,

typename U = detail::uncvref\_t<CompatibleType>,

detail::enable\_if\_t<

not detail::is\_basic\_json<U>::value and detail::is\_compatible\_type<basic\_json\_t, U>::value, int> = 0>

basic\_json(CompatibleType && val) noexcept(noexcept(

JSONSerializer<U>::to\_json(std::declval<basic\_json\_t&>(),

std::forward<CompatibleType>(val))))

{

JSONSerializer<U>::to\_json(\*this, std::forward<CompatibleType>(val));

assert\_invariant();

}

/\*!

@brief create a JSON value from an existing one

This is a constructor for existing @ref basic\_json types.

It does not hijack copy/move constructors, since the parameter has different

template arguments than the current ones.

The constructor tries to convert the internal @ref m\_value of the parameter.

@tparam BasicJsonType a type such that:

- @a BasicJsonType is a @ref basic\_json type.

- @a BasicJsonType has different template arguments than @ref basic\_json\_t.

@param[in] val the @ref basic\_json value to be converted.

@complexity Usually linear in the size of the passed @a val, also

depending on the implementation of the called `to\_json()`

method.

@exceptionsafety Depends on the called constructor. For types directly

supported by the library (i.e., all types for which no `to\_json()` function

was provided), strong guarantee holds: if an exception is thrown, there are

no changes to any JSON value.

@since version 3.2.0

\*/

template <typename BasicJsonType,

detail::enable\_if\_t<

detail::is\_basic\_json<BasicJsonType>::value and not std::is\_same<basic\_json, BasicJsonType>::value, int> = 0>

basic\_json(const BasicJsonType& val)

{

using other\_boolean\_t = typename BasicJsonType::boolean\_t;

using other\_number\_float\_t = typename BasicJsonType::number\_float\_t;

using other\_number\_integer\_t = typename BasicJsonType::number\_integer\_t;

using other\_number\_unsigned\_t = typename BasicJsonType::number\_unsigned\_t;

using other\_string\_t = typename BasicJsonType::string\_t;

using other\_object\_t = typename BasicJsonType::object\_t;

using other\_array\_t = typename BasicJsonType::array\_t;

switch (val.type())

{

case value\_t::boolean:

JSONSerializer<other\_boolean\_t>::to\_json(\*this, val.template get<other\_boolean\_t>());

break;

case value\_t::number\_float:

JSONSerializer<other\_number\_float\_t>::to\_json(\*this, val.template get<other\_number\_float\_t>());

break;

case value\_t::number\_integer:

JSONSerializer<other\_number\_integer\_t>::to\_json(\*this, val.template get<other\_number\_integer\_t>());

break;

case value\_t::number\_unsigned:

JSONSerializer<other\_number\_unsigned\_t>::to\_json(\*this, val.template get<other\_number\_unsigned\_t>());

break;

case value\_t::string:

JSONSerializer<other\_string\_t>::to\_json(\*this, val.template get\_ref<const other\_string\_t&>());

break;

case value\_t::object:

JSONSerializer<other\_object\_t>::to\_json(\*this, val.template get\_ref<const other\_object\_t&>());

break;

case value\_t::array:

JSONSerializer<other\_array\_t>::to\_json(\*this, val.template get\_ref<const other\_array\_t&>());

break;

case value\_t::null:

\*this = nullptr;

break;

case value\_t::discarded:

m\_type = value\_t::discarded;

break;

default: // LCOV\_EXCL\_LINE

assert(false); // LCOV\_EXCL\_LINE

}

assert\_invariant();

}

/\*!

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

Creates a JSON value of type array or object from the passed initializer

list @a init. In case @a type\_deduction is `true` (default), the type of

the JSON value to be created is deducted from the initializer list @a init

according to the following rules:

1. If the list is empty, an empty JSON object value `{}` is created.

2. If the list consists of pairs whose first element is a string, a JSON

object value is created where the first elements of the pairs are

treated as keys and the second elements are as values.

3. In all other cases, an array is created.

The rules aim to create the best fit between a C++ initializer list and

JSON values. The rationale is as follows:

1. The empty initializer list is written as `{}` which is exactly an empty

JSON object.

2. C++ has no way of describing mapped types other than to list a list of

pairs. As JSON requires that keys must be of type string, rule 2 is the

weakest constraint one can pose on initializer lists to interpret them

as an object.

3. In all other cases, the initializer list could not be interpreted as

JSON object type, so interpreting it as JSON array type is safe.

With the rules described above, the following JSON values cannot be

expressed by an initializer list:

- the empty array (`[]`): use @ref array(initializer\_list\_t)

with an empty initializer list in this case

- arrays whose elements satisfy rule 2: use @ref

array(initializer\_list\_t) with the same initializer list

in this case

@note When used without parentheses around an empty initializer list, @ref

basic\_json() is called instead of this function, yielding the JSON null

value.

@param[in] init initializer list with JSON values

@param[in] type\_deduction internal parameter; when set to `true`, the type

of the JSON value is deducted from the initializer list @a init; when set

to `false`, the type provided via @a manual\_type is forced. This mode is

used by the functions @ref array(initializer\_list\_t) and

@ref object(initializer\_list\_t).

@param[in] manual\_type internal parameter; when @a type\_deduction is set

to `false`, the created JSON value will use the provided type (only @ref

value\_t::array and @ref value\_t::object are valid); when @a type\_deduction

is set to `true`, this parameter has no effect

@throw type\_error.301 if @a type\_deduction is `false`, @a manual\_type is

`value\_t::object`, but @a init contains an element which is not a pair

whose first element is a string. In this case, the constructor could not

create an object. If @a type\_deduction would have be `true`, an array

would have been created. See @ref object(initializer\_list\_t)

for an example.

@complexity Linear in the size of the initializer list @a init.

@exceptionsafety Strong guarantee: if an exception is thrown, there are no

changes to any JSON value.

@liveexample{The example below shows how JSON values are created from

initializer lists.,basic\_json\_\_list\_init\_t}

@sa @ref array(initializer\_list\_t) -- create a JSON array

value from an initializer list

@sa @ref object(initializer\_list\_t) -- create a JSON object

value from an initializer list

@since version 1.0.0

\*/

basic\_json(initializer\_list\_t init,

bool type\_deduction = true,

value\_t manual\_type = value\_t::array)

{

// check if each element is an array with two elements whose first

// element is a string

bool is\_an\_object = std::all\_of(init.begin(), init.end(),

[](const detail::json\_ref<basic\_json>& element\_ref)

{

return element\_ref->is\_array() and element\_ref->size() == 2 and (\*element\_ref)[0].is\_string();

});

// adjust type if type deduction is not wanted

if (not type\_deduction)

{

// if array is wanted, do not create an object though possible

if (manual\_type == value\_t::array)

{

is\_an\_object = false;

}

// if object is wanted but impossible, throw an exception

if (JSON\_HEDLEY\_UNLIKELY(manual\_type == value\_t::object and not is\_an\_object))

{

JSON\_THROW(type\_error::create(301, "cannot create object from initializer list"));

}

}

if (is\_an\_object)

{

// the initializer list is a list of pairs -> create object

m\_type = value\_t::object;

m\_value = value\_t::object;

std::for\_each(init.begin(), init.end(), [this](const detail::json\_ref<basic\_json>& element\_ref)

{

auto element = element\_ref.moved\_or\_copied();

m\_value.object->emplace(

std::move(\*((\*element.m\_value.array)[0].m\_value.string)),

std::move((\*element.m\_value.array)[1]));

});

}

else

{

// the initializer list describes an array -> create array

m\_type = value\_t::array;

m\_value.array = create<array\_t>(init.begin(), init.end());

}

assert\_invariant();

}

/\*!

@brief explicitly create an array from an initializer list

Creates a JSON array value from a given initializer list. That is, given a

list of values `a, b, c`, creates the JSON value `[a, b, c]`. If the

initializer list is empty, the empty array `[]` is created.

@note This function is only needed to express two edge cases that cannot

be realized with the initializer list constructor (@ref

basic\_json(initializer\_list\_t, bool, value\_t)). These cases

are:

1. creating an array whose elements are all pairs whose first element is a

string -- in this case, the initializer list constructor would create an

object, taking the first elements as keys

2. creating an empty array -- passing the empty initializer list to the

initializer list constructor yields an empty object

@param[in] init initializer list with JSON values to create an array from

(optional)

@return JSON array value

@complexity Linear in the size of @a init.

@exceptionsafety Strong guarantee: if an exception is thrown, there are no

changes to any JSON value.

@liveexample{The following code shows an example for the `array`

function.,array}

@sa @ref basic\_json(initializer\_list\_t, bool, value\_t) --

create a JSON value from an initializer list

@sa @ref object(initializer\_list\_t) -- create a JSON object

value from an initializer list

@since version 1.0.0

\*/

JSON\_HEDLEY\_WARN\_UNUSED\_RESULT

static basic\_json array(initializer\_list\_t init = {})

{

return basic\_json(init, false, value\_t::array);

}

/\*!

@brief explicitly create an object from an initializer list

Creates a JSON object value from a given initializer list. The initializer

lists elements must be pairs, and their first elements must be strings. If

the initializer list is empty, the empty object `{}` is created.

@note This function is only added for symmetry reasons. In contrast to the

related function @ref array(initializer\_list\_t), there are

no cases which can only be expressed by this function. That is, any

initializer list @a init can also be passed to the initializer list

constructor @ref basic\_json(initializer\_list\_t, bool, value\_t).

@param[in] init initializer list to create an object from (optional)

@return JSON object value

@throw type\_error.301 if @a init is not a list of pairs whose first

elements are strings. In this case, no object can be created. When such a

value is passed to @ref basic\_json(initializer\_list\_t, bool, value\_t),

an array would have been created from the passed initializer list @a init.

See example below.

@complexity Linear in the size of @a init.

@exceptionsafety Strong guarantee: if an exception is thrown, there are no

changes to any JSON value.

@liveexample{The following code shows an example for the `object`

function.,object}

@sa @ref basic\_json(initializer\_list\_t, bool, value\_t) --

create a JSON value from an initializer list

@sa @ref array(initializer\_list\_t) -- create a JSON array

value from an initializer list

@since version 1.0.0

\*/

JSON\_HEDLEY\_WARN\_UNUSED\_RESULT

static basic\_json object(initializer\_list\_t init = {})

{

return basic\_json(init, false, value\_t::object);

}

/\*!

@brief construct an array with count copies of given value

Constructs a JSON array value by creating @a cnt copies of a passed value.

In case @a cnt is `0`, an empty array is created.

@param[in] cnt the number of JSON copies of @a val to create

@param[in] val the JSON value to copy

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

@complexity Linear in @a cnt.

@exceptionsafety Strong guarantee: if an exception is thrown, there are no

changes to any JSON value.

@liveexample{The following code shows examples for the @ref

basic\_json(size\_type\, const basic\_json&)

constructor.,basic\_json\_\_size\_type\_basic\_json}

@since version 1.0.0

\*/

basic\_json(size\_type cnt, const basic\_json& val)

: m\_type(value\_t::array)

{

m\_value.array = create<array\_t>(cnt, val);

assert\_invariant();

}

/\*!

@brief construct a JSON container given an iterator range

Constructs the JSON value with the contents of the range `[first, last)`.

The semantics depends on the different types a JSON value can have:

- In case of a null type, invalid\_iterator.206 is thrown.

- In case of other primitive types (number, boolean, or string), @a first

must be `begin()` and @a last must be `end()`. In this case, the value is

copied. Otherwise, invalid\_iterator.204 is thrown.

- In case of structured types (array, object), the constructor behaves as

similar versions for `std::vector` or `std::map`; that is, a JSON array

or object is constructed from the values in the range.

@tparam InputIT an input iterator type (@ref iterator or @ref

const\_iterator)

@param[in] first begin of the range to copy from (included)

@param[in] last end of the range to copy from (excluded)

@pre Iterators @a first and @a last must be initialized. \*\*This

precondition is enforced with an assertion (see warning).\*\* If

assertions are switched off, a violation of this precondition yields

undefined behavior.

@pre Range `[first, last)` is valid. Usually, this precondition cannot be

checked efficiently. Only certain edge cases are detected; see the

description of the exceptions below. A violation of this precondition

yields undefined behavior.

@warning A precondition is enforced with a runtime assertion that will

result in calling `std::abort` if this precondition is not met.

Assertions can be disabled by defining `NDEBUG` at compile time.

See https://en.cppreference.com/w/cpp/error/assert for more

information.

@throw invalid\_iterator.201 if iterators @a first and @a last are not

compatible (i.e., do not belong to the same JSON value). In this case,

the range `[first, last)` is undefined.

@throw invalid\_iterator.204 if iterators @a first and @a last belong to a

primitive type (number, boolean, or string), but @a first does not point

to the first element any more. In this case, the range `[first, last)` is

undefined. See example code below.

@throw invalid\_iterator.206 if iterators @a first and @a last belong to a

null value. In this case, the range `[first, last)` is undefined.

@complexity Linear in distance between @a first and @a last.

@exceptionsafety Strong guarantee: if an exception is thrown, there are no

changes to any JSON value.

@liveexample{The example below shows several ways to create JSON values by

specifying a subrange with iterators.,basic\_json\_\_InputIt\_InputIt}

@since version 1.0.0

\*/

template<class InputIT, typename std::enable\_if<

std::is\_same<InputIT, typename basic\_json\_t::iterator>::value or

std::is\_same<InputIT, typename basic\_json\_t::const\_iterator>::value, int>::type = 0>

basic\_json(InputIT first, InputIT last)

{

assert(first.m\_object != nullptr);

assert(last.m\_object != nullptr);

// make sure iterator fits the current value

if (JSON\_HEDLEY\_UNLIKELY(first.m\_object != last.m\_object))

{

JSON\_THROW(invalid\_iterator::create(201, "iterators are not compatible"));

}

// copy type from first iterator

m\_type = first.m\_object->m\_type;

// check if iterator range is complete for primitive values

switch (m\_type)

{

case value\_t::boolean:

case value\_t::number\_float:

case value\_t::number\_integer:

case value\_t::number\_unsigned:

case value\_t::string:

{

if (JSON\_HEDLEY\_UNLIKELY(not first.m\_it.primitive\_iterator.is\_begin()

or not last.m\_it.primitive\_iterator.is\_end()))

{

JSON\_THROW(invalid\_iterator::create(204, "iterators out of range"));

}

break;

}

default:

break;

}

switch (m\_type)

{

case value\_t::number\_integer:

{

m\_value.number\_integer = first.m\_object->m\_value.number\_integer;

break;

}

case value\_t::number\_unsigned:

{

m\_value.number\_unsigned = first.m\_object->m\_value.number\_unsigned;

break;

}

case value\_t::number\_float:

{

m\_value.number\_float = first.m\_object->m\_value.number\_float;

break;

}

case value\_t::boolean:

{

m\_value.boolean = first.m\_object->m\_value.boolean;

break;

}

case value\_t::string:

{

m\_value = \*first.m\_object->m\_value.string;

break;

}

case value\_t::object:

{

m\_value.object = create<object\_t>(first.m\_it.object\_iterator,

last.m\_it.object\_iterator);

break;

}

case value\_t::array:

{

m\_value.array = create<array\_t>(first.m\_it.array\_iterator,

last.m\_it.array\_iterator);

break;

}

default:

JSON\_THROW(invalid\_iterator::create(206, "cannot construct with iterators from " +

std::string(first.m\_object->type\_name())));

}

assert\_invariant();

}

///////////////////////////////////////

// other constructors and destructor //

///////////////////////////////////////

/// @private

basic\_json(const detail::json\_ref<basic\_json>& ref)

: basic\_json(ref.moved\_or\_copied())

{}

/\*!

@brief copy constructor

Creates a copy of a given JSON value.

@param[in] other the JSON value to copy

@post `\*this == other`

@complexity Linear in the size of @a other.

@exceptionsafety Strong guarantee: if an exception is thrown, there are no

changes to any JSON value.

@requirement This function helps `basic\_json` satisfying the

[Container](https://en.cppreference.com/w/cpp/named\_req/Container)

requirements:

- The complexity is linear.

- As postcondition, it holds: `other == basic\_json(other)`.

@liveexample{The following code shows an example for the copy

constructor.,basic\_json\_\_basic\_json}

@since version 1.0.0

\*/

basic\_json(const basic\_json& other)

: m\_type(other.m\_type)

{

// check of passed value is valid

other.assert\_invariant();

switch (m\_type)

{

case value\_t::object:

{

m\_value = \*other.m\_value.object;

break;

}

case value\_t::array:

{

m\_value = \*other.m\_value.array;

break;

}

case value\_t::string:

{

m\_value = \*other.m\_value.string;

break;

}

case value\_t::boolean:

{

m\_value = other.m\_value.boolean;

break;

}

case value\_t::number\_integer:

{

m\_value = other.m\_value.number\_integer;

break;

}

case value\_t::number\_unsigned:

{

m\_value = other.m\_value.number\_unsigned;

break;

}

case value\_t::number\_float:

{

m\_value = other.m\_value.number\_float;

break;

}

default:

break;

}

assert\_invariant();

}

/\*!

@brief move constructor

Move constructor. Constructs a JSON value with the contents of the given

value @a other using move semantics. It "steals" the resources from @a

other and leaves it as JSON null value.

@param[in,out] other value to move to this object

@post `\*this` has the same value as @a other before the call.

@post @a other is a JSON null value.

@complexity Constant.

@exceptionsafety No-throw guarantee: this constructor never throws

exceptions.

@requirement This function helps `basic\_json` satisfying the

[MoveConstructible](https://en.cppreference.com/w/cpp/named\_req/MoveConstructible)

requirements.

@liveexample{The code below shows the move constructor explicitly called

via std::move.,basic\_json\_\_moveconstructor}

@since version 1.0.0

\*/

basic\_json(basic\_json&& other) noexcept

: m\_type(std::move(other.m\_type)),

m\_value(std::move(other.m\_value))

{

// check that passed value is valid

other.assert\_invariant();

// invalidate payload

other.m\_type = value\_t::null;

other.m\_value = {};

assert\_invariant();

}

/\*!

@brief copy assignment

Copy assignment operator. Copies a JSON value via the "copy and swap"

strategy: It is expressed in terms of the copy constructor, destructor,

and the `swap()` member function.

@param[in] other value to copy from

@complexity Linear.

@requirement This function helps `basic\_json` satisfying the

[Container](https://en.cppreference.com/w/cpp/named\_req/Container)

requirements:

- The complexity is linear.

@liveexample{The code below shows and example for the copy assignment. It

creates a copy of value `a` which is then swapped with `b`. Finally\, the

copy of `a` (which is the null value after the swap) is

destroyed.,basic\_json\_\_copyassignment}

@since version 1.0.0

\*/

basic\_json& operator=(basic\_json other) noexcept (

std::is\_nothrow\_move\_constructible<value\_t>::value and

std::is\_nothrow\_move\_assignable<value\_t>::value and

std::is\_nothrow\_move\_constructible<json\_value>::value and

std::is\_nothrow\_move\_assignable<json\_value>::value

)

{

// check that passed value is valid

other.assert\_invariant();

using std::swap;

swap(m\_type, other.m\_type);

swap(m\_value, other.m\_value);

assert\_invariant();

return \*this;

}

/\*!

@brief destructor

Destroys the JSON value and frees all allocated memory.

@complexity Linear.

@requirement This function helps `basic\_json` satisfying the

[Container](https://en.cppreference.com/w/cpp/named\_req/Container)

requirements:

- The complexity is linear.

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

@since version 1.0.0

\*/

~basic\_json() noexcept

{

assert\_invariant();

m\_value.destroy(m\_type);

}

/// @}

public:

///////////////////////

// object inspection //

///////////////////////

/// @name object inspection

/// Functions to inspect the type of a JSON value.

/// @{

/\*!

@brief serialization

Serialization function for JSON values. The function tries to mimic

Python's `json.dumps()` function, and currently supports its @a indent

and @a ensure\_ascii parameters.

@param[in] indent If indent is nonnegative, then array elements and object

members will be pretty-printed with that indent level. An indent level of

`0` will only insert newlines. `-1` (the default) selects the most compact

representation.

@param[in] indent\_char The character to use for indentation if @a indent is

greater than `0`. The default is ` ` (space).

@param[in] ensure\_ascii If @a ensure\_ascii is true, all non-ASCII characters

in the output are escaped with `\uXXXX` sequences, and the result consists

of ASCII characters only.

@param[in] error\_handler how to react on decoding errors; there are three

possible values: `strict` (throws and exception in case a decoding error

occurs; default), `replace` (replace invalid UTF-8 sequences with U+FFFD),

and `ignore` (ignore invalid UTF-8 sequences during serialization).

@return string containing the serialization of the JSON value

@throw type\_error.316 if a string stored inside the JSON value is not

UTF-8 encoded

@complexity Linear.

@exceptionsafety Strong guarantee: if an exception is thrown, there are no

changes in the JSON value.

@liveexample{The following example shows the effect of different @a indent\,

@a indent\_char\, and @a ensure\_ascii parameters to the result of the

serialization.,dump}

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

@since version 1.0.0; indentation character @a indent\_char, option

@a ensure\_ascii and exceptions added in version 3.0.0; error

handlers added in version 3.4.0.

\*/

string\_t dump(const int indent = -1,

const char indent\_char = ' ',

const bool ensure\_ascii = false,

const error\_handler\_t error\_handler = error\_handler\_t::strict) const

{

string\_t result;

serializer s(detail::output\_adapter<char, string\_t>(result), indent\_char, error\_handler);

if (indent >= 0)

{

s.dump(\*this, true, ensure\_ascii, static\_cast<unsigned int>(indent));

}

else

{

s.dump(\*this, false, ensure\_ascii, 0);

}

return result;

}

/\*!

@brief return the type of the JSON value (explicit)

Return the type of the JSON value as a value from the @ref value\_t

enumeration.

@return the type of the JSON value

Value type | return value

------------------------- | -------------------------

null | value\_t::null

boolean | value\_t::boolean

string | value\_t::string

number (integer) | value\_t::number\_integer

number (unsigned integer) | value\_t::number\_unsigned

number (floating-point) | value\_t::number\_float

object | value\_t::object

array | value\_t::array

discarded | value\_t::discarded

@complexity Constant.

@exceptionsafety No-throw guarantee: this member function never throws

exceptions.

@liveexample{The following code exemplifies `type()` for all JSON

types.,type}

@sa @ref operator value\_t() -- return the type of the JSON value (implicit)

@sa @ref type\_name() -- return the type as string

@since version 1.0.0

\*/

constexpr value\_t type() const noexcept

{

return m\_type;

}

/\*!

@brief return whether type is primitive

This function returns true if and only if the JSON type is primitive

(string, number, boolean, or null).

@return `true` if type is primitive (string, number, boolean, or null),

`false` otherwise.

@complexity Constant.

@exceptionsafety No-throw guarantee: this member function never throws

exceptions.

@liveexample{The following code exemplifies `is\_primitive()` for all JSON

types.,is\_primitive}

@sa @ref is\_structured() -- returns whether JSON value is structured

@sa @ref is\_null() -- returns whether JSON value is `null`

@sa @ref is\_string() -- returns whether JSON value is a string

@sa @ref is\_boolean() -- returns whether JSON value is a boolean

@sa @ref is\_number() -- returns whether JSON value is a number

@since version 1.0.0

\*/

constexpr bool is\_primitive() const noexcept

{

return is\_null() or is\_string() or is\_boolean() or is\_number();

}

/\*!

@brief return whether type is structured

This function returns true if and only if the JSON type is structured

(array or object).

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

@complexity Constant.

@exceptionsafety No-throw guarantee: this member function never throws

exceptions.

@liveexample{The following code exemplifies `is\_structured()` for all JSON

types.,is\_structured}

@sa @ref is\_primitive() -- returns whether value is primitive

@sa @ref is\_array() -- returns whether value is an array

@sa @ref is\_object() -- returns whether value is an object

@since version 1.0.0

\*/

constexpr bool is\_structured() const noexcept

{

return is\_array() or is\_object();

}

/\*!

@brief return whether value is null

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

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

@complexity Constant.

@exceptionsafety No-throw guarantee: this member function never throws

exceptions.

@liveexample{The following code exemplifies `is\_null()` for all JSON

types.,is\_null}

@since version 1.0.0

\*/

constexpr bool is\_null() const noexcept

{

return m\_type == value\_t::null;

}

/\*!

@brief return whether value is a boolean

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

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

@complexity Constant.

@exceptionsafety No-throw guarantee: this member function never throws

exceptions.

@liveexample{The following code exemplifies `is\_boolean()` for all JSON

types.,is\_boolean}

@since version 1.0.0

\*/

constexpr bool is\_boolean() const noexcept

{

return m\_type == value\_t::boolean;

}

/\*!

@brief return whether value is a number

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

includes both integer (signed and unsigned) and floating-point values.

@return `true` if type is number (regardless whether integer, unsigned

integer or floating-type), `false` otherwise.

@complexity Constant.

@exceptionsafety No-throw guarantee: this member function never throws

exceptions.

@liveexample{The following code exemplifies `is\_number()` for all JSON

types.,is\_number}

@sa @ref is\_number\_integer() -- check if value is an integer or unsigned

integer number

@sa @ref is\_number\_unsigned() -- check if value is an unsigned integer

number

@sa @ref is\_number\_float() -- check if value is a floating-point number

@since version 1.0.0

\*/

constexpr bool is\_number() const noexcept

{

return is\_number\_integer() or is\_number\_float();

}

/\*!

@brief return whether value is an integer number

This function returns true if and only if the JSON value is a signed or

unsigned integer number. This excludes floating-point values.

@return `true` if type is an integer or unsigned integer number, `false`

otherwise.

@complexity Constant.

@exceptionsafety No-throw guarantee: this member function never throws

exceptions.

@liveexample{The following code exemplifies `is\_number\_integer()` for all

JSON types.,is\_number\_integer}

@sa @ref is\_number() -- check if value is a number

@sa @ref is\_number\_unsigned() -- check if value is an unsigned integer

number

@sa @ref is\_number\_float() -- check if value is a floating-point number

@since version 1.0.0

\*/

constexpr bool is\_number\_integer() const noexcept

{

return m\_type == value\_t::number\_integer or m\_type == value\_t::number\_unsigned;

}

/\*!

@brief return whether value is an unsigned integer number

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

integer number. This excludes floating-point and signed integer values.

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

@complexity Constant.

@exceptionsafety No-throw guarantee: this member function never throws

exceptions.

@liveexample{The following code exemplifies `is\_number\_unsigned()` for all

JSON types.,is\_number\_unsigned}

@sa @ref is\_number() -- check if value is a number

@sa @ref is\_number\_integer() -- check if value is an integer or unsigned

integer number

@sa @ref is\_number\_float() -- check if value is a floating-point number

@since version 2.0.0

\*/

constexpr bool is\_number\_unsigned() const noexcept

{

return m\_type == value\_t::number\_unsigned;

}

/\*!

@brief return whether value is a floating-point number

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

floating-point number. This excludes signed and unsigned integer values.

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

@complexity Constant.

@exceptionsafety No-throw guarantee: this member function never throws

exceptions.

@liveexample{The following code exemplifies `is\_number\_float()` for all

JSON types.,is\_number\_float}

@sa @ref is\_number() -- check if value is number

@sa @ref is\_number\_integer() -- check if value is an integer number

@sa @ref is\_number\_unsigned() -- check if value is an unsigned integer

number

@since version 1.0.0

\*/

constexpr bool is\_number\_float() const noexcept

{

return m\_type == value\_t::number\_float;

}

/\*!

@brief return whether value is an object

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

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

@complexity Constant.

@exceptionsafety No-throw guarantee: this member function never throws

exceptions.

@liveexample{The following code exemplifies `is\_object()` for all JSON

types.,is\_object}

@since version 1.0.0

\*/

constexpr bool is\_object() const noexcept

{

return m\_type == value\_t::object;

}

/\*!

@brief return whether value is an array

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

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

@complexity Constant.

@exceptionsafety No-throw guarantee: this member function never throws

exceptions.

@liveexample{The following code exemplifies `is\_array()` for all JSON

types.,is\_array}

@since version 1.0.0

\*/

constexpr bool is\_array() const noexcept

{

return m\_type == value\_t::array;

}

/\*!

@brief return whether value is a string

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

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

@complexity Constant.

@exceptionsafety No-throw guarantee: this member function never throws

exceptions.

@liveexample{The following code exemplifies `is\_string()` for all JSON

types.,is\_string}

@since version 1.0.0

\*/

constexpr bool is\_string() const noexcept

{

return m\_type == value\_t::string;

}

/\*!

@brief return whether value is discarded

This function returns true if and only if the JSON value was discarded

during parsing with a callback function (see @ref parser\_callback\_t).

@note This function will always be `false` for JSON values after parsing.

That is, discarded values can only occur during parsing, but will be

removed when inside a structured value or replaced by null in other cases.

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

@complexity Constant.

@exceptionsafety No-throw guarantee: this member function never throws

exceptions.

@liveexample{The following code exemplifies `is\_discarded()` for all JSON

types.,is\_discarded}

@since version 1.0.0

\*/

constexpr bool is\_discarded() const noexcept

{

return m\_type == value\_t::discarded;

}

/\*!

@brief return the type of the JSON value (implicit)

Implicitly return the type of the JSON value as a value from the @ref

value\_t enumeration.

@return the type of the JSON value

@complexity Constant.

@exceptionsafety No-throw guarantee: this member function never throws

exceptions.

@liveexample{The following code exemplifies the @ref value\_t operator for

all JSON types.,operator\_\_value\_t}

@sa @ref type() -- return the type of the JSON value (explicit)

@sa @ref type\_name() -- return the type as string

@since version 1.0.0

\*/

constexpr operator value\_t() const noexcept

{

return m\_type;

}

/// @}

private:

//////////////////

// value access //

//////////////////

/// get a boolean (explicit)

boolean\_t get\_impl(boolean\_t\* /\*unused\*/) const

{

if (JSON\_HEDLEY\_LIKELY(is\_boolean()))

{

return m\_value.boolean;

}

JSON\_THROW(type\_error::create(302, "type must be boolean, but is " + std::string(type\_name())));

}

/// get a pointer to the value (object)

object\_t\* get\_impl\_ptr(object\_t\* /\*unused\*/) noexcept

{

return is\_object() ? m\_value.object : nullptr;

}

/// get a pointer to the value (object)

constexpr const object\_t\* get\_impl\_ptr(const object\_t\* /\*unused\*/) const noexcept

{

return is\_object() ? m\_value.object : nullptr;

}

/// get a pointer to the value (array)

array\_t\* get\_impl\_ptr(array\_t\* /\*unused\*/) noexcept

{

return is\_array() ? m\_value.array : nullptr;

}

/// get a pointer to the value (array)

constexpr const array\_t\* get\_impl\_ptr(const array\_t\* /\*unused\*/) const noexcept

{

return is\_array() ? m\_value.array : nullptr;

}

/// get a pointer to the value (string)

string\_t\* get\_impl\_ptr(string\_t\* /\*unused\*/) noexcept

{

return is\_string() ? m\_value.string : nullptr;

}

/// get a pointer to the value (string)

constexpr const string\_t\* get\_impl\_ptr(const string\_t\* /\*unused\*/) const noexcept

{

return is\_string() ? m\_value.string : nullptr;

}

/// get a pointer to the value (boolean)

boolean\_t\* get\_impl\_ptr(boolean\_t\* /\*unused\*/) noexcept

{

return is\_boolean() ? &m\_value.boolean : nullptr;

}

/// get a pointer to the value (boolean)

constexpr const boolean\_t\* get\_impl\_ptr(const boolean\_t\* /\*unused\*/) const noexcept

{

return is\_boolean() ? &m\_value.boolean : nullptr;

}

/// get a pointer to the value (integer number)

number\_integer\_t\* get\_impl\_ptr(number\_integer\_t\* /\*unused\*/) noexcept

{

return is\_number\_integer() ? &m\_value.number\_integer : nullptr;

}

/// get a pointer to the value (integer number)

constexpr const number\_integer\_t\* get\_impl\_ptr(const number\_integer\_t\* /\*unused\*/) const noexcept

{

return is\_number\_integer() ? &m\_value.number\_integer : nullptr;

}

/// get a pointer to the value (unsigned number)

number\_unsigned\_t\* get\_impl\_ptr(number\_unsigned\_t\* /\*unused\*/) noexcept

{

return is\_number\_unsigned() ? &m\_value.number\_unsigned : nullptr;

}

/// get a pointer to the value (unsigned number)

constexpr const number\_unsigned\_t\* get\_impl\_ptr(const number\_unsigned\_t\* /\*unused\*/) const noexcept

{

return is\_number\_unsigned() ? &m\_value.number\_unsigned : nullptr;

}

/// get a pointer to the value (floating-point number)

number\_float\_t\* get\_impl\_ptr(number\_float\_t\* /\*unused\*/) noexcept

{

return is\_number\_float() ? &m\_value.number\_float : nullptr;

}

/// get a pointer to the value (floating-point number)

constexpr const number\_float\_t\* get\_impl\_ptr(const number\_float\_t\* /\*unused\*/) const noexcept

{

return is\_number\_float() ? &m\_value.number\_float : nullptr;

}

/\*!

@brief helper function to implement get\_ref()

This function helps to implement get\_ref() without code duplication for

const and non-const overloads

@tparam ThisType will be deduced as `basic\_json` or `const basic\_json`

@throw type\_error.303 if ReferenceType does not match underlying value

type of the current JSON

\*/

template<typename ReferenceType, typename ThisType>

static ReferenceType get\_ref\_impl(ThisType& obj)

{

// delegate the call to get\_ptr<>()

auto ptr = obj.template get\_ptr<typename std::add\_pointer<ReferenceType>::type>();

if (JSON\_HEDLEY\_LIKELY(ptr != nullptr))

{

return \*ptr;

}

JSON\_THROW(type\_error::create(303, "incompatible ReferenceType for get\_ref, actual type is " + std::string(obj.type\_name())));

}

public:

/// @name value access

/// Direct access to the stored value of a JSON value.

/// @{

/\*!

@brief get special-case overload

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

identity method

@tparam BasicJsonType == @ref basic\_json

@return a copy of \*this

@complexity Constant.

@since version 2.1.0

\*/

template<typename BasicJsonType, detail::enable\_if\_t<

std::is\_same<typename std::remove\_const<BasicJsonType>::type, basic\_json\_t>::value,

int> = 0>

basic\_json get() const

{

return \*this;

}

/\*!

@brief get special-case overload

This overloads converts the current @ref basic\_json in a different

@ref basic\_json type

@tparam BasicJsonType == @ref basic\_json

@return a copy of \*this, converted into @tparam BasicJsonType

@complexity Depending on the implementation of the called `from\_json()`

method.

@since version 3.2.0

\*/

template<typename BasicJsonType, detail::enable\_if\_t<

not std::is\_same<BasicJsonType, basic\_json>::value and

detail::is\_basic\_json<BasicJsonType>::value, int> = 0>

BasicJsonType get() const

{

return \*this;

}

/\*!

@brief get a value (explicit)

Explicit type conversion between the JSON value and a compatible value

which is [CopyConstructible](https://en.cppreference.com/w/cpp/named\_req/CopyConstructible)

and [DefaultConstructible](https://en.cppreference.com/w/cpp/named\_req/DefaultConstructible).

The value is converted by calling the @ref json\_serializer<ValueType>

`from\_json()` method.

The function is equivalent to executing

@code {.cpp}

ValueType ret;

JSONSerializer<ValueType>::from\_json(\*this, ret);

return ret;

@endcode

This overloads is chosen if:

- @a ValueType is not @ref basic\_json,

- @ref json\_serializer<ValueType> has a `from\_json()` method of the form

`void from\_json(const basic\_json&, ValueType&)`, and

- @ref json\_serializer<ValueType> does not have a `from\_json()` method of

the form `ValueType from\_json(const basic\_json&)`

@tparam ValueTypeCV the provided value type

@tparam ValueType the returned value type

@return copy of the JSON value, converted to @a ValueType

@throw what @ref json\_serializer<ValueType> `from\_json()` method throws

@liveexample{The example below shows several conversions from JSON values

to other types. There a few things to note: (1) Floating-point numbers can

be converted to integers\, (2) A JSON array can be converted to a standard

`std::vector<short>`\, (3) A JSON object can be converted to C++

associative containers such as `std::unordered\_map<std::string\,

json>`.,get\_\_ValueType\_const}

@since version 2.1.0

\*/

template<typename ValueTypeCV, typename ValueType = detail::uncvref\_t<ValueTypeCV>,

detail::enable\_if\_t <

not detail::is\_basic\_json<ValueType>::value and

detail::has\_from\_json<basic\_json\_t, ValueType>::value and

not detail::has\_non\_default\_from\_json<basic\_json\_t, ValueType>::value,

int> = 0>

ValueType get() const noexcept(noexcept(

JSONSerializer<ValueType>::from\_json(std::declval<const basic\_json\_t&>(), std::declval<ValueType&>())))

{

// we cannot static\_assert on ValueTypeCV being non-const, because

// there is support for get<const basic\_json\_t>(), which is why we

// still need the uncvref

static\_assert(not std::is\_reference<ValueTypeCV>::value,

"get() cannot be used with reference types, you might want to use get\_ref()");

static\_assert(std::is\_default\_constructible<ValueType>::value,

"types must be DefaultConstructible when used with get()");

ValueType ret;

JSONSerializer<ValueType>::from\_json(\*this, ret);

return ret;

}

/\*!

@brief get a value (explicit); special case

Explicit type conversion between the JSON value and a compatible value

which is \*\*not\*\* [CopyConstructible](https://en.cppreference.com/w/cpp/named\_req/CopyConstructible)

and \*\*not\*\* [DefaultConstructible](https://en.cppreference.com/w/cpp/named\_req/DefaultConstructible).

The value is converted by calling the @ref json\_serializer<ValueType>

`from\_json()` method.

The function is equivalent to executing

@code {.cpp}

return JSONSerializer<ValueTypeCV>::from\_json(\*this);

@endcode

This overloads is chosen if:

- @a ValueType is not @ref basic\_json and

- @ref json\_serializer<ValueType> has a `from\_json()` method of the form

`ValueType from\_json(const basic\_json&)`

@note If @ref json\_serializer<ValueType> has both overloads of

`from\_json()`, this one is chosen.

@tparam ValueTypeCV the provided value type

@tparam ValueType the returned value type

@return copy of the JSON value, converted to @a ValueType

@throw what @ref json\_serializer<ValueType> `from\_json()` method throws

@since version 2.1.0

\*/

template<typename ValueTypeCV, typename ValueType = detail::uncvref\_t<ValueTypeCV>,

detail::enable\_if\_t<not std::is\_same<basic\_json\_t, ValueType>::value and

detail::has\_non\_default\_from\_json<basic\_json\_t, ValueType>::value,

int> = 0>

ValueType get() const noexcept(noexcept(

JSONSerializer<ValueType>::from\_json(std::declval<const basic\_json\_t&>())))

{

static\_assert(not std::is\_reference<ValueTypeCV>::value,

"get() cannot be used with reference types, you might want to use get\_ref()");

return JSONSerializer<ValueType>::from\_json(\*this);

}

/\*!

@brief get a value (explicit)

Explicit type conversion between the JSON value and a compatible value.

The value is filled into the input parameter by calling the @ref json\_serializer<ValueType>

`from\_json()` method.

The function is equivalent to executing

@code {.cpp}

ValueType v;

JSONSerializer<ValueType>::from\_json(\*this, v);

@endcode

This overloads is chosen if:

- @a ValueType is not @ref basic\_json,

- @ref json\_serializer<ValueType> has a `from\_json()` method of the form

`void from\_json(const basic\_json&, ValueType&)`, and

@tparam ValueType the input parameter type.

@return the input parameter, allowing chaining calls.

@throw what @ref json\_serializer<ValueType> `from\_json()` method throws

@liveexample{The example below shows several conversions from JSON values

to other types. There a few things to note: (1) Floating-point numbers can

be converted to integers\, (2) A JSON array can be converted to a standard

`std::vector<short>`\, (3) A JSON object can be converted to C++

associative containers such as `std::unordered\_map<std::string\,

json>`.,get\_to}

@since version 3.3.0

\*/

template<typename ValueType,

detail::enable\_if\_t <

not detail::is\_basic\_json<ValueType>::value and

detail::has\_from\_json<basic\_json\_t, ValueType>::value,

int> = 0>

ValueType & get\_to(ValueType& v) const noexcept(noexcept(

JSONSerializer<ValueType>::from\_json(std::declval<const basic\_json\_t&>(), v)))

{

JSONSerializer<ValueType>::from\_json(\*this, v);

return v;

}

template <

typename T, std::size\_t N,

typename Array = T (&)[N],

detail::enable\_if\_t <

detail::has\_from\_json<basic\_json\_t, Array>::value, int > = 0 >

Array get\_to(T (&v)[N]) const

noexcept(noexcept(JSONSerializer<Array>::from\_json(

std::declval<const basic\_json\_t&>(), v)))

{

JSONSerializer<Array>::from\_json(\*this, v);

return v;

}

/\*!

@brief get a pointer value (implicit)

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

made.

@warning Writing data to the pointee of the result yields an undefined

state.

@tparam PointerType pointer type; must be a pointer to @ref array\_t, @ref

object\_t, @ref string\_t, @ref boolean\_t, @ref number\_integer\_t,

@ref number\_unsigned\_t, or @ref number\_float\_t. Enforced by a static

assertion.

@return pointer to the internally stored JSON value if the requested

pointer type @a PointerType fits to the JSON value; `nullptr` otherwise

@complexity Constant.

@liveexample{The example below shows how pointers to internal values of a

JSON value can be requested. Note that no type conversions are made and a

`nullptr` is returned if the value and the requested pointer type does not

match.,get\_ptr}

@since version 1.0.0

\*/

template<typename PointerType, typename std::enable\_if<

std::is\_pointer<PointerType>::value, int>::type = 0>

auto get\_ptr() noexcept -> decltype(std::declval<basic\_json\_t&>().get\_impl\_ptr(std::declval<PointerType>()))

{

// delegate the call to get\_impl\_ptr<>()

return get\_impl\_ptr(static\_cast<PointerType>(nullptr));

}

/\*!

@brief get a pointer value (implicit)

@copydoc get\_ptr()

\*/

template<typename PointerType, typename std::enable\_if<

std::is\_pointer<PointerType>::value and

std::is\_const<typename std::remove\_pointer<PointerType>::type>::value, int>::type = 0>

constexpr auto get\_ptr() const noexcept -> decltype(std::declval<const basic\_json\_t&>().get\_impl\_ptr(std::declval<PointerType>()))

{

// delegate the call to get\_impl\_ptr<>() const

return get\_impl\_ptr(static\_cast<PointerType>(nullptr));

}

/\*!

@brief get a pointer value (explicit)

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

made.

@warning The pointer becomes invalid if the underlying JSON object

changes.

@tparam PointerType pointer type; must be a pointer to @ref array\_t, @ref

object\_t, @ref string\_t, @ref boolean\_t, @ref number\_integer\_t,

@ref number\_unsigned\_t, or @ref number\_float\_t.

@return pointer to the internally stored JSON value if the requested

pointer type @a PointerType fits to the JSON value; `nullptr` otherwise

@complexity Constant.

@liveexample{The example below shows how pointers to internal values of a

JSON value can be requested. Note that no type conversions are made and a

`nullptr` is returned if the value and the requested pointer type does not

match.,get\_\_PointerType}

@sa @ref get\_ptr() for explicit pointer-member access

@since version 1.0.0

\*/

template<typename PointerType, typename std::enable\_if<

std::is\_pointer<PointerType>::value, int>::type = 0>

auto get() noexcept -> decltype(std::declval<basic\_json\_t&>().template get\_ptr<PointerType>())

{

// delegate the call to get\_ptr

return get\_ptr<PointerType>();

}

/\*!

@brief get a pointer value (explicit)

@copydoc get()

\*/

template<typename PointerType, typename std::enable\_if<

std::is\_pointer<PointerType>::value, int>::type = 0>

constexpr auto get() const noexcept -> decltype(std::declval<const basic\_json\_t&>().template get\_ptr<PointerType>())

{

// delegate the call to get\_ptr

return get\_ptr<PointerType>();

}

/\*!

@brief get a reference value (implicit)

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

are made.

@warning Writing data to the referee of the result yields an undefined

state.

@tparam ReferenceType reference type; must be a reference to @ref array\_t,

@ref object\_t, @ref string\_t, @ref boolean\_t, @ref number\_integer\_t, or

@ref number\_float\_t. Enforced by static assertion.

@return reference to the internally stored JSON value if the requested

reference type @a ReferenceType fits to the JSON value; throws

type\_error.303 otherwise

@throw type\_error.303 in case passed type @a ReferenceType is incompatible

with the stored JSON value; see example below

@complexity Constant.

@liveexample{The example shows several calls to `get\_ref()`.,get\_ref}

@since version 1.1.0

\*/

template<typename ReferenceType, typename std::enable\_if<

std::is\_reference<ReferenceType>::value, int>::type = 0>

ReferenceType get\_ref()

{

// delegate call to get\_ref\_impl

return get\_ref\_impl<ReferenceType>(\*this);

}

/\*!

@brief get a reference value (implicit)

@copydoc get\_ref()

\*/

template<typename ReferenceType, typename std::enable\_if<

std::is\_reference<ReferenceType>::value and

std::is\_const<typename std::remove\_reference<ReferenceType>::type>::value, int>::type = 0>

ReferenceType get\_ref() const

{

// delegate call to get\_ref\_impl

return get\_ref\_impl<ReferenceType>(\*this);

}

/\*!

@brief get a value (implicit)

Implicit type conversion between the JSON value and a compatible value.

The call is realized by calling @ref get() const.

@tparam ValueType non-pointer type compatible to the JSON value, for

instance `int` for JSON integer numbers, `bool` for JSON booleans, or

`std::vector` types for JSON arrays. The character type of @ref string\_t

as well as an initializer list of this type is excluded to avoid

ambiguities as these types implicitly convert to `std::string`.

@return copy of the JSON value, converted to type @a ValueType

@throw type\_error.302 in case passed type @a ValueType is incompatible

to the JSON value type (e.g., the JSON value is of type boolean, but a

string is requested); see example below

@complexity Linear in the size of the JSON value.

@liveexample{The example below shows several conversions from JSON values

to other types. There a few things to note: (1) Floating-point numbers can

be converted to integers\, (2) A JSON array can be converted to a standard

`std::vector<short>`\, (3) A JSON object can be converted to C++

associative containers such as `std::unordered\_map<std::string\,

json>`.,operator\_\_ValueType}

@since version 1.0.0

\*/

template < typename ValueType, typename std::enable\_if <

not std::is\_pointer<ValueType>::value and

not std::is\_same<ValueType, detail::json\_ref<basic\_json>>::value and

not std::is\_same<ValueType, typename string\_t::value\_type>::value and

not detail::is\_basic\_json<ValueType>::value

#ifndef \_MSC\_VER // fix for issue #167 operator<< ambiguity under VS2015

and not std::is\_same<ValueType, std::initializer\_list<typename string\_t::value\_type>>::value

#if defined(JSON\_HAS\_CPP\_17) && (defined(\_\_GNUC\_\_) || (defined(\_MSC\_VER) and \_MSC\_VER <= 1914))

and not std::is\_same<ValueType, typename std::string\_view>::value

#endif

#endif

and detail::is\_detected<detail::get\_template\_function, const basic\_json\_t&, ValueType>::value

, int >::type = 0 >

operator ValueType() const

{

// delegate the call to get<>() const

return get<ValueType>();

}

/// @}

////////////////////

// element access //

////////////////////

/// @name element access

/// Access to the JSON value.

/// @{

/\*!

@brief access specified array element with bounds checking

Returns a reference to the element at specified location @a idx, with

bounds checking.

@param[in] idx index of the element to access

@return reference to the element at index @a idx

@throw type\_error.304 if the JSON value is not an array; in this case,

calling `at` with an index makes no sense. See example below.

@throw out\_of\_range.401 if the index @a idx is out of range of the array;

that is, `idx >= size()`. See example below.

@exceptionsafety Strong guarantee: if an exception is thrown, there are no

changes in the JSON value.

@complexity Constant.

@since version 1.0.0

@liveexample{The example below shows how array elements can be read and

written using `at()`. It also demonstrates the different exceptions that

can be thrown.,at\_\_size\_type}

\*/

reference at(size\_type idx)

{

// at only works for arrays

if (JSON\_HEDLEY\_LIKELY(is\_array()))

{

JSON\_TRY

{

return m\_value.array->at(idx);

}

JSON\_CATCH (std::out\_of\_range&)

{

// create better exception explanation

JSON\_THROW(out\_of\_range::create(401, "array index " + std::to\_string(idx) + " is out of range"));

}

}

else

{

JSON\_THROW(type\_error::create(304, "cannot use at() with " + std::string(type\_name())));

}

}

/\*!

@brief access specified array element with bounds checking

Returns a const reference to the element at specified location @a idx,

with bounds checking.

@param[in] idx index of the element to access

@return const reference to the element at index @a idx

@throw type\_error.304 if the JSON value is not an array; in this case,

calling `at` with an index makes no sense. See example below.

@throw out\_of\_range.401 if the index @a idx is out of range of the array;

that is, `idx >= size()`. See example below.

@exceptionsafety Strong guarantee: if an exception is thrown, there are no

changes in the JSON value.

@complexity Constant.

@since version 1.0.0

@liveexample{The example below shows how array elements can be read using

`at()`. It also demonstrates the different exceptions that can be thrown.,

at\_\_size\_type\_const}

\*/

const\_reference at(size\_type idx) const

{

// at only works for arrays

if (JSON\_HEDLEY\_LIKELY(is\_array()))

{

JSON\_TRY

{

return m\_value.array->at(idx);

}

JSON\_CATCH (std::out\_of\_range&)

{

// create better exception explanation

JSON\_THROW(out\_of\_range::create(401, "array index " + std::to\_string(idx) + " is out of range"));

}

}

else

{

JSON\_THROW(type\_error::create(304, "cannot use at() with " + std::string(type\_name())));

}

}

/\*!

@brief access specified object element with bounds checking

Returns a reference to the element at with specified key @a key, with

bounds checking.

@param[in] key key of the element to access

@return reference to the element at key @a key

@throw type\_error.304 if the JSON value is not an object; in this case,

calling `at` with a key makes no sense. See example below.

@throw out\_of\_range.403 if the key @a key is is not stored in the object;

that is, `find(key) == end()`. See example below.

@exceptionsafety Strong guarantee: if an exception is thrown, there are no

changes in the JSON value.

@complexity Logarithmic in the size of the container.

@sa @ref operator[](const typename object\_t::key\_type&) for unchecked

access by reference

@sa @ref value() for access by value with a default value

@since version 1.0.0

@liveexample{The example below shows how object elements can be read and

written using `at()`. It also demonstrates the different exceptions that

can be thrown.,at\_\_object\_t\_key\_type}

\*/

reference at(const typename object\_t::key\_type& key)

{

// at only works for objects

if (JSON\_HEDLEY\_LIKELY(is\_object()))

{

JSON\_TRY

{

return m\_value.object->at(key);

}

JSON\_CATCH (std::out\_of\_range&)

{

// create better exception explanation

JSON\_THROW(out\_of\_range::create(403, "key '" + key + "' not found"));

}

}

else

{

JSON\_THROW(type\_error::create(304, "cannot use at() with " + std::string(type\_name())));

}

}

/\*!

@brief access specified object element with bounds checking

Returns a const reference to the element at with specified key @a key,

with bounds checking.

@param[in] key key of the element to access

@return const reference to the element at key @a key

@throw type\_error.304 if the JSON value is not an object; in this case,

calling `at` with a key makes no sense. See example below.

@throw out\_of\_range.403 if the key @a key is is not stored in the object;

that is, `find(key) == end()`. See example below.

@exceptionsafety Strong guarantee: if an exception is thrown, there are no

changes in the JSON value.

@complexity Logarithmic in the size of the container.

@sa @ref operator[](const typename object\_t::key\_type&) for unchecked

access by reference

@sa @ref value() for access by value with a default value

@since version 1.0.0

@liveexample{The example below shows how object elements can be read using

`at()`. It also demonstrates the different exceptions that can be thrown.,

at\_\_object\_t\_key\_type\_const}

\*/

const\_reference at(const typename object\_t::key\_type& key) const

{

// at only works for objects

if (JSON\_HEDLEY\_LIKELY(is\_object()))

{

JSON\_TRY

{

return m\_value.object->at(key);

}

JSON\_CATCH (std::out\_of\_range&)

{

// create better exception explanation

JSON\_THROW(out\_of\_range::create(403, "key '" + key + "' not found"));

}

}

else

{

JSON\_THROW(type\_error::create(304, "cannot use at() with " + std::string(type\_name())));

}

}

/\*!

@brief access specified array element

Returns a reference to the element at specified location @a idx.

@note If @a idx is beyond the range of the array (i.e., `idx >= size()`),

then the array is silently filled up with `null` values to make `idx` a

valid reference to the last stored element.

@param[in] idx index of the element to access

@return reference to the element at index @a idx

@throw type\_error.305 if the JSON value is not an array or null; in that

cases, using the [] operator with an index makes no sense.

@complexity Constant if @a idx is in the range of the array. Otherwise

linear in `idx - size()`.

@liveexample{The example below shows how array elements can be read and

written using `[]` operator. Note the addition of `null`

values.,operatorarray\_\_size\_type}

@since version 1.0.0

\*/

reference operator[](size\_type idx)

{

// implicitly convert null value to an empty array

if (is\_null())

{

m\_type = value\_t::array;

m\_value.array = create<array\_t>();

assert\_invariant();

}

// operator[] only works for arrays

if (JSON\_HEDLEY\_LIKELY(is\_array()))

{

// fill up array with null values if given idx is outside range

if (idx >= m\_value.array->size())

{

m\_value.array->insert(m\_value.array->end(),

idx - m\_value.array->size() + 1,

basic\_json());

}

return m\_value.array->operator[](idx);

}

JSON\_THROW(type\_error::create(305, "cannot use operator[] with a numeric argument with " + std::string(type\_name())));

}

/\*!

@brief access specified array element

Returns a const reference to the element at specified location @a idx.

@param[in] idx index of the element to access

@return const reference to the element at index @a idx

@throw type\_error.305 if the JSON value is not an array; in that case,

using the [] operator with an index makes no sense.

@complexity Constant.

@liveexample{The example below shows how array elements can be read using

the `[]` operator.,operatorarray\_\_size\_type\_const}

@since version 1.0.0

\*/

const\_reference operator[](size\_type idx) const

{

// const operator[] only works for arrays

if (JSON\_HEDLEY\_LIKELY(is\_array()))

{

return m\_value.array->operator[](idx);

}

JSON\_THROW(type\_error::create(305, "cannot use operator[] with a numeric argument with " + std::string(type\_name())));

}

/\*!

@brief access specified object element

Returns a reference to the element at with specified key @a key.

@note If @a key is not found in the object, then it is silently added to

the object and filled with a `null` value to make `key` a valid reference.

In case the value was `null` before, it is converted to an object.

@param[in] key key of the element to access

@return reference to the element at key @a key

@throw type\_error.305 if the JSON value is not an object or null; in that

cases, using the [] operator with a key makes no sense.

@complexity Logarithmic in the size of the container.

@liveexample{The example below shows how object elements can be read and

written using the `[]` operator.,operatorarray\_\_key\_type}

@sa @ref at(const typename object\_t::key\_type&) for access by reference

with range checking

@sa @ref value() for access by value with a default value

@since version 1.0.0

\*/

reference operator[](const typename object\_t::key\_type& key)

{

// implicitly convert null value to an empty object

if (is\_null())

{

m\_type = value\_t::object;

m\_value.object = create<object\_t>();

assert\_invariant();

}

// operator[] only works for objects

if (JSON\_HEDLEY\_LIKELY(is\_object()))

{

return m\_value.object->operator[](key);

}

JSON\_THROW(type\_error::create(305, "cannot use operator[] with a string argument with " + std::string(type\_name())));

}

/\*!

@brief read-only access specified object element

Returns a const reference to the element at with specified key @a key. No

bounds checking is performed.

@warning If the element with key @a key does not exist, the behavior is

undefined.

@param[in] key key of the element to access

@return const reference to the element at key @a key

@pre The element with key @a key must exist. \*\*This precondition is

enforced with an assertion.\*\*

@throw type\_error.305 if the JSON value is not an object; in that case,

using the [] operator with a key makes no sense.

@complexity Logarithmic in the size of the container.

@liveexample{The example below shows how object elements can be read using

the `[]` operator.,operatorarray\_\_key\_type\_const}

@sa @ref at(const typename object\_t::key\_type&) for access by reference

with range checking

@sa @ref value() for access by value with a default value

@since version 1.0.0

\*/

const\_reference operator[](const typename object\_t::key\_type& key) const

{

// const operator[] only works for objects

if (JSON\_HEDLEY\_LIKELY(is\_object()))

{

assert(m\_value.object->find(key) != m\_value.object->end());

return m\_value.object->find(key)->second;

}

JSON\_THROW(type\_error::create(305, "cannot use operator[] with a string argument with " + std::string(type\_name())));

}

/\*!

@brief access specified object element

Returns a reference to the element at with specified key @a key.

@note If @a key is not found in the object, then it is silently added to

the object and filled with a `null` value to make `key` a valid reference.

In case the value was `null` before, it is converted to an object.

@param[in] key key of the element to access

@return reference to the element at key @a key

@throw type\_error.305 if the JSON value is not an object or null; in that

cases, using the [] operator with a key makes no sense.

@complexity Logarithmic in the size of the container.

@liveexample{The example below shows how object elements can be read and

written using the `[]` operator.,operatorarray\_\_key\_type}

@sa @ref at(const typename object\_t::key\_type&) for access by reference

with range checking

@sa @ref value() for access by value with a default value

@since version 1.1.0

\*/

template<typename T>

JSON\_HEDLEY\_NON\_NULL(2)

reference operator[](T\* key)

{

// implicitly convert null to object

if (is\_null())

{

m\_type = value\_t::object;

m\_value = value\_t::object;

assert\_invariant();

}

// at only works for objects

if (JSON\_HEDLEY\_LIKELY(is\_object()))

{

return m\_value.object->operator[](key);

}

JSON\_THROW(type\_error::create(305, "cannot use operator[] with a string argument with " + std::string(type\_name())));

}

/\*!

@brief read-only access specified object element

Returns a const reference to the element at with specified key @a key. No

bounds checking is performed.

@warning If the element with key @a key does not exist, the behavior is

undefined.

@param[in] key key of the element to access

@return const reference to the element at key @a key

@pre The element with key @a key must exist. \*\*This precondition is

enforced with an assertion.\*\*

@throw type\_error.305 if the JSON value is not an object; in that case,

using the [] operator with a key makes no sense.

@complexity Logarithmic in the size of the container.

@liveexample{The example below shows how object elements can be read using

the `[]` operator.,operatorarray\_\_key\_type\_const}

@sa @ref at(const typename object\_t::key\_type&) for access by reference

with range checking

@sa @ref value() for access by value with a default value

@since version 1.1.0

\*/

template<typename T>

JSON\_HEDLEY\_NON\_NULL(2)

const\_reference operator[](T\* key) const

{

// at only works for objects

if (JSON\_HEDLEY\_LIKELY(is\_object()))

{

assert(m\_value.object->find(key) != m\_value.object->end());

return m\_value.object->find(key)->second;

}

JSON\_THROW(type\_error::create(305, "cannot use operator[] with a string argument with " + std::string(type\_name())));

}

/\*!

@brief access specified object element with default value

Returns either a copy of an object's element at the specified key @a key

or a given default value if no element with key @a key exists.

The function is basically equivalent to executing

@code {.cpp}

try {

return at(key);

} catch(out\_of\_range) {

return default\_value;

}

@endcode

@note Unlike @ref at(const typename object\_t::key\_type&), this function

does not throw if the given key @a key was not found.

@note Unlike @ref operator[](const typename object\_t::key\_type& key), this

function does not implicitly add an element to the position defined by @a

key. This function is furthermore also applicable to const objects.

@param[in] key key of the element to access

@param[in] default\_value the value to return if @a key is not found

@tparam ValueType type compatible to JSON values, for instance `int` for

JSON integer numbers, `bool` for JSON booleans, or `std::vector` types for

JSON arrays. Note the type of the expected value at @a key and the default

value @a default\_value must be compatible.

@return copy of the element at key @a key or @a default\_value if @a key

is not found

@throw type\_error.302 if @a default\_value does not match the type of the

value at @a key

@throw type\_error.306 if the JSON value is not an object; in that case,

using `value()` with a key makes no sense.

@complexity Logarithmic in the size of the container.

@liveexample{The example below shows how object elements can be queried

with a default value.,basic\_json\_\_value}

@sa @ref at(const typename object\_t::key\_type&) for access by reference

with range checking

@sa @ref operator[](const typename object\_t::key\_type&) for unchecked

access by reference

@since version 1.0.0

\*/

template<class ValueType, typename std::enable\_if<

std::is\_convertible<basic\_json\_t, ValueType>::value, int>::type = 0>

ValueType value(const typename object\_t::key\_type& key, const ValueType& default\_value) const

{

// at only works for objects

if (JSON\_HEDLEY\_LIKELY(is\_object()))

{

// if key is found, return value and given default value otherwise

const auto it = find(key);

if (it != end())

{

return \*it;

}

return default\_value;

}

JSON\_THROW(type\_error::create(306, "cannot use value() with " + std::string(type\_name())));

}

/\*!

@brief overload for a default value of type const char\*

@copydoc basic\_json::value(const typename object\_t::key\_type&, const ValueType&) const

\*/

string\_t value(const typename object\_t::key\_type& key, const char\* default\_value) const

{

return value(key, string\_t(default\_value));

}

/\*!

@brief access specified object element via JSON Pointer with default value

Returns either a copy of an object's element at the specified key @a key

or a given default value if no element with key @a key exists.

The function is basically equivalent to executing

@code {.cpp}

try {

return at(ptr);

} catch(out\_of\_range) {

return default\_value;

}

@endcode

@note Unlike @ref at(const json\_pointer&), this function does not throw

if the given key @a key was not found.

@param[in] ptr a JSON pointer to the element to access

@param[in] default\_value the value to return if @a ptr found no value

@tparam ValueType type compatible to JSON values, for instance `int` for

JSON integer numbers, `bool` for JSON booleans, or `std::vector` types for

JSON arrays. Note the type of the expected value at @a key and the default

value @a default\_value must be compatible.

@return copy of the element at key @a key or @a default\_value if @a key

is not found

@throw type\_error.302 if @a default\_value does not match the type of the

value at @a ptr

@throw type\_error.306 if the JSON value is not an object; in that case,

using `value()` with a key makes no sense.

@complexity Logarithmic in the size of the container.

@liveexample{The example below shows how object elements can be queried

with a default value.,basic\_json\_\_value\_ptr}

@sa @ref operator[](const json\_pointer&) for unchecked access by reference

@since version 2.0.2

\*/

template<class ValueType, typename std::enable\_if<

std::is\_convertible<basic\_json\_t, ValueType>::value, int>::type = 0>

ValueType value(const json\_pointer& ptr, const ValueType& default\_value) const

{

// at only works for objects

if (JSON\_HEDLEY\_LIKELY(is\_object()))

{

// if pointer resolves a value, return it or use default value

JSON\_TRY

{

return ptr.get\_checked(this);

}

JSON\_INTERNAL\_CATCH (out\_of\_range&)

{

return default\_value;

}

}

JSON\_THROW(type\_error::create(306, "cannot use value() with " + std::string(type\_name())));

}

/\*!

@brief overload for a default value of type const char\*

@copydoc basic\_json::value(const json\_pointer&, ValueType) const

\*/

JSON\_HEDLEY\_NON\_NULL(3)

string\_t value(const json\_pointer& ptr, const char\* default\_value) const

{

return value(ptr, string\_t(default\_value));

}

/\*!

@brief access the first element

Returns a reference to the first element in the container. For a JSON

container `c`, the expression `c.front()` is equivalent to `\*c.begin()`.

@return In case of a structured type (array or object), a reference to the

first element is returned. In case of number, string, or boolean values, a

reference to the value is returned.

@complexity Constant.

@pre The JSON value must not be `null` (would throw `std::out\_of\_range`)

or an empty array or object (undefined behavior, \*\*guarded by

assertions\*\*).

@post The JSON value remains unchanged.

@throw invalid\_iterator.214 when called on `null` value

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

@sa @ref back() -- access the last element

@since version 1.0.0

\*/

reference front()

{

return \*begin();

}

/\*!

@copydoc basic\_json::front()

\*/

const\_reference front() const

{

return \*cbegin();

}

/\*!

@brief access the last element

Returns a reference to the last element in the container. For a JSON

container `c`, the expression `c.back()` is equivalent to

@code {.cpp}

auto tmp = c.end();

--tmp;

return \*tmp;

@endcode

@return In case of a structured type (array or object), a reference to the

last element is returned. In case of number, string, or boolean values, a

reference to the value is returned.

@complexity Constant.

@pre The JSON value must not be `null` (would throw `std::out\_of\_range`)

or an empty array or object (undefined behavior, \*\*guarded by

assertions\*\*).

@post The JSON value remains unchanged.

@throw invalid\_iterator.214 when called on a `null` value. See example

below.

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

@sa @ref front() -- access the first element

@since version 1.0.0

\*/

reference back()

{

auto tmp = end();

--tmp;

return \*tmp;

}

/\*!

@copydoc basic\_json::back()

\*/

const\_reference back() const

{

auto tmp = cend();

--tmp;

return \*tmp;

}

/\*!

@brief remove element given an iterator

Removes the element specified by iterator @a pos. The iterator @a pos must

be valid and dereferenceable. Thus the `end()` iterator (which is valid,

but is not dereferenceable) cannot be used as a value for @a pos.

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

will be `null`.

@param[in] pos iterator to the element to remove

@return Iterator following the last removed element. If the iterator @a

pos refers to the last element, the `end()` iterator is returned.

@tparam IteratorType an @ref iterator or @ref const\_iterator

@post Invalidates iterators and references at or after the point of the

erase, including the `end()` iterator.

@throw type\_error.307 if called on a `null` value; example: `"cannot use

erase() with null"`

@throw invalid\_iterator.202 if called on an iterator which does not belong

to the current JSON value; example: `"iterator does not fit current

value"`

@throw invalid\_iterator.205 if called on a primitive type with invalid

iterator (i.e., any iterator which is not `begin()`); example: `"iterator

out of range"`

@complexity The complexity depends on the type:

- objects: amortized constant

- arrays: linear in distance between @a pos and the end of the container

- strings: linear in the length of the string

- other types: constant

@liveexample{The example shows the result of `erase()` for different JSON

types.,erase\_\_IteratorType}

@sa @ref erase(IteratorType, IteratorType) -- removes the elements in

the given range

@sa @ref erase(const typename object\_t::key\_type&) -- removes the element

from an object at the given key

@sa @ref erase(const size\_type) -- removes the element from an array at

the given index

@since version 1.0.0

\*/

template<class IteratorType, typename std::enable\_if<

std::is\_same<IteratorType, typename basic\_json\_t::iterator>::value or

std::is\_same<IteratorType, typename basic\_json\_t::const\_iterator>::value, int>::type

= 0>

IteratorType erase(IteratorType pos)

{

// make sure iterator fits the current value

if (JSON\_HEDLEY\_UNLIKELY(this != pos.m\_object))

{

JSON\_THROW(invalid\_iterator::create(202, "iterator does not fit current value"));

}

IteratorType result = end();

switch (m\_type)

{

case value\_t::boolean:

case value\_t::number\_float:

case value\_t::number\_integer:

case value\_t::number\_unsigned:

case value\_t::string:

{

if (JSON\_HEDLEY\_UNLIKELY(not pos.m\_it.primitive\_iterator.is\_begin()))

{

JSON\_THROW(invalid\_iterator::create(205, "iterator out of range"));

}

if (is\_string())

{

AllocatorType<string\_t> alloc;

std::allocator\_traits<decltype(alloc)>::destroy(alloc, m\_value.string);

std::allocator\_traits<decltype(alloc)>::deallocate(alloc, m\_value.string, 1);

m\_value.string = nullptr;

}

m\_type = value\_t::null;

assert\_invariant();

break;

}

case value\_t::object:

{

result.m\_it.object\_iterator = m\_value.object->erase(pos.m\_it.object\_iterator);

break;

}

case value\_t::array:

{

result.m\_it.array\_iterator = m\_value.array->erase(pos.m\_it.array\_iterator);

break;

}

default:

JSON\_THROW(type\_error::create(307, "cannot use erase() with " + std::string(type\_name())));

}

return result;

}

/\*!

@brief remove elements given an iterator range

Removes the element specified by the range `[first; last)`. The iterator

@a first does not need to be dereferenceable if `first == last`: erasing

an empty range is a no-op.

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

will be `null`.

@param[in] first iterator to the beginning of the range to remove

@param[in] last iterator past the end of the range to remove

@return Iterator following the last removed element. If the iterator @a

second refers to the last element, the `end()` iterator is returned.

@tparam IteratorType an @ref iterator or @ref const\_iterator

@post Invalidates iterators and references at or after the point of the

erase, including the `end()` iterator.

@throw type\_error.307 if called on a `null` value; example: `"cannot use

erase() with null"`

@throw invalid\_iterator.203 if called on iterators which does not belong

to the current JSON value; example: `"iterators do not fit current value"`

@throw invalid\_iterator.204 if called on a primitive type with invalid

iterators (i.e., if `first != begin()` and `last != end()`); example:

`"iterators out of range"`

@complexity The complexity depends on the type:

- objects: `log(size()) + std::distance(first, last)`

- arrays: linear in the distance between @a first and @a last, plus linear

in the distance between @a last and end of the container

- strings: linear in the length of the string

- other types: constant

@liveexample{The example shows the result of `erase()` for different JSON

types.,erase\_\_IteratorType\_IteratorType}

@sa @ref erase(IteratorType) -- removes the element at a given position

@sa @ref erase(const typename object\_t::key\_type&) -- removes the element

from an object at the given key

@sa @ref erase(const size\_type) -- removes the element from an array at

the given index

@since version 1.0.0

\*/

template<class IteratorType, typename std::enable\_if<

std::is\_same<IteratorType, typename basic\_json\_t::iterator>::value or

std::is\_same<IteratorType, typename basic\_json\_t::const\_iterator>::value, int>::type

= 0>

IteratorType erase(IteratorType first, IteratorType last)

{

// make sure iterator fits the current value

if (JSON\_HEDLEY\_UNLIKELY(this != first.m\_object or this != last.m\_object))

{

JSON\_THROW(invalid\_iterator::create(203, "iterators do not fit current value"));

}

IteratorType result = end();

switch (m\_type)

{

case value\_t::boolean:

case value\_t::number\_float:

case value\_t::number\_integer:

case value\_t::number\_unsigned:

case value\_t::string:

{

if (JSON\_HEDLEY\_LIKELY(not first.m\_it.primitive\_iterator.is\_begin()

or not last.m\_it.primitive\_iterator.is\_end()))

{

JSON\_THROW(invalid\_iterator::create(204, "iterators out of range"));

}

if (is\_string())

{

AllocatorType<string\_t> alloc;

std::allocator\_traits<decltype(alloc)>::destroy(alloc, m\_value.string);

std::allocator\_traits<decltype(alloc)>::deallocate(alloc, m\_value.string, 1);

m\_value.string = nullptr;

}

m\_type = value\_t::null;

assert\_invariant();

break;

}

case value\_t::object:

{

result.m\_it.object\_iterator = m\_value.object->erase(first.m\_it.object\_iterator,

last.m\_it.object\_iterator);

break;

}

case value\_t::array:

{

result.m\_it.array\_iterator = m\_value.array->erase(first.m\_it.array\_iterator,

last.m\_it.array\_iterator);

break;

}

default:

JSON\_THROW(type\_error::create(307, "cannot use erase() with " + std::string(type\_name())));

}

return result;

}

/\*!

@brief remove element from a JSON object given a key

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

@param[in] key value of the elements to remove

@return Number of elements removed. If @a ObjectType is the default

`std::map` type, the return value will always be `0` (@a key was not

found) or `1` (@a key was found).

@post References and iterators to the erased elements are invalidated.

Other references and iterators are not affected.

@throw type\_error.307 when called on a type other than JSON object;

example: `"cannot use erase() with null"`

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

@liveexample{The example shows the effect of `erase()`.,erase\_\_key\_type}

@sa @ref erase(IteratorType) -- removes the element at a given position

@sa @ref erase(IteratorType, IteratorType) -- removes the elements in

the given range

@sa @ref erase(const size\_type) -- removes the element from an array at

the given index

@since version 1.0.0

\*/

size\_type erase(const typename object\_t::key\_type& key)

{

// this erase only works for objects

if (JSON\_HEDLEY\_LIKELY(is\_object()))

{

return m\_value.object->erase(key);

}

JSON\_THROW(type\_error::create(307, "cannot use erase() with " + std::string(type\_name())));

}

/\*!

@brief remove element from a JSON array given an index

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

@param[in] idx index of the element to remove

@throw type\_error.307 when called on a type other than JSON object;

example: `"cannot use erase() with null"`

@throw out\_of\_range.401 when `idx >= size()`; example: `"array index 17

is out of range"`

@complexity Linear in distance between @a idx and the end of the container.

@liveexample{The example shows the effect of `erase()`.,erase\_\_size\_type}

@sa @ref erase(IteratorType) -- removes the element at a given position

@sa @ref erase(IteratorType, IteratorType) -- removes the elements in

the given range

@sa @ref erase(const typename object\_t::key\_type&) -- removes the element

from an object at the given key

@since version 1.0.0

\*/

void erase(const size\_type idx)

{

// this erase only works for arrays

if (JSON\_HEDLEY\_LIKELY(is\_array()))

{

if (JSON\_HEDLEY\_UNLIKELY(idx >= size()))

{

JSON\_THROW(out\_of\_range::create(401, "array index " + std::to\_string(idx) + " is out of range"));

}

m\_value.array->erase(m\_value.array->begin() + static\_cast<difference\_type>(idx));

}

else

{

JSON\_THROW(type\_error::create(307, "cannot use erase() with " + std::string(type\_name())));

}

}

/// @}

////////////

// lookup //

////////////

/// @name lookup

/// @{

/\*!

@brief find an element in a JSON object

Finds an element in a JSON object with key equivalent to @a key. If the

element is not found or the JSON value is not an object, end() is

returned.

@note This method always returns @ref end() when executed on a JSON type

that is not an object.

@param[in] key key value of the element to search for.

@return Iterator to an element with key equivalent to @a key. If no such

element is found or the JSON value is not an object, past-the-end (see

@ref end()) iterator is returned.

@complexity Logarithmic in the size of the JSON object.

@liveexample{The example shows how `find()` is used.,find\_\_key\_type}

@sa @ref contains(KeyT&&) const -- checks whether a key exists

@since version 1.0.0

\*/

template<typename KeyT>

iterator find(KeyT&& key)

{

auto result = end();

if (is\_object())

{

result.m\_it.object\_iterator = m\_value.object->find(std::forward<KeyT>(key));

}

return result;

}

/\*!

@brief find an element in a JSON object

@copydoc find(KeyT&&)

\*/

template<typename KeyT>

const\_iterator find(KeyT&& key) const

{

auto result = cend();

if (is\_object())

{

result.m\_it.object\_iterator = m\_value.object->find(std::forward<KeyT>(key));

}

return result;

}

/\*!

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

Returns the number of elements with key @a key. If ObjectType is the

default `std::map` type, the return value will always be `0` (@a key was

not found) or `1` (@a key was found).

@note This method always returns `0` when executed on a JSON type that is

not an object.

@param[in] key key value of the element to count

@return Number of elements with key @a key. If the JSON value is not an

object, the return value will be `0`.

@complexity Logarithmic in the size of the JSON object.

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

@since version 1.0.0

\*/

template<typename KeyT>

size\_type count(KeyT&& key) const

{

// return 0 for all nonobject types

return is\_object() ? m\_value.object->count(std::forward<KeyT>(key)) : 0;

}

/\*!

@brief check the existence of an element in a JSON object

Check whether an element exists in a JSON object with key equivalent to

@a key. If the element is not found or the JSON value is not an object,

false is returned.

@note This method always returns false when executed on a JSON type

that is not an object.

@param[in] key key value to check its existence.

@return true if an element with specified @a key exists. If no such

element with such key is found or the JSON value is not an object,

false is returned.

@complexity Logarithmic in the size of the JSON object.

@liveexample{The following code shows an example for `contains()`.,contains}

@sa @ref find(KeyT&&) -- returns an iterator to an object element

@sa @ref contains(const json\_pointer&) const -- checks the existence for a JSON pointer

@since version 3.6.0

\*/

template<typename KeyT, typename std::enable\_if<

not std::is\_same<typename std::decay<KeyT>::type, json\_pointer>::value, int>::type = 0>

bool contains(KeyT && key) const

{

return is\_object() and m\_value.object->find(std::forward<KeyT>(key)) != m\_value.object->end();

}

/\*!

@brief check the existence of an element in a JSON object given a JSON pointer

Check whether the given JSON pointer @a ptr can be resolved in the current

JSON value.

@note This method can be executed on any JSON value type.

@param[in] ptr JSON pointer to check its existence.

@return true if the JSON pointer can be resolved to a stored value, false

otherwise.

@post If `j.contains(ptr)` returns true, it is safe to call `j[ptr]`.

@throw parse\_error.106 if an array index begins with '0'

@throw parse\_error.109 if an array index was not a number

@complexity Logarithmic in the size of the JSON object.

@liveexample{The following code shows an example for `contains()`.,contains\_json\_pointer}

@sa @ref contains(KeyT &&) const -- checks the existence of a key

@since version 3.7.0

\*/

bool contains(const json\_pointer& ptr) const

{

return ptr.contains(this);

}

/// @}

///////////////

// iterators //

///////////////

/// @name iterators

/// @{

/\*!

@brief returns an iterator to the first element

Returns an iterator to the first element.

@image html range-begin-end.svg "Illustration from cppreference.com"

@return iterator to the first element

@complexity Constant.

@requirement This function helps `basic\_json` satisfying the

[Container](https://en.cppreference.com/w/cpp/named\_req/Container)

requirements:

- The complexity is constant.

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

@sa @ref cbegin() -- returns a const iterator to the beginning

@sa @ref end() -- returns an iterator to the end

@sa @ref cend() -- returns a const iterator to the end

@since version 1.0.0

\*/

iterator begin() noexcept

{

iterator result(this);

result.set\_begin();

return result;

}

/\*!

@copydoc basic\_json::cbegin()

\*/

const\_iterator begin() const noexcept

{

return cbegin();

}

/\*!

@brief returns a const iterator to the first element

Returns a const iterator to the first element.

@image html range-begin-end.svg "Illustration from cppreference.com"

@return const iterator to the first element

@complexity Constant.

@requirement This function helps `basic\_json` satisfying the

[Container](https://en.cppreference.com/w/cpp/named\_req/Container)

requirements:

- The complexity is constant.

- Has the semantics of `const\_cast<const basic\_json&>(\*this).begin()`.

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

@sa @ref begin() -- returns an iterator to the beginning

@sa @ref end() -- returns an iterator to the end

@sa @ref cend() -- returns a const iterator to the end

@since version 1.0.0

\*/

const\_iterator cbegin() const noexcept

{

const\_iterator result(this);

result.set\_begin();

return result;

}

/\*!

@brief returns an iterator to one past the last element

Returns an iterator to one past the last element.

@image html range-begin-end.svg "Illustration from cppreference.com"

@return iterator one past the last element

@complexity Constant.

@requirement This function helps `basic\_json` satisfying the

[Container](https://en.cppreference.com/w/cpp/named\_req/Container)

requirements:

- The complexity is constant.

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

@sa @ref cend() -- returns a const iterator to the end

@sa @ref begin() -- returns an iterator to the beginning

@sa @ref cbegin() -- returns a const iterator to the beginning

@since version 1.0.0

\*/

iterator end() noexcept

{

iterator result(this);

result.set\_end();

return result;

}

/\*!

@copydoc basic\_json::cend()

\*/

const\_iterator end() const noexcept

{

return cend();

}

/\*!

@brief returns a const iterator to one past the last element

Returns a const iterator to one past the last element.

@image html range-begin-end.svg "Illustration from cppreference.com"

@return const iterator one past the last element

@complexity Constant.

@requirement This function helps `basic\_json` satisfying the

[Container](https://en.cppreference.com/w/cpp/named\_req/Container)

requirements:

- The complexity is constant.

- Has the semantics of `const\_cast<const basic\_json&>(\*this).end()`.

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

@sa @ref end() -- returns an iterator to the end

@sa @ref begin() -- returns an iterator to the beginning

@sa @ref cbegin() -- returns a const iterator to the beginning

@since version 1.0.0

\*/

const\_iterator cend() const noexcept

{

const\_iterator result(this);

result.set\_end();

return result;

}

/\*!

@brief returns an iterator to the reverse-beginning

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

@image html range-rbegin-rend.svg "Illustration from cppreference.com"

@complexity Constant.

@requirement This function helps `basic\_json` satisfying the

[ReversibleContainer](https://en.cppreference.com/w/cpp/named\_req/ReversibleContainer)

requirements:

- The complexity is constant.

- Has the semantics of `reverse\_iterator(end())`.

@liveexample{The following code shows an example for `rbegin()`.,rbegin}

@sa @ref crbegin() -- returns a const reverse iterator to the beginning

@sa @ref rend() -- returns a reverse iterator to the end

@sa @ref crend() -- returns a const reverse iterator to the end

@since version 1.0.0

\*/

reverse\_iterator rbegin() noexcept

{

return reverse\_iterator(end());

}

/\*!

@copydoc basic\_json::crbegin()

\*/

const\_reverse\_iterator rbegin() const noexcept

{

return crbegin();

}

/\*!

@brief returns an iterator to the reverse-end

Returns an iterator to the reverse-end; that is, one before the first

element.

@image html range-rbegin-rend.svg "Illustration from cppreference.com"

@complexity Constant.

@requirement This function helps `basic\_json` satisfying the

[ReversibleContainer](https://en.cppreference.com/w/cpp/named\_req/ReversibleContainer)

requirements:

- The complexity is constant.

- Has the semantics of `reverse\_iterator(begin())`.

@liveexample{The following code shows an example for `rend()`.,rend}

@sa @ref crend() -- returns a const reverse iterator to the end

@sa @ref rbegin() -- returns a reverse iterator to the beginning

@sa @ref crbegin() -- returns a const reverse iterator to the beginning

@since version 1.0.0

\*/

reverse\_iterator rend() noexcept

{

return reverse\_iterator(begin());

}

/\*!

@copydoc basic\_json::crend()

\*/

const\_reverse\_iterator rend() const noexcept

{

return crend();

}

/\*!

@brief returns a const reverse iterator to the last element

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

element.

@image html range-rbegin-rend.svg "Illustration from cppreference.com"

@complexity Constant.

@requirement This function helps `basic\_json` satisfying the

[ReversibleContainer](https://en.cppreference.com/w/cpp/named\_req/ReversibleContainer)

requirements:

- The complexity is constant.

- Has the semantics of `const\_cast<const basic\_json&>(\*this).rbegin()`.

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

@sa @ref rbegin() -- returns a reverse iterator to the beginning

@sa @ref rend() -- returns a reverse iterator to the end

@sa @ref crend() -- returns a const reverse iterator to the end

@since version 1.0.0

\*/

const\_reverse\_iterator crbegin() const noexcept

{

return const\_reverse\_iterator(cend());

}

/\*!

@brief returns a const reverse iterator to one before the first

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

the first element.

@image html range-rbegin-rend.svg "Illustration from cppreference.com"

@complexity Constant.

@requirement This function helps `basic\_json` satisfying the

[ReversibleContainer](https://en.cppreference.com/w/cpp/named\_req/ReversibleContainer)

requirements:

- The complexity is constant.

- Has the semantics of `const\_cast<const basic\_json&>(\*this).rend()`.

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

@sa @ref rend() -- returns a reverse iterator to the end

@sa @ref rbegin() -- returns a reverse iterator to the beginning

@sa @ref crbegin() -- returns a const reverse iterator to the beginning

@since version 1.0.0

\*/

const\_reverse\_iterator crend() const noexcept

{

return const\_reverse\_iterator(cbegin());

}

public:

/\*!

@brief wrapper to access iterator member functions in range-based for

This function allows to access @ref iterator::key() and @ref

iterator::value() during range-based for loops. In these loops, a

reference to the JSON values is returned, so there is no access to the

underlying iterator.

For loop without iterator\_wrapper:

@code{cpp}

for (auto it = j\_object.begin(); it != j\_object.end(); ++it)

{

std::cout << "key: " << it.key() << ", value:" << it.value() << '\n';

}

@endcode

Range-based for loop without iterator proxy:

@code{cpp}

for (auto it : j\_object)

{

// "it" is of type json::reference and has no key() member

std::cout << "value: " << it << '\n';

}

@endcode

Range-based for loop with iterator proxy:

@code{cpp}

for (auto it : json::iterator\_wrapper(j\_object))

{

std::cout << "key: " << it.key() << ", value:" << it.value() << '\n';

}

@endcode

@note When iterating over an array, `key()` will return the index of the

element as string (see example).

@param[in] ref reference to a JSON value

@return iteration proxy object wrapping @a ref with an interface to use in

range-based for loops

@liveexample{The following code shows how the wrapper is used,iterator\_wrapper}

@exceptionsafety Strong guarantee: if an exception is thrown, there are no

changes in the JSON value.

@complexity Constant.

@note The name of this function is not yet final and may change in the

future.

@deprecated This stream operator is deprecated and will be removed in

future 4.0.0 of the library. Please use @ref items() instead;

that is, replace `json::iterator\_wrapper(j)` with `j.items()`.

\*/

JSON\_HEDLEY\_DEPRECATED(3.1.0)

static iteration\_proxy<iterator> iterator\_wrapper(reference ref) noexcept

{

return ref.items();

}

/\*!

@copydoc iterator\_wrapper(reference)

\*/

JSON\_HEDLEY\_DEPRECATED(3.1.0)

static iteration\_proxy<const\_iterator> iterator\_wrapper(const\_reference ref) noexcept

{

return ref.items();

}

/\*!

@brief helper to access iterator member functions in range-based for

This function allows to access @ref iterator::key() and @ref

iterator::value() during range-based for loops. In these loops, a

reference to the JSON values is returned, so there is no access to the

underlying iterator.

For loop without `items()` function:

@code{cpp}

for (auto it = j\_object.begin(); it != j\_object.end(); ++it)

{

std::cout << "key: " << it.key() << ", value:" << it.value() << '\n';

}

@endcode

Range-based for loop without `items()` function:

@code{cpp}

for (auto it : j\_object)

{

// "it" is of type json::reference and has no key() member

std::cout << "value: " << it << '\n';

}

@endcode

Range-based for loop with `items()` function:

@code{cpp}

for (auto& el : j\_object.items())

{

std::cout << "key: " << el.key() << ", value:" << el.value() << '\n';

}

@endcode

The `items()` function also allows to use

[structured bindings](https://en.cppreference.com/w/cpp/language/structured\_binding)

(C++17):

@code{cpp}

for (auto& [key, val] : j\_object.items())

{

std::cout << "key: " << key << ", value:" << val << '\n';

}

@endcode

@note When iterating over an array, `key()` will return the index of the

element as string (see example). For primitive types (e.g., numbers),

`key()` returns an empty string.

@return iteration proxy object wrapping @a ref with an interface to use in

range-based for loops

@liveexample{The following code shows how the function is used.,items}

@exceptionsafety Strong guarantee: if an exception is thrown, there are no

changes in the JSON value.

@complexity Constant.

@since version 3.1.0, structured bindings support since 3.5.0.

\*/

iteration\_proxy<iterator> items() noexcept

{

return iteration\_proxy<iterator>(\*this);

}

/\*!

@copydoc items()

\*/

iteration\_proxy<const\_iterator> items() const noexcept

{

return iteration\_proxy<const\_iterator>(\*this);

}

/// @}

//////////////

// capacity //

//////////////

/// @name capacity

/// @{

/\*!

@brief checks whether the container is empty.

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

@return The return value depends on the different types and is

defined as follows:

Value type | return value

----------- | -------------

null | `true`

boolean | `false`

string | `false`

number | `false`

object | result of function `object\_t::empty()`

array | result of function `array\_t::empty()`

@liveexample{The following code uses `empty()` to check if a JSON

object contains any elements.,empty}

@complexity Constant, as long as @ref array\_t and @ref object\_t satisfy

the Container concept; that is, their `empty()` functions have constant

complexity.

@iterators No changes.

@exceptionsafety No-throw guarantee: this function never throws exceptions.

@note This function does not return whether a string stored as JSON value

is empty - it returns whether the JSON container itself is empty which is

false in the case of a string.

@requirement This function helps `basic\_json` satisfying the

[Container](https://en.cppreference.com/w/cpp/named\_req/Container)

requirements:

- The complexity is constant.

- Has the semantics of `begin() == end()`.

@sa @ref size() -- returns the number of elements

@since version 1.0.0

\*/

bool empty() const noexcept

{

switch (m\_type)

{

case value\_t::null:

{

// null values are empty

return true;

}

case value\_t::array:

{

// delegate call to array\_t::empty()

return m\_value.array->empty();

}

case value\_t::object:

{

// delegate call to object\_t::empty()

return m\_value.object->empty();

}

default:

{

// all other types are nonempty

return false;

}

}

}

/\*!

@brief returns the number of elements

Returns the number of elements in a JSON value.

@return The return value depends on the different types and is

defined as follows:

Value type | return value

----------- | -------------

null | `0`

boolean | `1`

string | `1`

number | `1`

object | result of function object\_t::size()

array | result of function array\_t::size()

@liveexample{The following code calls `size()` on the different value

types.,size}

@complexity Constant, as long as @ref array\_t and @ref object\_t satisfy

the Container concept; that is, their size() functions have constant

complexity.

@iterators No changes.

@exceptionsafety No-throw guarantee: this function never throws exceptions.

@note This function does not return the length of a string stored as JSON

value - it returns the number of elements in the JSON value which is 1 in

the case of a string.

@requirement This function helps `basic\_json` satisfying the

[Container](https://en.cppreference.com/w/cpp/named\_req/Container)

requirements:

- The complexity is constant.

- Has the semantics of `std::distance(begin(), end())`.

@sa @ref empty() -- checks whether the container is empty

@sa @ref max\_size() -- returns the maximal number of elements

@since version 1.0.0

\*/

size\_type size() const noexcept

{

switch (m\_type)

{

case value\_t::null:

{

// null values are empty

return 0;

}

case value\_t::array:

{

// delegate call to array\_t::size()

return m\_value.array->size();

}

case value\_t::object:

{

// delegate call to object\_t::size()

return m\_value.object->size();

}

default:

{

// all other types have size 1

return 1;

}

}

}

/\*!

@brief returns the maximum possible number of elements

Returns the maximum number of elements a JSON value is able to hold due to

system or library implementation limitations, i.e. `std::distance(begin(),

end())` for the JSON value.

@return The return value depends on the different types and is

defined as follows:

Value type | return value

----------- | -------------

null | `0` (same as `size()`)

boolean | `1` (same as `size()`)

string | `1` (same as `size()`)

number | `1` (same as `size()`)

object | result of function `object\_t::max\_size()`

array | result of function `array\_t::max\_size()`

@liveexample{The following code calls `max\_size()` on the different value

types. Note the output is implementation specific.,max\_size}

@complexity Constant, as long as @ref array\_t and @ref object\_t satisfy

the Container concept; that is, their `max\_size()` functions have constant

complexity.

@iterators No changes.

@exceptionsafety No-throw guarantee: this function never throws exceptions.

@requirement This function helps `basic\_json` satisfying the

[Container](https://en.cppreference.com/w/cpp/named\_req/Container)

requirements:

- The complexity is constant.

- Has the semantics of returning `b.size()` where `b` is the largest

possible JSON value.

@sa @ref size() -- returns the number of elements

@since version 1.0.0

\*/

size\_type max\_size() const noexcept

{

switch (m\_type)

{

case value\_t::array:

{

// delegate call to array\_t::max\_size()

return m\_value.array->max\_size();

}

case value\_t::object:

{

// delegate call to object\_t::max\_size()

return m\_value.object->max\_size();

}

default:

{

// all other types have max\_size() == size()

return size();

}

}

}

/// @}

///////////////

// modifiers //

///////////////

/// @name modifiers

/// @{

/\*!

@brief clears the contents

Clears the content of a JSON value and resets it to the default value as

if @ref basic\_json(value\_t) would have been called with the current value

type from @ref type():

Value type | initial value

----------- | -------------

null | `null`

boolean | `false`

string | `""`

number | `0`

object | `{}`

array | `[]`

@post Has the same effect as calling

@code {.cpp}

\*this = basic\_json(type());

@endcode

@liveexample{The example below shows the effect of `clear()` to different

JSON types.,clear}

@complexity Linear in the size of the JSON value.

@iterators All iterators, pointers and references related to this container

are invalidated.

@exceptionsafety No-throw guarantee: this function never throws exceptions.

@sa @ref basic\_json(value\_t) -- constructor that creates an object with the

same value than calling `clear()`

@since version 1.0.0

\*/

void clear() noexcept

{

switch (m\_type)

{

case value\_t::number\_integer:

{

m\_value.number\_integer = 0;

break;

}

case value\_t::number\_unsigned:

{

m\_value.number\_unsigned = 0;

break;

}

case value\_t::number\_float:

{

m\_value.number\_float = 0.0;

break;

}

case value\_t::boolean:

{

m\_value.boolean = false;

break;

}

case value\_t::string:

{

m\_value.string->clear();

break;

}

case value\_t::array:

{

m\_value.array->clear();

break;

}

case value\_t::object:

{

m\_value.object->clear();

break;

}

default:

break;

}

}

/\*!

@brief add an object to an array

Appends the given element @a val to the end of the JSON value. If the

function is called on a JSON null value, an empty array is created before

appending @a val.

@param[in] val the value to add to the JSON array

@throw type\_error.308 when called on a type other than JSON array or

null; example: `"cannot use push\_back() with number"`

@complexity Amortized constant.

@liveexample{The example shows how `push\_back()` and `+=` can be used to

add elements to a JSON array. Note how the `null` value was silently

converted to a JSON array.,push\_back}

@since version 1.0.0

\*/

void push\_back(basic\_json&& val)

{

// push\_back only works for null objects or arrays

if (JSON\_HEDLEY\_UNLIKELY(not(is\_null() or is\_array())))

{

JSON\_THROW(type\_error::create(308, "cannot use push\_back() with " + std::string(type\_name())));

}

// transform null object into an array

if (is\_null())

{

m\_type = value\_t::array;

m\_value = value\_t::array;

assert\_invariant();

}

// add element to array (move semantics)

m\_value.array->push\_back(std::move(val));

// invalidate object: mark it null so we do not call the destructor

// cppcheck-suppress accessMoved

val.m\_type = value\_t::null;

}

/\*!

@brief add an object to an array

@copydoc push\_back(basic\_json&&)

\*/

reference operator+=(basic\_json&& val)

{

push\_back(std::move(val));

return \*this;

}

/\*!

@brief add an object to an array

@copydoc push\_back(basic\_json&&)

\*/

void push\_back(const basic\_json& val)

{

// push\_back only works for null objects or arrays

if (JSON\_HEDLEY\_UNLIKELY(not(is\_null() or is\_array())))

{

JSON\_THROW(type\_error::create(308, "cannot use push\_back() with " + std::string(type\_name())));

}

// transform null object into an array

if (is\_null())

{

m\_type = value\_t::array;

m\_value = value\_t::array;

assert\_invariant();

}

// add element to array

m\_value.array->push\_back(val);

}

/\*!

@brief add an object to an array

@copydoc push\_back(basic\_json&&)

\*/

reference operator+=(const basic\_json& val)

{

push\_back(val);

return \*this;

}

/\*!

@brief add an object to an object

Inserts the given element @a val to the JSON object. If the function is

called on a JSON null value, an empty object is created before inserting

@a val.

@param[in] val the value to add to the JSON object

@throw type\_error.308 when called on a type other than JSON object or

null; example: `"cannot use push\_back() with number"`

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

@liveexample{The example shows how `push\_back()` and `+=` can be used to

add elements to a JSON object. Note how the `null` value was silently

converted to a JSON object.,push\_back\_\_object\_t\_\_value}

@since version 1.0.0

\*/

void push\_back(const typename object\_t::value\_type& val)

{

// push\_back only works for null objects or objects

if (JSON\_HEDLEY\_UNLIKELY(not(is\_null() or is\_object())))

{

JSON\_THROW(type\_error::create(308, "cannot use push\_back() with " + std::string(type\_name())));

}

// transform null object into an object

if (is\_null())

{

m\_type = value\_t::object;

m\_value = value\_t::object;

assert\_invariant();

}

// add element to array

m\_value.object->insert(val);

}

/\*!

@brief add an object to an object

@copydoc push\_back(const typename object\_t::value\_type&)

\*/

reference operator+=(const typename object\_t::value\_type& val)

{

push\_back(val);

return \*this;

}

/\*!

@brief add an object to an object

This function allows to use `push\_back` with an initializer list. In case

1. the current value is an object,

2. the initializer list @a init contains only two elements, and

3. the first element of @a init is a string,

@a init is converted into an object element and added using

@ref push\_back(const typename object\_t::value\_type&). Otherwise, @a init

is converted to a JSON value and added using @ref push\_back(basic\_json&&).

@param[in] init an initializer list

@complexity Linear in the size of the initializer list @a init.

@note This function is required to resolve an ambiguous overload error,

because pairs like `{"key", "value"}` can be both interpreted as

`object\_t::value\_type` or `std::initializer\_list<basic\_json>`, see

https://github.com/nlohmann/json/issues/235 for more information.

@liveexample{The example shows how initializer lists are treated as

objects when possible.,push\_back\_\_initializer\_list}

\*/

void push\_back(initializer\_list\_t init)

{

if (is\_object() and init.size() == 2 and (\*init.begin())->is\_string())

{

basic\_json&& key = init.begin()->moved\_or\_copied();

push\_back(typename object\_t::value\_type(

std::move(key.get\_ref<string\_t&>()), (init.begin() + 1)->moved\_or\_copied()));

}

else

{

push\_back(basic\_json(init));

}

}

/\*!

@brief add an object to an object

@copydoc push\_back(initializer\_list\_t)

\*/

reference operator+=(initializer\_list\_t init)

{

push\_back(init);

return \*this;

}

/\*!

@brief add an object to an array

Creates a JSON value from the passed parameters @a args to the end of the

JSON value. If the function is called on a JSON null value, an empty array

is created before appending the value created from @a args.

@param[in] args arguments to forward to a constructor of @ref basic\_json

@tparam Args compatible types to create a @ref basic\_json object

@return reference to the inserted element

@throw type\_error.311 when called on a type other than JSON array or

null; example: `"cannot use emplace\_back() with number"`

@complexity Amortized constant.

@liveexample{The example shows how `push\_back()` can be used to add

elements to a JSON array. Note how the `null` value was silently converted

to a JSON array.,emplace\_back}

@since version 2.0.8, returns reference since 3.7.0

\*/

template<class... Args>

reference emplace\_back(Args&& ... args)

{

// emplace\_back only works for null objects or arrays

if (JSON\_HEDLEY\_UNLIKELY(not(is\_null() or is\_array())))

{

JSON\_THROW(type\_error::create(311, "cannot use emplace\_back() with " + std::string(type\_name())));

}

// transform null object into an array

if (is\_null())

{

m\_type = value\_t::array;

m\_value = value\_t::array;

assert\_invariant();

}

// add element to array (perfect forwarding)

#ifdef JSON\_HAS\_CPP\_17

return m\_value.array->emplace\_back(std::forward<Args>(args)...);

#else

m\_value.array->emplace\_back(std::forward<Args>(args)...);

return m\_value.array->back();

#endif

}

/\*!

@brief add an object to an object if key does not exist

Inserts a new element into a JSON object constructed in-place with the

given @a args if there is no element with the key in the container. If the

function is called on a JSON null value, an empty object is created before

appending the value created from @a args.

@param[in] args arguments to forward to a constructor of @ref basic\_json

@tparam Args compatible types to create a @ref basic\_json object

@return a pair consisting of an iterator to the inserted element, or the

already-existing element if no insertion happened, and a bool

denoting whether the insertion took place.

@throw type\_error.311 when called on a type other than JSON object or

null; example: `"cannot use emplace() with number"`

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

@liveexample{The example shows how `emplace()` can be used to add elements

to a JSON object. Note how the `null` value was silently converted to a

JSON object. Further note how no value is added if there was already one

value stored with the same key.,emplace}

@since version 2.0.8

\*/

template<class... Args>

std::pair<iterator, bool> emplace(Args&& ... args)

{

// emplace only works for null objects or arrays

if (JSON\_HEDLEY\_UNLIKELY(not(is\_null() or is\_object())))

{

JSON\_THROW(type\_error::create(311, "cannot use emplace() with " + std::string(type\_name())));

}

// transform null object into an object

if (is\_null())

{

m\_type = value\_t::object;

m\_value = value\_t::object;

assert\_invariant();

}

// add element to array (perfect forwarding)

auto res = m\_value.object->emplace(std::forward<Args>(args)...);

// create result iterator and set iterator to the result of emplace

auto it = begin();

it.m\_it.object\_iterator = res.first;

// return pair of iterator and boolean

return {it, res.second};

}

/// Helper for insertion of an iterator

/// @note: This uses std::distance to support GCC 4.8,

/// see https://github.com/nlohmann/json/pull/1257

template<typename... Args>

iterator insert\_iterator(const\_iterator pos, Args&& ... args)

{

iterator result(this);

assert(m\_value.array != nullptr);

auto insert\_pos = std::distance(m\_value.array->begin(), pos.m\_it.array\_iterator);

m\_value.array->insert(pos.m\_it.array\_iterator, std::forward<Args>(args)...);

result.m\_it.array\_iterator = m\_value.array->begin() + insert\_pos;

// This could have been written as:

// result.m\_it.array\_iterator = m\_value.array->insert(pos.m\_it.array\_iterator, cnt, val);

// but the return value of insert is missing in GCC 4.8, so it is written this way instead.

return result;

}

/\*!

@brief inserts element

Inserts element @a val before iterator @a pos.

@param[in] pos iterator before which the content will be inserted; may be

the end() iterator

@param[in] val element to insert

@return iterator pointing to the inserted @a val.

@throw type\_error.309 if called on JSON values other than arrays;

example: `"cannot use insert() with string"`

@throw invalid\_iterator.202 if @a pos is not an iterator of \*this;

example: `"iterator does not fit current value"`

@complexity Constant plus linear in the distance between @a pos and end of

the container.

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

@since version 1.0.0

\*/

iterator insert(const\_iterator pos, const basic\_json& val)

{

// insert only works for arrays

if (JSON\_HEDLEY\_LIKELY(is\_array()))

{

// check if iterator pos fits to this JSON value

if (JSON\_HEDLEY\_UNLIKELY(pos.m\_object != this))

{

JSON\_THROW(invalid\_iterator::create(202, "iterator does not fit current value"));

}

// insert to array and return iterator

return insert\_iterator(pos, val);

}

JSON\_THROW(type\_error::create(309, "cannot use insert() with " + std::string(type\_name())));

}

/\*!

@brief inserts element

@copydoc insert(const\_iterator, const basic\_json&)

\*/

iterator insert(const\_iterator pos, basic\_json&& val)

{

return insert(pos, val);

}

/\*!

@brief inserts elements

Inserts @a cnt copies of @a val before iterator @a pos.

@param[in] pos iterator before which the content will be inserted; may be

the end() iterator

@param[in] cnt number of copies of @a val to insert

@param[in] val element to insert

@return iterator pointing to the first element inserted, or @a pos if

`cnt==0`

@throw type\_error.309 if called on JSON values other than arrays; example:

`"cannot use insert() with string"`

@throw invalid\_iterator.202 if @a pos is not an iterator of \*this;

example: `"iterator does not fit current value"`

@complexity Linear in @a cnt plus linear in the distance between @a pos

and end of the container.

@liveexample{The example shows how `insert()` is used.,insert\_\_count}

@since version 1.0.0

\*/

iterator insert(const\_iterator pos, size\_type cnt, const basic\_json& val)

{

// insert only works for arrays

if (JSON\_HEDLEY\_LIKELY(is\_array()))

{

// check if iterator pos fits to this JSON value

if (JSON\_HEDLEY\_UNLIKELY(pos.m\_object != this))

{

JSON\_THROW(invalid\_iterator::create(202, "iterator does not fit current value"));

}

// insert to array and return iterator

return insert\_iterator(pos, cnt, val);

}

JSON\_THROW(type\_error::create(309, "cannot use insert() with " + std::string(type\_name())));

}

/\*!

@brief inserts elements

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

@param[in] pos iterator before which the content will be inserted; may be

the end() iterator

@param[in] first begin of the range of elements to insert

@param[in] last end of the range of elements to insert

@throw type\_error.309 if called on JSON values other than arrays; example:

`"cannot use insert() with string"`

@throw invalid\_iterator.202 if @a pos is not an iterator of \*this;

example: `"iterator does not fit current value"`

@throw invalid\_iterator.210 if @a first and @a last do not belong to the

same JSON value; example: `"iterators do not fit"`

@throw invalid\_iterator.211 if @a first or @a last are iterators into

container for which insert is called; example: `"passed iterators may not

belong to container"`

@return iterator pointing to the first element inserted, or @a pos if

`first==last`

@complexity Linear in `std::distance(first, last)` plus linear in the

distance between @a pos and end of the container.

@liveexample{The example shows how `insert()` is used.,insert\_\_range}

@since version 1.0.0

\*/

iterator insert(const\_iterator pos, const\_iterator first, const\_iterator last)

{

// insert only works for arrays

if (JSON\_HEDLEY\_UNLIKELY(not is\_array()))

{

JSON\_THROW(type\_error::create(309, "cannot use insert() with " + std::string(type\_name())));

}

// check if iterator pos fits to this JSON value

if (JSON\_HEDLEY\_UNLIKELY(pos.m\_object != this))

{

JSON\_THROW(invalid\_iterator::create(202, "iterator does not fit current value"));

}

// check if range iterators belong to the same JSON object

if (JSON\_HEDLEY\_UNLIKELY(first.m\_object != last.m\_object))

{

JSON\_THROW(invalid\_iterator::create(210, "iterators do not fit"));

}

if (JSON\_HEDLEY\_UNLIKELY(first.m\_object == this))

{

JSON\_THROW(invalid\_iterator::create(211, "passed iterators may not belong to container"));

}

// insert to array and return iterator

return insert\_iterator(pos, first.m\_it.array\_iterator, last.m\_it.array\_iterator);

}

/\*!

@brief inserts elements

Inserts elements from initializer list @a ilist before iterator @a pos.

@param[in] pos iterator before which the content will be inserted; may be

the end() iterator

@param[in] ilist initializer list to insert the values from

@throw type\_error.309 if called on JSON values other than arrays; example:

`"cannot use insert() with string"`

@throw invalid\_iterator.202 if @a pos is not an iterator of \*this;

example: `"iterator does not fit current value"`

@return iterator pointing to the first element inserted, or @a pos if

`ilist` is empty

@complexity Linear in `ilist.size()` plus linear in the distance between

@a pos and end of the container.

@liveexample{The example shows how `insert()` is used.,insert\_\_ilist}

@since version 1.0.0

\*/

iterator insert(const\_iterator pos, initializer\_list\_t ilist)

{

// insert only works for arrays

if (JSON\_HEDLEY\_UNLIKELY(not is\_array()))

{

JSON\_THROW(type\_error::create(309, "cannot use insert() with " + std::string(type\_name())));

}

// check if iterator pos fits to this JSON value

if (JSON\_HEDLEY\_UNLIKELY(pos.m\_object != this))

{

JSON\_THROW(invalid\_iterator::create(202, "iterator does not fit current value"));

}

// insert to array and return iterator

return insert\_iterator(pos, ilist.begin(), ilist.end());

}

/\*!

@brief inserts elements

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

@param[in] first begin of the range of elements to insert

@param[in] last end of the range of elements to insert

@throw type\_error.309 if called on JSON values other than objects; example:

`"cannot use insert() with string"`

@throw invalid\_iterator.202 if iterator @a first or @a last does does not

point to an object; example: `"iterators first and last must point to

objects"`

@throw invalid\_iterator.210 if @a first and @a last do not belong to the

same JSON value; example: `"iterators do not fit"`

@complexity Logarithmic: `O(N\*log(size() + N))`, where `N` is the number

of elements to insert.

@liveexample{The example shows how `insert()` is used.,insert\_\_range\_object}

@since version 3.0.0

\*/

void insert(const\_iterator first, const\_iterator last)

{

// insert only works for objects

if (JSON\_HEDLEY\_UNLIKELY(not is\_object()))

{

JSON\_THROW(type\_error::create(309, "cannot use insert() with " + std::string(type\_name())));

}

// check if range iterators belong to the same JSON object

if (JSON\_HEDLEY\_UNLIKELY(first.m\_object != last.m\_object))

{

JSON\_THROW(invalid\_iterator::create(210, "iterators do not fit"));

}

// passed iterators must belong to objects

if (JSON\_HEDLEY\_UNLIKELY(not first.m\_object->is\_object()))

{

JSON\_THROW(invalid\_iterator::create(202, "iterators first and last must point to objects"));

}

m\_value.object->insert(first.m\_it.object\_iterator, last.m\_it.object\_iterator);

}

/\*!

@brief updates a JSON object from another object, overwriting existing keys

Inserts all values from JSON object @a j and overwrites existing keys.

@param[in] j JSON object to read values from

@throw type\_error.312 if called on JSON values other than objects; example:

`"cannot use update() with string"`

@complexity O(N\*log(size() + N)), where N is the number of elements to

insert.

@liveexample{The example shows how `update()` is used.,update}

@sa https://docs.python.org/3.6/library/stdtypes.html#dict.update

@since version 3.0.0

\*/

void update(const\_reference j)

{

// implicitly convert null value to an empty object

if (is\_null())

{

m\_type = value\_t::object;

m\_value.object = create<object\_t>();

assert\_invariant();

}

if (JSON\_HEDLEY\_UNLIKELY(not is\_object()))

{

JSON\_THROW(type\_error::create(312, "cannot use update() with " + std::string(type\_name())));

}

if (JSON\_HEDLEY\_UNLIKELY(not j.is\_object()))

{

JSON\_THROW(type\_error::create(312, "cannot use update() with " + std::string(j.type\_name())));

}

for (auto it = j.cbegin(); it != j.cend(); ++it)

{

m\_value.object->operator[](it.key()) = it.value();

}

}

/\*!

@brief updates a JSON object from another object, overwriting existing keys

Inserts all values from from range `[first, last)` and overwrites existing

keys.

@param[in] first begin of the range of elements to insert

@param[in] last end of the range of elements to insert

@throw type\_error.312 if called on JSON values other than objects; example:

`"cannot use update() with string"`

@throw invalid\_iterator.202 if iterator @a first or @a last does does not

point to an object; example: `"iterators first and last must point to

objects"`

@throw invalid\_iterator.210 if @a first and @a last do not belong to the

same JSON value; example: `"iterators do not fit"`

@complexity O(N\*log(size() + N)), where N is the number of elements to

insert.

@liveexample{The example shows how `update()` is used\_\_range.,update}

@sa https://docs.python.org/3.6/library/stdtypes.html#dict.update

@since version 3.0.0

\*/

void update(const\_iterator first, const\_iterator last)

{

// implicitly convert null value to an empty object

if (is\_null())

{

m\_type = value\_t::object;

m\_value.object = create<object\_t>();

assert\_invariant();

}

if (JSON\_HEDLEY\_UNLIKELY(not is\_object()))

{

JSON\_THROW(type\_error::create(312, "cannot use update() with " + std::string(type\_name())));

}

// check if range iterators belong to the same JSON object

if (JSON\_HEDLEY\_UNLIKELY(first.m\_object != last.m\_object))

{

JSON\_THROW(invalid\_iterator::create(210, "iterators do not fit"));

}

// passed iterators must belong to objects

if (JSON\_HEDLEY\_UNLIKELY(not first.m\_object->is\_object()

or not last.m\_object->is\_object()))

{

JSON\_THROW(invalid\_iterator::create(202, "iterators first and last must point to objects"));

}

for (auto it = first; it != last; ++it)

{

m\_value.object->operator[](it.key()) = it.value();

}

}

/\*!

@brief exchanges the values

Exchanges the contents of the JSON value with those of @a other. Does not

invoke any move, copy, or swap operations on individual elements. All

iterators and references remain valid. The past-the-end iterator is

invalidated.

@param[in,out] other JSON value to exchange the contents with

@complexity Constant.

@liveexample{The example below shows how JSON values can be swapped with

`swap()`.,swap\_\_reference}

@since version 1.0.0

\*/

void swap(reference other) noexcept (

std::is\_nothrow\_move\_constructible<value\_t>::value and

std::is\_nothrow\_move\_assignable<value\_t>::value and

std::is\_nothrow\_move\_constructible<json\_value>::value and

std::is\_nothrow\_move\_assignable<json\_value>::value

)

{

std::swap(m\_type, other.m\_type);

std::swap(m\_value, other.m\_value);

assert\_invariant();

}

/\*!

@brief exchanges the values

Exchanges the contents of a JSON array with those of @a other. Does not

invoke any move, copy, or swap operations on individual elements. All

iterators and references remain valid. The past-the-end iterator is

invalidated.

@param[in,out] other array to exchange the contents with

@throw type\_error.310 when JSON value is not an array; example: `"cannot

use swap() with string"`

@complexity Constant.

@liveexample{The example below shows how arrays can be swapped with

`swap()`.,swap\_\_array\_t}

@since version 1.0.0

\*/

void swap(array\_t& other)

{

// swap only works for arrays

if (JSON\_HEDLEY\_LIKELY(is\_array()))

{

std::swap(\*(m\_value.array), other);

}

else

{

JSON\_THROW(type\_error::create(310, "cannot use swap() with " + std::string(type\_name())));

}

}

/\*!

@brief exchanges the values

Exchanges the contents of a JSON object with those of @a other. Does not

invoke any move, copy, or swap operations on individual elements. All

iterators and references remain valid. The past-the-end iterator is

invalidated.

@param[in,out] other object to exchange the contents with

@throw type\_error.310 when JSON value is not an object; example:

`"cannot use swap() with string"`

@complexity Constant.

@liveexample{The example below shows how objects can be swapped with

`swap()`.,swap\_\_object\_t}

@since version 1.0.0

\*/

void swap(object\_t& other)

{

// swap only works for objects

if (JSON\_HEDLEY\_LIKELY(is\_object()))

{

std::swap(\*(m\_value.object), other);

}

else

{

JSON\_THROW(type\_error::create(310, "cannot use swap() with " + std::string(type\_name())));

}

}

/\*!

@brief exchanges the values

Exchanges the contents of a JSON string with those of @a other. Does not

invoke any move, copy, or swap operations on individual elements. All

iterators and references remain valid. The past-the-end iterator is

invalidated.

@param[in,out] other string to exchange the contents with

@throw type\_error.310 when JSON value is not a string; example: `"cannot

use swap() with boolean"`

@complexity Constant.

@liveexample{The example below shows how strings can be swapped with

`swap()`.,swap\_\_string\_t}

@since version 1.0.0

\*/

void swap(string\_t& other)

{

// swap only works for strings

if (JSON\_HEDLEY\_LIKELY(is\_string()))

{

std::swap(\*(m\_value.string), other);

}

else

{

JSON\_THROW(type\_error::create(310, "cannot use swap() with " + std::string(type\_name())));

}

}

/// @}

public:

//////////////////////////////////////////

// lexicographical comparison operators //

//////////////////////////////////////////

/// @name lexicographical comparison operators

/// @{

/\*!

@brief comparison: equal

Compares two JSON values for equality according to the following rules:

- Two JSON values are equal if (1) they are from the same type and (2)

their stored values are the same according to their respective

`operator==`.

- Integer and floating-point numbers are automatically converted before

comparison. Note than two NaN values are always treated as unequal.

- Two JSON null values are equal.

@note Floating-point inside JSON values numbers are compared with

`json::number\_float\_t::operator==` which is `double::operator==` by

default. To compare floating-point while respecting an epsilon, an alternative

[comparison function](https://github.com/mariokonrad/marnav/blob/master/src/marnav/math/floatingpoint.hpp#L34-#L39)

could be used, for instance

@code {.cpp}

template<typename T, typename = typename std::enable\_if<std::is\_floating\_point<T>::value, T>::type>

inline bool is\_same(T a, T b, T epsilon = std::numeric\_limits<T>::epsilon()) noexcept

{

return std::abs(a - b) <= epsilon;

}

@endcode

@note NaN values never compare equal to themselves or to other NaN values.

@param[in] lhs first JSON value to consider

@param[in] rhs second JSON value to consider

@return whether the values @a lhs and @a rhs are equal

@exceptionsafety No-throw guarantee: this function never throws exceptions.

@complexity Linear.

@liveexample{The example demonstrates comparing several JSON

types.,operator\_\_equal}

@since version 1.0.0

\*/

friend bool operator==(const\_reference lhs, const\_reference rhs) noexcept

{

const auto lhs\_type = lhs.type();

const auto rhs\_type = rhs.type();

if (lhs\_type == rhs\_type)

{

switch (lhs\_type)

{

case value\_t::array:

return \*lhs.m\_value.array == \*rhs.m\_value.array;

case value\_t::object:

return \*lhs.m\_value.object == \*rhs.m\_value.object;

case value\_t::null:

return true;

case value\_t::string:

return \*lhs.m\_value.string == \*rhs.m\_value.string;

case value\_t::boolean:

return lhs.m\_value.boolean == rhs.m\_value.boolean;

case value\_t::number\_integer:

return lhs.m\_value.number\_integer == rhs.m\_value.number\_integer;

case value\_t::number\_unsigned:

return lhs.m\_value.number\_unsigned == rhs.m\_value.number\_unsigned;

case value\_t::number\_float:

return lhs.m\_value.number\_float == rhs.m\_value.number\_float;

default:

return false;

}

}

else if (lhs\_type == value\_t::number\_integer and rhs\_type == value\_t::number\_float)

{

return static\_cast<number\_float\_t>(lhs.m\_value.number\_integer) == rhs.m\_value.number\_float;

}

else if (lhs\_type == value\_t::number\_float and rhs\_type == value\_t::number\_integer)

{

return lhs.m\_value.number\_float == static\_cast<number\_float\_t>(rhs.m\_value.number\_integer);

}

else if (lhs\_type == value\_t::number\_unsigned and rhs\_type == value\_t::number\_float)

{

return static\_cast<number\_float\_t>(lhs.m\_value.number\_unsigned) == rhs.m\_value.number\_float;

}

else if (lhs\_type == value\_t::number\_float and rhs\_type == value\_t::number\_unsigned)

{

return lhs.m\_value.number\_float == static\_cast<number\_float\_t>(rhs.m\_value.number\_unsigned);

}

else if (lhs\_type == value\_t::number\_unsigned and rhs\_type == value\_t::number\_integer)

{

return static\_cast<number\_integer\_t>(lhs.m\_value.number\_unsigned) == rhs.m\_value.number\_integer;

}

else if (lhs\_type == value\_t::number\_integer and rhs\_type == value\_t::number\_unsigned)

{

return lhs.m\_value.number\_integer == static\_cast<number\_integer\_t>(rhs.m\_value.number\_unsigned);

}

return false;

}

/\*!

@brief comparison: equal

@copydoc operator==(const\_reference, const\_reference)

\*/

template<typename ScalarType, typename std::enable\_if<

std::is\_scalar<ScalarType>::value, int>::type = 0>

friend bool operator==(const\_reference lhs, const ScalarType rhs) noexcept

{

return lhs == basic\_json(rhs);

}

/\*!

@brief comparison: equal

@copydoc operator==(const\_reference, const\_reference)

\*/

template<typename ScalarType, typename std::enable\_if<

std::is\_scalar<ScalarType>::value, int>::type = 0>

friend bool operator==(const ScalarType lhs, const\_reference rhs) noexcept

{

return basic\_json(lhs) == rhs;

}

/\*!

@brief comparison: not equal

Compares two JSON values for inequality by calculating `not (lhs == rhs)`.

@param[in] lhs first JSON value to consider

@param[in] rhs second JSON value to consider

@return whether the values @a lhs and @a rhs are not equal

@complexity Linear.

@exceptionsafety No-throw guarantee: this function never throws exceptions.

@liveexample{The example demonstrates comparing several JSON

types.,operator\_\_notequal}

@since version 1.0.0

\*/

friend bool operator!=(const\_reference lhs, const\_reference rhs) noexcept

{

return not (lhs == rhs);

}

/\*!

@brief comparison: not equal

@copydoc operator!=(const\_reference, const\_reference)

\*/

template<typename ScalarType, typename std::enable\_if<

std::is\_scalar<ScalarType>::value, int>::type = 0>

friend bool operator!=(const\_reference lhs, const ScalarType rhs) noexcept

{

return lhs != basic\_json(rhs);

}

/\*!

@brief comparison: not equal

@copydoc operator!=(const\_reference, const\_reference)

\*/

template<typename ScalarType, typename std::enable\_if<

std::is\_scalar<ScalarType>::value, int>::type = 0>

friend bool operator!=(const ScalarType lhs, const\_reference rhs) noexcept

{

return basic\_json(lhs) != rhs;

}

/\*!

@brief comparison: less than

Compares whether one JSON value @a lhs is less than another JSON value @a

rhs according to the following rules:

- If @a lhs and @a rhs have the same type, the values are compared using

the default `<` operator.

- Integer and floating-point numbers are automatically converted before

comparison

- In case @a lhs and @a rhs have different types, the values are ignored

and the order of the types is considered, see

@ref operator<(const value\_t, const value\_t).

@param[in] lhs first JSON value to consider

@param[in] rhs second JSON value to consider

@return whether @a lhs is less than @a rhs

@complexity Linear.

@exceptionsafety No-throw guarantee: this function never throws exceptions.

@liveexample{The example demonstrates comparing several JSON

types.,operator\_\_less}

@since version 1.0.0

\*/

friend bool operator<(const\_reference lhs, const\_reference rhs) noexcept

{

const auto lhs\_type = lhs.type();

const auto rhs\_type = rhs.type();

if (lhs\_type == rhs\_type)

{

switch (lhs\_type)

{

case value\_t::array:

// note parentheses are necessary, see

// https://github.com/nlohmann/json/issues/1530

return (\*lhs.m\_value.array) < (\*rhs.m\_value.array);

case value\_t::object:

return (\*lhs.m\_value.object) < (\*rhs.m\_value.object);

case value\_t::null:

return false;

case value\_t::string:

return (\*lhs.m\_value.string) < (\*rhs.m\_value.string);

case value\_t::boolean:

return (lhs.m\_value.boolean) < (rhs.m\_value.boolean);

case value\_t::number\_integer:

return (lhs.m\_value.number\_integer) < (rhs.m\_value.number\_integer);

case value\_t::number\_unsigned:

return (lhs.m\_value.number\_unsigned) < (rhs.m\_value.number\_unsigned);

case value\_t::number\_float:

return (lhs.m\_value.number\_float) < (rhs.m\_value.number\_float);

default:

return false;

}

}

else if (lhs\_type == value\_t::number\_integer and rhs\_type == value\_t::number\_float)

{

return static\_cast<number\_float\_t>(lhs.m\_value.number\_integer) < rhs.m\_value.number\_float;

}

else if (lhs\_type == value\_t::number\_float and rhs\_type == value\_t::number\_integer)

{

return lhs.m\_value.number\_float < static\_cast<number\_float\_t>(rhs.m\_value.number\_integer);

}

else if (lhs\_type == value\_t::number\_unsigned and rhs\_type == value\_t::number\_float)

{

return static\_cast<number\_float\_t>(lhs.m\_value.number\_unsigned) < rhs.m\_value.number\_float;

}

else if (lhs\_type == value\_t::number\_float and rhs\_type == value\_t::number\_unsigned)

{

return lhs.m\_value.number\_float < static\_cast<number\_float\_t>(rhs.m\_value.number\_unsigned);

}

else if (lhs\_type == value\_t::number\_integer and rhs\_type == value\_t::number\_unsigned)

{

return lhs.m\_value.number\_integer < static\_cast<number\_integer\_t>(rhs.m\_value.number\_unsigned);

}

else if (lhs\_type == value\_t::number\_unsigned and rhs\_type == value\_t::number\_integer)

{

return static\_cast<number\_integer\_t>(lhs.m\_value.number\_unsigned) < rhs.m\_value.number\_integer;

}

// We only reach this line if we cannot compare values. In that case,

// we compare types. Note we have to call the operator explicitly,

// because MSVC has problems otherwise.

return operator<(lhs\_type, rhs\_type);

}

/\*!

@brief comparison: less than

@copydoc operator<(const\_reference, const\_reference)

\*/

template<typename ScalarType, typename std::enable\_if<

std::is\_scalar<ScalarType>::value, int>::type = 0>

friend bool operator<(const\_reference lhs, const ScalarType rhs) noexcept

{

return lhs < basic\_json(rhs);

}

/\*!

@brief comparison: less than

@copydoc operator<(const\_reference, const\_reference)

\*/

template<typename ScalarType, typename std::enable\_if<

std::is\_scalar<ScalarType>::value, int>::type = 0>

friend bool operator<(const ScalarType lhs, const\_reference rhs) noexcept

{

return basic\_json(lhs) < rhs;

}

/\*!

@brief comparison: less than or equal

Compares whether one JSON value @a lhs is less than or equal to another

JSON value by calculating `not (rhs < lhs)`.

@param[in] lhs first JSON value to consider

@param[in] rhs second JSON value to consider

@return whether @a lhs is less than or equal to @a rhs

@complexity Linear.

@exceptionsafety No-throw guarantee: this function never throws exceptions.

@liveexample{The example demonstrates comparing several JSON

types.,operator\_\_greater}

@since version 1.0.0

\*/

friend bool operator<=(const\_reference lhs, const\_reference rhs) noexcept

{

return not (rhs < lhs);

}

/\*!

@brief comparison: less than or equal

@copydoc operator<=(const\_reference, const\_reference)

\*/

template<typename ScalarType, typename std::enable\_if<

std::is\_scalar<ScalarType>::value, int>::type = 0>

friend bool operator<=(const\_reference lhs, const ScalarType rhs) noexcept

{

return lhs <= basic\_json(rhs);

}

/\*!

@brief comparison: less than or equal

@copydoc operator<=(const\_reference, const\_reference)

\*/

template<typename ScalarType, typename std::enable\_if<

std::is\_scalar<ScalarType>::value, int>::type = 0>

friend bool operator<=(const ScalarType lhs, const\_reference rhs) noexcept

{

return basic\_json(lhs) <= rhs;

}

/\*!

@brief comparison: greater than

Compares whether one JSON value @a lhs is greater than another

JSON value by calculating `not (lhs <= rhs)`.

@param[in] lhs first JSON value to consider

@param[in] rhs second JSON value to consider

@return whether @a lhs is greater than to @a rhs

@complexity Linear.

@exceptionsafety No-throw guarantee: this function never throws exceptions.

@liveexample{The example demonstrates comparing several JSON

types.,operator\_\_lessequal}

@since version 1.0.0

\*/

friend bool operator>(const\_reference lhs, const\_reference rhs) noexcept

{

return not (lhs <= rhs);

}

/\*!

@brief comparison: greater than

@copydoc operator>(const\_reference, const\_reference)

\*/

template<typename ScalarType, typename std::enable\_if<

std::is\_scalar<ScalarType>::value, int>::type = 0>

friend bool operator>(const\_reference lhs, const ScalarType rhs) noexcept

{

return lhs > basic\_json(rhs);

}

/\*!

@brief comparison: greater than

@copydoc operator>(const\_reference, const\_reference)

\*/

template<typename ScalarType, typename std::enable\_if<

std::is\_scalar<ScalarType>::value, int>::type = 0>

friend bool operator>(const ScalarType lhs, const\_reference rhs) noexcept

{

return basic\_json(lhs) > rhs;

}

/\*!

@brief comparison: greater than or equal

Compares whether one JSON value @a lhs is greater than or equal to another

JSON value by calculating `not (lhs < rhs)`.

@param[in] lhs first JSON value to consider

@param[in] rhs second JSON value to consider

@return whether @a lhs is greater than or equal to @a rhs

@complexity Linear.

@exceptionsafety No-throw guarantee: this function never throws exceptions.

@liveexample{The example demonstrates comparing several JSON

types.,operator\_\_greaterequal}

@since version 1.0.0

\*/

friend bool operator>=(const\_reference lhs, const\_reference rhs) noexcept

{

return not (lhs < rhs);

}

/\*!

@brief comparison: greater than or equal

@copydoc operator>=(const\_reference, const\_reference)

\*/

template<typename ScalarType, typename std::enable\_if<

std::is\_scalar<ScalarType>::value, int>::type = 0>

friend bool operator>=(const\_reference lhs, const ScalarType rhs) noexcept

{

return lhs >= basic\_json(rhs);

}

/\*!

@brief comparison: greater than or equal

@copydoc operator>=(const\_reference, const\_reference)

\*/

template<typename ScalarType, typename std::enable\_if<

std::is\_scalar<ScalarType>::value, int>::type = 0>

friend bool operator>=(const ScalarType lhs, const\_reference rhs) noexcept

{

return basic\_json(lhs) >= rhs;

}

/// @}

///////////////////

// serialization //

///////////////////

/// @name serialization

/// @{

/\*!

@brief serialize to stream

Serialize the given JSON value @a j to the output stream @a o. The JSON

value will be serialized using the @ref dump member function.

- The indentation of the output can be controlled with the member variable

`width` of the output stream @a o. For instance, using the manipulator

`std::setw(4)` on @a o sets the indentation level to `4` and the

serialization result is the same as calling `dump(4)`.

- The indentation character can be controlled with the member variable

`fill` of the output stream @a o. For instance, the manipulator

`std::setfill('\\t')` sets indentation to use a tab character rather than

the default space character.

@param[in,out] o stream to serialize to

@param[in] j JSON value to serialize

@return the stream @a o

@throw type\_error.316 if a string stored inside the JSON value is not

UTF-8 encoded

@complexity Linear.

@liveexample{The example below shows the serialization with different

parameters to `width` to adjust the indentation level.,operator\_serialize}

@since version 1.0.0; indentation character added in version 3.0.0

\*/

friend std::ostream& operator<<(std::ostream& o, const basic\_json& j)

{

// read width member and use it as indentation parameter if nonzero

const bool pretty\_print = o.width() > 0;

const auto indentation = pretty\_print ? o.width() : 0;

// reset width to 0 for subsequent calls to this stream

o.width(0);

// do the actual serialization

serializer s(detail::output\_adapter<char>(o), o.fill());

s.dump(j, pretty\_print, false, static\_cast<unsigned int>(indentation));

return o;

}

/\*!

@brief serialize to stream

@deprecated This stream operator is deprecated and will be removed in

future 4.0.0 of the library. Please use

@ref operator<<(std::ostream&, const basic\_json&)

instead; that is, replace calls like `j >> o;` with `o << j;`.

@since version 1.0.0; deprecated since version 3.0.0

\*/

JSON\_HEDLEY\_DEPRECATED(3.0.0)

friend std::ostream& operator>>(const basic\_json& j, std::ostream& o)

{

return o << j;

}

/// @}

/////////////////////

// deserialization //

/////////////////////

/// @name deserialization

/// @{

/\*!

@brief deserialize from a compatible input

This function reads from a compatible input. Examples are:

- an array of 1-byte values

- strings with character/literal type with size of 1 byte

- input streams

- container with contiguous storage of 1-byte values. Compatible container

types include `std::vector`, `std::string`, `std::array`,

`std::valarray`, and `std::initializer\_list`. Furthermore, C-style

arrays can be used with `std::begin()`/`std::end()`. User-defined

containers can be used as long as they implement random-access iterators

and a contiguous storage.

@pre Each element of the container has a size of 1 byte. Violating this

precondition yields undefined behavior. \*\*This precondition is enforced

with a static assertion.\*\*

@pre The container storage is contiguous. Violating this precondition

yields undefined behavior. \*\*This precondition is enforced with an

assertion.\*\*

@warning There is no way to enforce all preconditions at compile-time. If

the function is called with a noncompliant container and with

assertions switched off, the behavior is undefined and will most

likely yield segmentation violation.

@param[in] i input to read from

@param[in] cb a parser callback function of type @ref parser\_callback\_t

which is used to control the deserialization by filtering unwanted values

(optional)

@param[in] allow\_exceptions whether to throw exceptions in case of a

parse error (optional, true by default)

@return deserialized JSON value; in case of a parse error and

@a allow\_exceptions set to `false`, the return value will be

value\_t::discarded.

@throw parse\_error.101 if a parse error occurs; example: `""unexpected end

of input; expected string literal""`

@throw parse\_error.102 if to\_unicode fails or surrogate error

@throw parse\_error.103 if to\_unicode fails

@complexity Linear in the length of the input. The parser is a predictive

LL(1) parser. The complexity can be higher if the parser callback function

@a cb has a super-linear complexity.

@note A UTF-8 byte order mark is silently ignored.

@liveexample{The example below demonstrates the `parse()` function reading

from an array.,parse\_\_array\_\_parser\_callback\_t}

@liveexample{The example below demonstrates the `parse()` function with

and without callback function.,parse\_\_string\_\_parser\_callback\_t}

@liveexample{The example below demonstrates the `parse()` function with

and without callback function.,parse\_\_istream\_\_parser\_callback\_t}

@liveexample{The example below demonstrates the `parse()` function reading

from a contiguous container.,parse\_\_contiguouscontainer\_\_parser\_callback\_t}

@since version 2.0.3 (contiguous containers)

\*/

JSON\_HEDLEY\_WARN\_UNUSED\_RESULT

static basic\_json parse(detail::input\_adapter&& i,

const parser\_callback\_t cb = nullptr,

const bool allow\_exceptions = true)

{

basic\_json result;

parser(i, cb, allow\_exceptions).parse(true, result);

return result;

}

static bool accept(detail::input\_adapter&& i)

{

return parser(i).accept(true);

}

/\*!

@brief generate SAX events

The SAX event lister must follow the interface of @ref json\_sax.

This function reads from a compatible input. Examples are:

- an array of 1-byte values

- strings with character/literal type with size of 1 byte

- input streams

- container with contiguous storage of 1-byte values. Compatible container

types include `std::vector`, `std::string`, `std::array`,

`std::valarray`, and `std::initializer\_list`. Furthermore, C-style

arrays can be used with `std::begin()`/`std::end()`. User-defined

containers can be used as long as they implement random-access iterators

and a contiguous storage.

@pre Each element of the container has a size of 1 byte. Violating this

precondition yields undefined behavior. \*\*This precondition is enforced

with a static assertion.\*\*

@pre The container storage is contiguous. Violating this precondition

yields undefined behavior. \*\*This precondition is enforced with an

assertion.\*\*

@warning There is no way to enforce all preconditions at compile-time. If

the function is called with a noncompliant container and with

assertions switched off, the behavior is undefined and will most

likely yield segmentation violation.

@param[in] i input to read from

@param[in,out] sax SAX event listener

@param[in] format the format to parse (JSON, CBOR, MessagePack, or UBJSON)

@param[in] strict whether the input has to be consumed completely

@return return value of the last processed SAX event

@throw parse\_error.101 if a parse error occurs; example: `""unexpected end

of input; expected string literal""`

@throw parse\_error.102 if to\_unicode fails or surrogate error

@throw parse\_error.103 if to\_unicode fails

@complexity Linear in the length of the input. The parser is a predictive

LL(1) parser. The complexity can be higher if the SAX consumer @a sax has

a super-linear complexity.

@note A UTF-8 byte order mark is silently ignored.

@liveexample{The example below demonstrates the `sax\_parse()` function

reading from string and processing the events with a user-defined SAX

event consumer.,sax\_parse}

@since version 3.2.0

\*/

template <typename SAX>

JSON\_HEDLEY\_NON\_NULL(2)

static bool sax\_parse(detail::input\_adapter&& i, SAX\* sax,

input\_format\_t format = input\_format\_t::json,

const bool strict = true)

{

assert(sax);

return format == input\_format\_t::json

? parser(std::move(i)).sax\_parse(sax, strict)

: detail::binary\_reader<basic\_json, SAX>(std::move(i)).sax\_parse(format, sax, strict);

}

/\*!

@brief deserialize from an iterator range with contiguous storage

This function reads from an iterator range of a container with contiguous

storage of 1-byte values. Compatible container types include

`std::vector`, `std::string`, `std::array`, `std::valarray`, and

`std::initializer\_list`. Furthermore, C-style arrays can be used with

`std::begin()`/`std::end()`. User-defined containers can be used as long

as they implement random-access iterators and a contiguous storage.

@pre The iterator range is contiguous. Violating this precondition yields

undefined behavior. \*\*This precondition is enforced with an assertion.\*\*

@pre Each element in the range has a size of 1 byte. Violating this

precondition yields undefined behavior. \*\*This precondition is enforced

with a static assertion.\*\*

@warning There is no way to enforce all preconditions at compile-time. If

the function is called with noncompliant iterators and with

assertions switched off, the behavior is undefined and will most

likely yield segmentation violation.

@tparam IteratorType iterator of container with contiguous storage

@param[in] first begin of the range to parse (included)

@param[in] last end of the range to parse (excluded)

@param[in] cb a parser callback function of type @ref parser\_callback\_t

which is used to control the deserialization by filtering unwanted values

(optional)

@param[in] allow\_exceptions whether to throw exceptions in case of a

parse error (optional, true by default)

@return deserialized JSON value; in case of a parse error and

@a allow\_exceptions set to `false`, the return value will be

value\_t::discarded.

@throw parse\_error.101 in case of an unexpected token

@throw parse\_error.102 if to\_unicode fails or surrogate error

@throw parse\_error.103 if to\_unicode fails

@complexity Linear in the length of the input. The parser is a predictive

LL(1) parser. The complexity can be higher if the parser callback function

@a cb has a super-linear complexity.

@note A UTF-8 byte order mark is silently ignored.

@liveexample{The example below demonstrates the `parse()` function reading

from an iterator range.,parse\_\_iteratortype\_\_parser\_callback\_t}

@since version 2.0.3

\*/

template<class IteratorType, typename std::enable\_if<

std::is\_base\_of<

std::random\_access\_iterator\_tag,

typename std::iterator\_traits<IteratorType>::iterator\_category>::value, int>::type = 0>

static basic\_json parse(IteratorType first, IteratorType last,

const parser\_callback\_t cb = nullptr,

const bool allow\_exceptions = true)

{

basic\_json result;

parser(detail::input\_adapter(first, last), cb, allow\_exceptions).parse(true, result);

return result;

}

template<class IteratorType, typename std::enable\_if<

std::is\_base\_of<

std::random\_access\_iterator\_tag,

typename std::iterator\_traits<IteratorType>::iterator\_category>::value, int>::type = 0>

static bool accept(IteratorType first, IteratorType last)

{

return parser(detail::input\_adapter(first, last)).accept(true);

}

template<class IteratorType, class SAX, typename std::enable\_if<

std::is\_base\_of<

std::random\_access\_iterator\_tag,

typename std::iterator\_traits<IteratorType>::iterator\_category>::value, int>::type = 0>

JSON\_HEDLEY\_NON\_NULL(3)

static bool sax\_parse(IteratorType first, IteratorType last, SAX\* sax)

{

return parser(detail::input\_adapter(first, last)).sax\_parse(sax);

}

/\*!

@brief deserialize from stream

@deprecated This stream operator is deprecated and will be removed in

version 4.0.0 of the library. Please use

@ref operator>>(std::istream&, basic\_json&)

instead; that is, replace calls like `j << i;` with `i >> j;`.

@since version 1.0.0; deprecated since version 3.0.0

\*/

JSON\_HEDLEY\_DEPRECATED(3.0.0)

friend std::istream& operator<<(basic\_json& j, std::istream& i)

{

return operator>>(i, j);

}

/\*!

@brief deserialize from stream

Deserializes an input stream to a JSON value.

@param[in,out] i input stream to read a serialized JSON value from

@param[in,out] j JSON value to write the deserialized input to

@throw parse\_error.101 in case of an unexpected token

@throw parse\_error.102 if to\_unicode fails or surrogate error

@throw parse\_error.103 if to\_unicode fails

@complexity Linear in the length of the input. The parser is a predictive

LL(1) parser.

@note A UTF-8 byte order mark is silently ignored.

@liveexample{The example below shows how a JSON value is constructed by

reading a serialization from a stream.,operator\_deserialize}

@sa parse(std::istream&, const parser\_callback\_t) for a variant with a

parser callback function to filter values while parsing

@since version 1.0.0

\*/

friend std::istream& operator>>(std::istream& i, basic\_json& j)

{

parser(detail::input\_adapter(i)).parse(false, j);

return i;

}

/// @}

///////////////////////////

// convenience functions //

///////////////////////////

/\*!

@brief return the type as string

Returns the type name as string to be used in error messages - usually to

indicate that a function was called on a wrong JSON type.

@return a string representation of a the @a m\_type member:

Value type | return value

----------- | -------------

null | `"null"`

boolean | `"boolean"`

string | `"string"`

number | `"number"` (for all number types)

object | `"object"`

array | `"array"`

discarded | `"discarded"`

@exceptionsafety No-throw guarantee: this function never throws exceptions.

@complexity Constant.

@liveexample{The following code exemplifies `type\_name()` for all JSON

types.,type\_name}

@sa @ref type() -- return the type of the JSON value

@sa @ref operator value\_t() -- return the type of the JSON value (implicit)

@since version 1.0.0, public since 2.1.0, `const char\*` and `noexcept`

since 3.0.0

\*/

JSON\_HEDLEY\_RETURNS\_NON\_NULL

const char\* type\_name() const noexcept

{

{

switch (m\_type)

{

case value\_t::null:

return "null";

case value\_t::object:

return "object";

case value\_t::array:

return "array";

case value\_t::string:

return "string";

case value\_t::boolean:

return "boolean";

case value\_t::discarded:

return "discarded";

default:

return "number";

}

}

}

private:

//////////////////////

// member variables //

//////////////////////

/// the type of the current element

value\_t m\_type = value\_t::null;

/// the value of the current element

json\_value m\_value = {};

//////////////////////////////////////////

// binary serialization/deserialization //

//////////////////////////////////////////

/// @name binary serialization/deserialization support

/// @{

public:

/\*!

@brief create a CBOR serialization of a given JSON value

Serializes a given JSON value @a j to a byte vector using the CBOR (Concise

Binary Object Representation) serialization format. CBOR is a binary

serialization format which aims to be more compact than JSON itself, yet

more efficient to parse.

The library uses the following mapping from JSON values types to

CBOR types according to the CBOR specification (RFC 7049):

JSON value type | value/range | CBOR type | first byte

--------------- | ------------------------------------------ | ---------------------------------- | ---------------

null | `null` | Null | 0xF6

boolean | `true` | True | 0xF5

boolean | `false` | False | 0xF4

number\_integer | -9223372036854775808..-2147483649 | Negative integer (8 bytes follow) | 0x3B

number\_integer | -2147483648..-32769 | Negative integer (4 bytes follow) | 0x3A

number\_integer | -32768..-129 | Negative integer (2 bytes follow) | 0x39

number\_integer | -128..-25 | Negative integer (1 byte follow) | 0x38

number\_integer | -24..-1 | Negative integer | 0x20..0x37

number\_integer | 0..23 | Integer | 0x00..0x17

number\_integer | 24..255 | Unsigned integer (1 byte follow) | 0x18

number\_integer | 256..65535 | Unsigned integer (2 bytes follow) | 0x19

number\_integer | 65536..4294967295 | Unsigned integer (4 bytes follow) | 0x1A

number\_integer | 4294967296..18446744073709551615 | Unsigned integer (8 bytes follow) | 0x1B

number\_unsigned | 0..23 | Integer | 0x00..0x17

number\_unsigned | 24..255 | Unsigned integer (1 byte follow) | 0x18

number\_unsigned | 256..65535 | Unsigned integer (2 bytes follow) | 0x19

number\_unsigned | 65536..4294967295 | Unsigned integer (4 bytes follow) | 0x1A

number\_unsigned | 4294967296..18446744073709551615 | Unsigned integer (8 bytes follow) | 0x1B

number\_float | \*any value\* | Double-Precision Float | 0xFB

string | \*length\*: 0..23 | UTF-8 string | 0x60..0x77

string | \*length\*: 23..255 | UTF-8 string (1 byte follow) | 0x78

string | \*length\*: 256..65535 | UTF-8 string (2 bytes follow) | 0x79

string | \*length\*: 65536..4294967295 | UTF-8 string (4 bytes follow) | 0x7A

string | \*length\*: 4294967296..18446744073709551615 | UTF-8 string (8 bytes follow) | 0x7B

array | \*size\*: 0..23 | array | 0x80..0x97

array | \*size\*: 23..255 | array (1 byte follow) | 0x98

array | \*size\*: 256..65535 | array (2 bytes follow) | 0x99

array | \*size\*: 65536..4294967295 | array (4 bytes follow) | 0x9A

array | \*size\*: 4294967296..18446744073709551615 | array (8 bytes follow) | 0x9B

object | \*size\*: 0..23 | map | 0xA0..0xB7

object | \*size\*: 23..255 | map (1 byte follow) | 0xB8

object | \*size\*: 256..65535 | map (2 bytes follow) | 0xB9

object | \*size\*: 65536..4294967295 | map (4 bytes follow) | 0xBA

object | \*size\*: 4294967296..18446744073709551615 | map (8 bytes follow) | 0xBB

@note The mapping is \*\*complete\*\* in the sense that any JSON value type

can be converted to a CBOR value.

@note If NaN or Infinity are stored inside a JSON number, they are

serialized properly. This behavior differs from the @ref dump()

function which serializes NaN or Infinity to `null`.

@note The following CBOR types are not used in the conversion:

- byte strings (0x40..0x5F)

- UTF-8 strings terminated by "break" (0x7F)

- arrays terminated by "break" (0x9F)

- maps terminated by "break" (0xBF)

- date/time (0xC0..0xC1)

- bignum (0xC2..0xC3)

- decimal fraction (0xC4)

- bigfloat (0xC5)

- tagged items (0xC6..0xD4, 0xD8..0xDB)

- expected conversions (0xD5..0xD7)

- simple values (0xE0..0xF3, 0xF8)

- undefined (0xF7)

- half and single-precision floats (0xF9-0xFA)

- break (0xFF)

@param[in] j JSON value to serialize

@return MessagePack serialization as byte vector

@complexity Linear in the size of the JSON value @a j.

@liveexample{The example shows the serialization of a JSON value to a byte

vector in CBOR format.,to\_cbor}

@sa http://cbor.io

@sa @ref from\_cbor(detail::input\_adapter&&, const bool, const bool) for the

analogous deserialization

@sa @ref to\_msgpack(const basic\_json&) for the related MessagePack format

@sa @ref to\_ubjson(const basic\_json&, const bool, const bool) for the

related UBJSON format

@since version 2.0.9

\*/

static std::vector<uint8\_t> to\_cbor(const basic\_json& j)

{

std::vector<uint8\_t> result;

to\_cbor(j, result);

return result;

}

static void to\_cbor(const basic\_json& j, detail::output\_adapter<uint8\_t> o)

{

binary\_writer<uint8\_t>(o).write\_cbor(j);

}

static void to\_cbor(const basic\_json& j, detail::output\_adapter<char> o)

{

binary\_writer<char>(o).write\_cbor(j);

}

/\*!

@brief create a MessagePack serialization of a given JSON value

Serializes a given JSON value @a j to a byte vector using the MessagePack

serialization format. MessagePack is a binary serialization format which

aims to be more compact than JSON itself, yet more efficient to parse.

The library uses the following mapping from JSON values types to

MessagePack types according to the MessagePack specification:

JSON value type | value/range | MessagePack type | first byte

--------------- | --------------------------------- | ---------------- | ----------

null | `null` | nil | 0xC0

boolean | `true` | true | 0xC3

boolean | `false` | false | 0xC2

number\_integer | -9223372036854775808..-2147483649 | int64 | 0xD3

number\_integer | -2147483648..-32769 | int32 | 0xD2

number\_integer | -32768..-129 | int16 | 0xD1

number\_integer | -128..-33 | int8 | 0xD0

number\_integer | -32..-1 | negative fixint | 0xE0..0xFF

number\_integer | 0..127 | positive fixint | 0x00..0x7F

number\_integer | 128..255 | uint 8 | 0xCC

number\_integer | 256..65535 | uint 16 | 0xCD

number\_integer | 65536..4294967295 | uint 32 | 0xCE

number\_integer | 4294967296..18446744073709551615 | uint 64 | 0xCF

number\_unsigned | 0..127 | positive fixint | 0x00..0x7F

number\_unsigned | 128..255 | uint 8 | 0xCC

number\_unsigned | 256..65535 | uint 16 | 0xCD

number\_unsigned | 65536..4294967295 | uint 32 | 0xCE

number\_unsigned | 4294967296..18446744073709551615 | uint 64 | 0xCF

number\_float | \*any value\* | float 64 | 0xCB

string | \*length\*: 0..31 | fixstr | 0xA0..0xBF

string | \*length\*: 32..255 | str 8 | 0xD9

string | \*length\*: 256..65535 | str 16 | 0xDA

string | \*length\*: 65536..4294967295 | str 32 | 0xDB

array | \*size\*: 0..15 | fixarray | 0x90..0x9F

array | \*size\*: 16..65535 | array 16 | 0xDC

array | \*size\*: 65536..4294967295 | array 32 | 0xDD

object | \*size\*: 0..15 | fix map | 0x80..0x8F

object | \*size\*: 16..65535 | map 16 | 0xDE

object | \*size\*: 65536..4294967295 | map 32 | 0xDF

@note The mapping is \*\*complete\*\* in the sense that any JSON value type

can be converted to a MessagePack value.

@note The following values can \*\*not\*\* be converted to a MessagePack value:

- strings with more than 4294967295 bytes

- arrays with more than 4294967295 elements

- objects with more than 4294967295 elements

@note The following MessagePack types are not used in the conversion:

- bin 8 - bin 32 (0xC4..0xC6)

- ext 8 - ext 32 (0xC7..0xC9)

- float 32 (0xCA)

- fixext 1 - fixext 16 (0xD4..0xD8)

@note Any MessagePack output created @ref to\_msgpack can be successfully

parsed by @ref from\_msgpack.

@note If NaN or Infinity are stored inside a JSON number, they are

serialized properly. This behavior differs from the @ref dump()

function which serializes NaN or Infinity to `null`.

@param[in] j JSON value to serialize

@return MessagePack serialization as byte vector

@complexity Linear in the size of the JSON value @a j.

@liveexample{The example shows the serialization of a JSON value to a byte

vector in MessagePack format.,to\_msgpack}

@sa http://msgpack.org

@sa @ref from\_msgpack for the analogous deserialization

@sa @ref to\_cbor(const basic\_json& for the related CBOR format

@sa @ref to\_ubjson(const basic\_json&, const bool, const bool) for the

related UBJSON format

@since version 2.0.9

\*/

static std::vector<uint8\_t> to\_msgpack(const basic\_json& j)

{

std::vector<uint8\_t> result;

to\_msgpack(j, result);

return result;

}

static void to\_msgpack(const basic\_json& j, detail::output\_adapter<uint8\_t> o)

{

binary\_writer<uint8\_t>(o).write\_msgpack(j);

}

static void to\_msgpack(const basic\_json& j, detail::output\_adapter<char> o)

{

binary\_writer<char>(o).write\_msgpack(j);

}

/\*!

@brief create a UBJSON serialization of a given JSON value

Serializes a given JSON value @a j to a byte vector using the UBJSON

(Universal Binary JSON) serialization format. UBJSON aims to be more compact

than JSON itself, yet more efficient to parse.

The library uses the following mapping from JSON values types to

UBJSON types according to the UBJSON specification:

JSON value type | value/range | UBJSON type | marker

--------------- | --------------------------------- | ----------- | ------

null | `null` | null | `Z`

boolean | `true` | true | `T`

boolean | `false` | false | `F`

number\_integer | -9223372036854775808..-2147483649 | int64 | `L`

number\_integer | -2147483648..-32769 | int32 | `l`

number\_integer | -32768..-129 | int16 | `I`

number\_integer | -128..127 | int8 | `i`

number\_integer | 128..255 | uint8 | `U`

number\_integer | 256..32767 | int16 | `I`

number\_integer | 32768..2147483647 | int32 | `l`

number\_integer | 2147483648..9223372036854775807 | int64 | `L`

number\_unsigned | 0..127 | int8 | `i`

number\_unsigned | 128..255 | uint8 | `U`

number\_unsigned | 256..32767 | int16 | `I`

number\_unsigned | 32768..2147483647 | int32 | `l`

number\_unsigned | 2147483648..9223372036854775807 | int64 | `L`

number\_float | \*any value\* | float64 | `D`

string | \*with shortest length indicator\* | string | `S`

array | \*see notes on optimized format\* | array | `[`

object | \*see notes on optimized format\* | map | `{`

@note The mapping is \*\*complete\*\* in the sense that any JSON value type

can be converted to a UBJSON value.

@note The following values can \*\*not\*\* be converted to a UBJSON value:

- strings with more than 9223372036854775807 bytes (theoretical)

- unsigned integer numbers above 9223372036854775807

@note The following markers are not used in the conversion:

- `Z`: no-op values are not created.

- `C`: single-byte strings are serialized with `S` markers.

@note Any UBJSON output created @ref to\_ubjson can be successfully parsed

by @ref from\_ubjson.

@note If NaN or Infinity are stored inside a JSON number, they are

serialized properly. This behavior differs from the @ref dump()

function which serializes NaN or Infinity to `null`.

@note The optimized formats for containers are supported: Parameter

@a use\_size adds size information to the beginning of a container and

removes the closing marker. Parameter @a use\_type further checks

whether all elements of a container have the same type and adds the

type marker to the beginning of the container. The @a use\_type

parameter must only be used together with @a use\_size = true. Note

that @a use\_size = true alone may result in larger representations -

the benefit of this parameter is that the receiving side is

immediately informed on the number of elements of the container.

@param[in] j JSON value to serialize

@param[in] use\_size whether to add size annotations to container types

@param[in] use\_type whether to add type annotations to container types

(must be combined with @a use\_size = true)

@return UBJSON serialization as byte vector

@complexity Linear in the size of the JSON value @a j.

@liveexample{The example shows the serialization of a JSON value to a byte

vector in UBJSON format.,to\_ubjson}

@sa http://ubjson.org

@sa @ref from\_ubjson(detail::input\_adapter&&, const bool, const bool) for the

analogous deserialization

@sa @ref to\_cbor(const basic\_json& for the related CBOR format

@sa @ref to\_msgpack(const basic\_json&) for the related MessagePack format

@since version 3.1.0

\*/

static std::vector<uint8\_t> to\_ubjson(const basic\_json& j,

const bool use\_size = false,

const bool use\_type = false)

{

std::vector<uint8\_t> result;

to\_ubjson(j, result, use\_size, use\_type);

return result;

}

static void to\_ubjson(const basic\_json& j, detail::output\_adapter<uint8\_t> o,

const bool use\_size = false, const bool use\_type = false)

{

binary\_writer<uint8\_t>(o).write\_ubjson(j, use\_size, use\_type);

}

static void to\_ubjson(const basic\_json& j, detail::output\_adapter<char> o,

const bool use\_size = false, const bool use\_type = false)

{

binary\_writer<char>(o).write\_ubjson(j, use\_size, use\_type);

}

/\*!

@brief Serializes the given JSON object `j` to BSON and returns a vector

containing the corresponding BSON-representation.

BSON (Binary JSON) is a binary format in which zero or more ordered key/value pairs are

stored as a single entity (a so-called document).

The library uses the following mapping from JSON values types to BSON types:

JSON value type | value/range | BSON type | marker

--------------- | --------------------------------- | ----------- | ------

null | `null` | null | 0x0A

boolean | `true`, `false` | boolean | 0x08

number\_integer | -9223372036854775808..-2147483649 | int64 | 0x12

number\_integer | -2147483648..2147483647 | int32 | 0x10

number\_integer | 2147483648..9223372036854775807 | int64 | 0x12

number\_unsigned | 0..2147483647 | int32 | 0x10

number\_unsigned | 2147483648..9223372036854775807 | int64 | 0x12

number\_unsigned | 9223372036854775808..18446744073709551615| -- | --

number\_float | \*any value\* | double | 0x01

string | \*any value\* | string | 0x02

array | \*any value\* | document | 0x04

object | \*any value\* | document | 0x03

@warning The mapping is \*\*incomplete\*\*, since only JSON-objects (and things

contained therein) can be serialized to BSON.

Also, integers larger than 9223372036854775807 cannot be serialized to BSON,

and the keys may not contain U+0000, since they are serialized a

zero-terminated c-strings.

@throw out\_of\_range.407 if `j.is\_number\_unsigned() && j.get<std::uint64\_t>() > 9223372036854775807`

@throw out\_of\_range.409 if a key in `j` contains a NULL (U+0000)

@throw type\_error.317 if `!j.is\_object()`

@pre The input `j` is required to be an object: `j.is\_object() == true`.

@note Any BSON output created via @ref to\_bson can be successfully parsed

by @ref from\_bson.

@param[in] j JSON value to serialize

@return BSON serialization as byte vector

@complexity Linear in the size of the JSON value @a j.

@liveexample{The example shows the serialization of a JSON value to a byte

vector in BSON format.,to\_bson}

@sa http://bsonspec.org/spec.html

@sa @ref from\_bson(detail::input\_adapter&&, const bool strict) for the

analogous deserialization

@sa @ref to\_ubjson(const basic\_json&, const bool, const bool) for the

related UBJSON format

@sa @ref to\_cbor(const basic\_json&) for the related CBOR format

@sa @ref to\_msgpack(const basic\_json&) for the related MessagePack format

\*/

static std::vector<uint8\_t> to\_bson(const basic\_json& j)

{

std::vector<uint8\_t> result;

to\_bson(j, result);

return result;

}

/\*!

@brief Serializes the given JSON object `j` to BSON and forwards the

corresponding BSON-representation to the given output\_adapter `o`.

@param j The JSON object to convert to BSON.

@param o The output adapter that receives the binary BSON representation.

@pre The input `j` shall be an object: `j.is\_object() == true`

@sa @ref to\_bson(const basic\_json&)

\*/

static void to\_bson(const basic\_json& j, detail::output\_adapter<uint8\_t> o)

{

binary\_writer<uint8\_t>(o).write\_bson(j);

}

/\*!

@copydoc to\_bson(const basic\_json&, detail::output\_adapter<uint8\_t>)

\*/

static void to\_bson(const basic\_json& j, detail::output\_adapter<char> o)

{

binary\_writer<char>(o).write\_bson(j);

}

/\*!

@brief create a JSON value from an input in CBOR format

Deserializes a given input @a i to a JSON value using the CBOR (Concise

Binary Object Representation) serialization format.

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

CBOR type | JSON value type | first byte

---------------------- | --------------- | ----------

Integer | number\_unsigned | 0x00..0x17

Unsigned integer | number\_unsigned | 0x18

Unsigned integer | number\_unsigned | 0x19

Unsigned integer | number\_unsigned | 0x1A

Unsigned integer | number\_unsigned | 0x1B

Negative integer | number\_integer | 0x20..0x37

Negative integer | number\_integer | 0x38

Negative integer | number\_integer | 0x39

Negative integer | number\_integer | 0x3A

Negative integer | number\_integer | 0x3B

Negative integer | number\_integer | 0x40..0x57

UTF-8 string | string | 0x60..0x77

UTF-8 string | string | 0x78

UTF-8 string | string | 0x79

UTF-8 string | string | 0x7A

UTF-8 string | string | 0x7B

UTF-8 string | string | 0x7F

array | array | 0x80..0x97

array | array | 0x98

array | array | 0x99

array | array | 0x9A

array | array | 0x9B

array | array | 0x9F

map | object | 0xA0..0xB7

map | object | 0xB8

map | object | 0xB9

map | object | 0xBA

map | object | 0xBB

map | object | 0xBF

False | `false` | 0xF4

True | `true` | 0xF5

Null | `null` | 0xF6

Half-Precision Float | number\_float | 0xF9

Single-Precision Float | number\_float | 0xFA

Double-Precision Float | number\_float | 0xFB

@warning The mapping is \*\*incomplete\*\* in the sense that not all CBOR

types can be converted to a JSON value. The following CBOR types

are not supported and will yield parse errors (parse\_error.112):

- byte strings (0x40..0x5F)

- date/time (0xC0..0xC1)

- bignum (0xC2..0xC3)

- decimal fraction (0xC4)

- bigfloat (0xC5)

- tagged items (0xC6..0xD4, 0xD8..0xDB)

- expected conversions (0xD5..0xD7)

- simple values (0xE0..0xF3, 0xF8)

- undefined (0xF7)

@warning CBOR allows map keys of any type, whereas JSON only allows

strings as keys in object values. Therefore, CBOR maps with keys

other than UTF-8 strings are rejected (parse\_error.113).

@note Any CBOR output created @ref to\_cbor can be successfully parsed by

@ref from\_cbor.

@param[in] i an input in CBOR format convertible to an input adapter

@param[in] strict whether to expect the input to be consumed until EOF

(true by default)

@param[in] allow\_exceptions whether to throw exceptions in case of a

parse error (optional, true by default)

@return deserialized JSON value; in case of a parse error and

@a allow\_exceptions set to `false`, the return value will be

value\_t::discarded.

@throw parse\_error.110 if the given input ends prematurely or the end of

file was not reached when @a strict was set to true

@throw parse\_error.112 if unsupported features from CBOR were

used in the given input @a v or if the input is not valid CBOR

@throw parse\_error.113 if a string was expected as map key, but not found

@complexity Linear in the size of the input @a i.

@liveexample{The example shows the deserialization of a byte vector in CBOR

format to a JSON value.,from\_cbor}

@sa http://cbor.io

@sa @ref to\_cbor(const basic\_json&) for the analogous serialization

@sa @ref from\_msgpack(detail::input\_adapter&&, const bool, const bool) for the

related MessagePack format

@sa @ref from\_ubjson(detail::input\_adapter&&, const bool, const bool) for the

related UBJSON format

@since version 2.0.9; parameter @a start\_index since 2.1.1; changed to

consume input adapters, removed start\_index parameter, and added

@a strict parameter since 3.0.0; added @a allow\_exceptions parameter

since 3.2.0

\*/

JSON\_HEDLEY\_WARN\_UNUSED\_RESULT

static basic\_json from\_cbor(detail::input\_adapter&& i,

const bool strict = true,

const bool allow\_exceptions = true)

{

basic\_json result;

detail::json\_sax\_dom\_parser<basic\_json> sdp(result, allow\_exceptions);

const bool res = binary\_reader(detail::input\_adapter(i)).sax\_parse(input\_format\_t::cbor, &sdp, strict);

return res ? result : basic\_json(value\_t::discarded);

}

/\*!

@copydoc from\_cbor(detail::input\_adapter&&, const bool, const bool)

\*/

template<typename A1, typename A2,

detail::enable\_if\_t<std::is\_constructible<detail::input\_adapter, A1, A2>::value, int> = 0>

JSON\_HEDLEY\_WARN\_UNUSED\_RESULT

static basic\_json from\_cbor(A1 && a1, A2 && a2,

const bool strict = true,

const bool allow\_exceptions = true)

{

basic\_json result;

detail::json\_sax\_dom\_parser<basic\_json> sdp(result, allow\_exceptions);

const bool res = binary\_reader(detail::input\_adapter(std::forward<A1>(a1), std::forward<A2>(a2))).sax\_parse(input\_format\_t::cbor, &sdp, strict);

return res ? result : basic\_json(value\_t::discarded);

}

/\*!

@brief create a JSON value from an input in MessagePack format

Deserializes a given input @a i to a JSON value using the MessagePack

serialization format.

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

MessagePack type | JSON value type | first byte

---------------- | --------------- | ----------

positive fixint | number\_unsigned | 0x00..0x7F

fixmap | object | 0x80..0x8F

fixarray | array | 0x90..0x9F

fixstr | string | 0xA0..0xBF

nil | `null` | 0xC0

false | `false` | 0xC2

true | `true` | 0xC3

float 32 | number\_float | 0xCA

float 64 | number\_float | 0xCB

uint 8 | number\_unsigned | 0xCC

uint 16 | number\_unsigned | 0xCD

uint 32 | number\_unsigned | 0xCE

uint 64 | number\_unsigned | 0xCF

int 8 | number\_integer | 0xD0

int 16 | number\_integer | 0xD1

int 32 | number\_integer | 0xD2

int 64 | number\_integer | 0xD3

str 8 | string | 0xD9

str 16 | string | 0xDA

str 32 | string | 0xDB

array 16 | array | 0xDC

array 32 | array | 0xDD

map 16 | object | 0xDE

map 32 | object | 0xDF

negative fixint | number\_integer | 0xE0-0xFF

@warning The mapping is \*\*incomplete\*\* in the sense that not all

MessagePack types can be converted to a JSON value. The following

MessagePack types are not supported and will yield parse errors:

- bin 8 - bin 32 (0xC4..0xC6)

- ext 8 - ext 32 (0xC7..0xC9)

- fixext 1 - fixext 16 (0xD4..0xD8)

@note Any MessagePack output created @ref to\_msgpack can be successfully

parsed by @ref from\_msgpack.

@param[in] i an input in MessagePack format convertible to an input

adapter

@param[in] strict whether to expect the input to be consumed until EOF

(true by default)

@param[in] allow\_exceptions whether to throw exceptions in case of a

parse error (optional, true by default)

@return deserialized JSON value; in case of a parse error and

@a allow\_exceptions set to `false`, the return value will be

value\_t::discarded.

@throw parse\_error.110 if the given input ends prematurely or the end of

file was not reached when @a strict was set to true

@throw parse\_error.112 if unsupported features from MessagePack were

used in the given input @a i or if the input is not valid MessagePack

@throw parse\_error.113 if a string was expected as map key, but not found

@complexity Linear in the size of the input @a i.

@liveexample{The example shows the deserialization of a byte vector in

MessagePack format to a JSON value.,from\_msgpack}

@sa http://msgpack.org

@sa @ref to\_msgpack(const basic\_json&) for the analogous serialization

@sa @ref from\_cbor(detail::input\_adapter&&, const bool, const bool) for the

related CBOR format

@sa @ref from\_ubjson(detail::input\_adapter&&, const bool, const bool) for

the related UBJSON format

@sa @ref from\_bson(detail::input\_adapter&&, const bool, const bool) for

the related BSON format

@since version 2.0.9; parameter @a start\_index since 2.1.1; changed to

consume input adapters, removed start\_index parameter, and added

@a strict parameter since 3.0.0; added @a allow\_exceptions parameter

since 3.2.0

\*/

JSON\_HEDLEY\_WARN\_UNUSED\_RESULT

static basic\_json from\_msgpack(detail::input\_adapter&& i,

const bool strict = true,

const bool allow\_exceptions = true)

{

basic\_json result;

detail::json\_sax\_dom\_parser<basic\_json> sdp(result, allow\_exceptions);

const bool res = binary\_reader(detail::input\_adapter(i)).sax\_parse(input\_format\_t::msgpack, &sdp, strict);

return res ? result : basic\_json(value\_t::discarded);

}

/\*!

@copydoc from\_msgpack(detail::input\_adapter&&, const bool, const bool)

\*/

template<typename A1, typename A2,

detail::enable\_if\_t<std::is\_constructible<detail::input\_adapter, A1, A2>::value, int> = 0>

JSON\_HEDLEY\_WARN\_UNUSED\_RESULT

static basic\_json from\_msgpack(A1 && a1, A2 && a2,

const bool strict = true,

const bool allow\_exceptions = true)

{

basic\_json result;

detail::json\_sax\_dom\_parser<basic\_json> sdp(result, allow\_exceptions);

const bool res = binary\_reader(detail::input\_adapter(std::forward<A1>(a1), std::forward<A2>(a2))).sax\_parse(input\_format\_t::msgpack, &sdp, strict);

return res ? result : basic\_json(value\_t::discarded);

}

/\*!

@brief create a JSON value from an input in UBJSON format

Deserializes a given input @a i to a JSON value using the UBJSON (Universal

Binary JSON) serialization format.

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

UBJSON type | JSON value type | marker

----------- | --------------------------------------- | ------

no-op | \*no value, next value is read\* | `N`

null | `null` | `Z`

false | `false` | `F`

true | `true` | `T`

float32 | number\_float | `d`

float64 | number\_float | `D`

uint8 | number\_unsigned | `U`

int8 | number\_integer | `i`

int16 | number\_integer | `I`

int32 | number\_integer | `l`

int64 | number\_integer | `L`

string | string | `S`

char | string | `C`

array | array (optimized values are supported) | `[`

object | object (optimized values are supported) | `{`

@note The mapping is \*\*complete\*\* in the sense that any UBJSON value can

be converted to a JSON value.

@param[in] i an input in UBJSON format convertible to an input adapter

@param[in] strict whether to expect the input to be consumed until EOF

(true by default)

@param[in] allow\_exceptions whether to throw exceptions in case of a

parse error (optional, true by default)

@return deserialized JSON value; in case of a parse error and

@a allow\_exceptions set to `false`, the return value will be

value\_t::discarded.

@throw parse\_error.110 if the given input ends prematurely or the end of

file was not reached when @a strict was set to true

@throw parse\_error.112 if a parse error occurs

@throw parse\_error.113 if a string could not be parsed successfully

@complexity Linear in the size of the input @a i.

@liveexample{The example shows the deserialization of a byte vector in

UBJSON format to a JSON value.,from\_ubjson}

@sa http://ubjson.org

@sa @ref to\_ubjson(const basic\_json&, const bool, const bool) for the

analogous serialization

@sa @ref from\_cbor(detail::input\_adapter&&, const bool, const bool) for the

related CBOR format

@sa @ref from\_msgpack(detail::input\_adapter&&, const bool, const bool) for

the related MessagePack format

@sa @ref from\_bson(detail::input\_adapter&&, const bool, const bool) for

the related BSON format

@since version 3.1.0; added @a allow\_exceptions parameter since 3.2.0

\*/

JSON\_HEDLEY\_WARN\_UNUSED\_RESULT

static basic\_json from\_ubjson(detail::input\_adapter&& i,

const bool strict = true,

const bool allow\_exceptions = true)

{

basic\_json result;

detail::json\_sax\_dom\_parser<basic\_json> sdp(result, allow\_exceptions);

const bool res = binary\_reader(detail::input\_adapter(i)).sax\_parse(input\_format\_t::ubjson, &sdp, strict);

return res ? result : basic\_json(value\_t::discarded);

}

/\*!

@copydoc from\_ubjson(detail::input\_adapter&&, const bool, const bool)

\*/

template<typename A1, typename A2,

detail::enable\_if\_t<std::is\_constructible<detail::input\_adapter, A1, A2>::value, int> = 0>

JSON\_HEDLEY\_WARN\_UNUSED\_RESULT

static basic\_json from\_ubjson(A1 && a1, A2 && a2,

const bool strict = true,

const bool allow\_exceptions = true)

{

basic\_json result;

detail::json\_sax\_dom\_parser<basic\_json> sdp(result, allow\_exceptions);

const bool res = binary\_reader(detail::input\_adapter(std::forward<A1>(a1), std::forward<A2>(a2))).sax\_parse(input\_format\_t::ubjson, &sdp, strict);

return res ? result : basic\_json(value\_t::discarded);

}

/\*!

@brief Create a JSON value from an input in BSON format

Deserializes a given input @a i to a JSON value using the BSON (Binary JSON)

serialization format.

The library maps BSON record types to JSON value types as follows:

BSON type | BSON marker byte | JSON value type

--------------- | ---------------- | ---------------------------

double | 0x01 | number\_float

string | 0x02 | string

document | 0x03 | object

array | 0x04 | array

binary | 0x05 | still unsupported

undefined | 0x06 | still unsupported

ObjectId | 0x07 | still unsupported

boolean | 0x08 | boolean

UTC Date-Time | 0x09 | still unsupported

null | 0x0A | null

Regular Expr. | 0x0B | still unsupported

DB Pointer | 0x0C | still unsupported

JavaScript Code | 0x0D | still unsupported

Symbol | 0x0E | still unsupported

JavaScript Code | 0x0F | still unsupported

int32 | 0x10 | number\_integer

Timestamp | 0x11 | still unsupported

128-bit decimal float | 0x13 | still unsupported

Max Key | 0x7F | still unsupported

Min Key | 0xFF | still unsupported

@warning The mapping is \*\*incomplete\*\*. The unsupported mappings

are indicated in the table above.

@param[in] i an input in BSON format convertible to an input adapter

@param[in] strict whether to expect the input to be consumed until EOF

(true by default)

@param[in] allow\_exceptions whether to throw exceptions in case of a

parse error (optional, true by default)

@return deserialized JSON value; in case of a parse error and

@a allow\_exceptions set to `false`, the return value will be

value\_t::discarded.

@throw parse\_error.114 if an unsupported BSON record type is encountered

@complexity Linear in the size of the input @a i.

@liveexample{The example shows the deserialization of a byte vector in

BSON format to a JSON value.,from\_bson}

@sa http://bsonspec.org/spec.html

@sa @ref to\_bson(const basic\_json&) for the analogous serialization

@sa @ref from\_cbor(detail::input\_adapter&&, const bool, const bool) for the

related CBOR format

@sa @ref from\_msgpack(detail::input\_adapter&&, const bool, const bool) for

the related MessagePack format

@sa @ref from\_ubjson(detail::input\_adapter&&, const bool, const bool) for the

related UBJSON format

\*/

JSON\_HEDLEY\_WARN\_UNUSED\_RESULT

static basic\_json from\_bson(detail::input\_adapter&& i,

const bool strict = true,

const bool allow\_exceptions = true)

{

basic\_json result;

detail::json\_sax\_dom\_parser<basic\_json> sdp(result, allow\_exceptions);

const bool res = binary\_reader(detail::input\_adapter(i)).sax\_parse(input\_format\_t::bson, &sdp, strict);

return res ? result : basic\_json(value\_t::discarded);

}

/\*!

@copydoc from\_bson(detail::input\_adapter&&, const bool, const bool)

\*/

template<typename A1, typename A2,

detail::enable\_if\_t<std::is\_constructible<detail::input\_adapter, A1, A2>::value, int> = 0>

JSON\_HEDLEY\_WARN\_UNUSED\_RESULT

static basic\_json from\_bson(A1 && a1, A2 && a2,

const bool strict = true,

const bool allow\_exceptions = true)

{

basic\_json result;

detail::json\_sax\_dom\_parser<basic\_json> sdp(result, allow\_exceptions);

const bool res = binary\_reader(detail::input\_adapter(std::forward<A1>(a1), std::forward<A2>(a2))).sax\_parse(input\_format\_t::bson, &sdp, strict);

return res ? result : basic\_json(value\_t::discarded);

}

/// @}

//////////////////////////

// JSON Pointer support //

//////////////////////////

/// @name JSON Pointer functions

/// @{

/\*!

@brief access specified element via JSON Pointer

Uses a JSON pointer to retrieve a reference to the respective JSON value.

No bound checking is performed. Similar to @ref operator[](const typename

object\_t::key\_type&), `null` values are created in arrays and objects if

necessary.

In particular:

- If the JSON pointer points to an object key that does not exist, it

is created an filled with a `null` value before a reference to it

is returned.

- If the JSON pointer points to an array index that does not exist, it

is created an filled with a `null` value before a reference to it

is returned. All indices between the current maximum and the given

index are also filled with `null`.

- The special value `-` is treated as a synonym for the index past the

end.

@param[in] ptr a JSON pointer

@return reference to the element pointed to by @a ptr

@complexity Constant.

@throw parse\_error.106 if an array index begins with '0'

@throw parse\_error.109 if an array index was not a number

@throw out\_of\_range.404 if the JSON pointer can not be resolved

@liveexample{The behavior is shown in the example.,operatorjson\_pointer}

@since version 2.0.0

\*/

reference operator[](const json\_pointer& ptr)

{

return ptr.get\_unchecked(this);

}

/\*!

@brief access specified element via JSON Pointer

Uses a JSON pointer to retrieve a reference to the respective JSON value.

No bound checking is performed. The function does not change the JSON

value; no `null` values are created. In particular, the the special value

`-` yields an exception.

@param[in] ptr JSON pointer to the desired element

@return const reference to the element pointed to by @a ptr

@complexity Constant.

@throw parse\_error.106 if an array index begins with '0'

@throw parse\_error.109 if an array index was not a number

@throw out\_of\_range.402 if the array index '-' is used

@throw out\_of\_range.404 if the JSON pointer can not be resolved

@liveexample{The behavior is shown in the example.,operatorjson\_pointer\_const}

@since version 2.0.0

\*/

const\_reference operator[](const json\_pointer& ptr) const

{

return ptr.get\_unchecked(this);

}

/\*!

@brief access specified element via JSON Pointer

Returns a reference to the element at with specified JSON pointer @a ptr,

with bounds checking.

@param[in] ptr JSON pointer to the desired element

@return reference to the element pointed to by @a ptr

@throw parse\_error.106 if an array index in the passed JSON pointer @a ptr

begins with '0'. See example below.

@throw parse\_error.109 if an array index in the passed JSON pointer @a ptr

is not a number. See example below.

@throw out\_of\_range.401 if an array index in the passed JSON pointer @a ptr

is out of range. See example below.

@throw out\_of\_range.402 if the array index '-' is used in the passed JSON

pointer @a ptr. As `at` provides checked access (and no elements are

implicitly inserted), the index '-' is always invalid. See example below.

@throw out\_of\_range.403 if the JSON pointer describes a key of an object

which cannot be found. See example below.

@throw out\_of\_range.404 if the JSON pointer @a ptr can not be resolved.

See example below.

@exceptionsafety Strong guarantee: if an exception is thrown, there are no

changes in the JSON value.

@complexity Constant.

@since version 2.0.0

@liveexample{The behavior is shown in the example.,at\_json\_pointer}

\*/

reference at(const json\_pointer& ptr)

{

return ptr.get\_checked(this);

}

/\*!

@brief access specified element via JSON Pointer

Returns a const reference to the element at with specified JSON pointer @a

ptr, with bounds checking.

@param[in] ptr JSON pointer to the desired element

@return reference to the element pointed to by @a ptr

@throw parse\_error.106 if an array index in the passed JSON pointer @a ptr

begins with '0'. See example below.

@throw parse\_error.109 if an array index in the passed JSON pointer @a ptr

is not a number. See example below.

@throw out\_of\_range.401 if an array index in the passed JSON pointer @a ptr

is out of range. See example below.

@throw out\_of\_range.402 if the array index '-' is used in the passed JSON

pointer @a ptr. As `at` provides checked access (and no elements are

implicitly inserted), the index '-' is always invalid. See example below.

@throw out\_of\_range.403 if the JSON pointer describes a key of an object

which cannot be found. See example below.

@throw out\_of\_range.404 if the JSON pointer @a ptr can not be resolved.

See example below.

@exceptionsafety Strong guarantee: if an exception is thrown, there are no

changes in the JSON value.

@complexity Constant.

@since version 2.0.0

@liveexample{The behavior is shown in the example.,at\_json\_pointer\_const}

\*/

const\_reference at(const json\_pointer& ptr) const

{

return ptr.get\_checked(this);

}

/\*!

@brief return flattened JSON value

The function creates a JSON object whose keys are JSON pointers (see [RFC

6901](https://tools.ietf.org/html/rfc6901)) and whose values are all

primitive. The original JSON value can be restored using the @ref

unflatten() function.

@return an object that maps JSON pointers to primitive values

@note Empty objects and arrays are flattened to `null` and will not be

reconstructed correctly by the @ref unflatten() function.

@complexity Linear in the size the JSON value.

@liveexample{The following code shows how a JSON object is flattened to an

object whose keys consist of JSON pointers.,flatten}

@sa @ref unflatten() for the reverse function

@since version 2.0.0

\*/

basic\_json flatten() const

{

basic\_json result(value\_t::object);

json\_pointer::flatten("", \*this, result);

return result;

}

/\*!

@brief unflatten a previously flattened JSON value

The function restores the arbitrary nesting of a JSON value that has been

flattened before using the @ref flatten() function. The JSON value must

meet certain constraints:

1. The value must be an object.

2. The keys must be JSON pointers (see

[RFC 6901](https://tools.ietf.org/html/rfc6901))

3. The mapped values must be primitive JSON types.

@return the original JSON from a flattened version

@note Empty objects and arrays are flattened by @ref flatten() to `null`

values and can not unflattened to their original type. Apart from

this example, for a JSON value `j`, the following is always true:

`j == j.flatten().unflatten()`.

@complexity Linear in the size the JSON value.

@throw type\_error.314 if value is not an object

@throw type\_error.315 if object values are not primitive

@liveexample{The following code shows how a flattened JSON object is

unflattened into the original nested JSON object.,unflatten}

@sa @ref flatten() for the reverse function

@since version 2.0.0

\*/

basic\_json unflatten() const

{

return json\_pointer::unflatten(\*this);

}

/// @}

//////////////////////////

// JSON Patch functions //

//////////////////////////

/// @name JSON Patch functions

/// @{

/\*!

@brief applies a JSON patch

[JSON Patch](http://jsonpatch.com) defines a JSON document structure for

expressing a sequence of operations to apply to a JSON) document. With

this function, a JSON Patch is applied to the current JSON value by

executing all operations from the patch.

@param[in] json\_patch JSON patch document

@return patched document

@note The application of a patch is atomic: Either all operations succeed

and the patched document is returned or an exception is thrown. In

any case, the original value is not changed: the patch is applied

to a copy of the value.

@throw parse\_error.104 if the JSON patch does not consist of an array of

objects

@throw parse\_error.105 if the JSON patch is malformed (e.g., mandatory

attributes are missing); example: `"operation add must have member path"`

@throw out\_of\_range.401 if an array index is out of range.

@throw out\_of\_range.403 if a JSON pointer inside the patch could not be

resolved successfully in the current JSON value; example: `"key baz not

found"`

@throw out\_of\_range.405 if JSON pointer has no parent ("add", "remove",

"move")

@throw other\_error.501 if "test" operation was unsuccessful

@complexity Linear in the size of the JSON value and the length of the

JSON patch. As usually only a fraction of the JSON value is affected by

the patch, the complexity can usually be neglected.

@liveexample{The following code shows how a JSON patch is applied to a

value.,patch}

@sa @ref diff -- create a JSON patch by comparing two JSON values

@sa [RFC 6902 (JSON Patch)](https://tools.ietf.org/html/rfc6902)

@sa [RFC 6901 (JSON Pointer)](https://tools.ietf.org/html/rfc6901)

@since version 2.0.0

\*/

basic\_json patch(const basic\_json& json\_patch) const

{

// make a working copy to apply the patch to

basic\_json result = \*this;

// the valid JSON Patch operations

enum class patch\_operations {add, remove, replace, move, copy, test, invalid};

const auto get\_op = [](const std::string & op)

{

if (op == "add")

{

return patch\_operations::add;

}

if (op == "remove")

{

return patch\_operations::remove;

}

if (op == "replace")

{

return patch\_operations::replace;

}

if (op == "move")

{

return patch\_operations::move;

}

if (op == "copy")

{

return patch\_operations::copy;

}

if (op == "test")

{

return patch\_operations::test;

}

return patch\_operations::invalid;

};

// wrapper for "add" operation; add value at ptr

const auto operation\_add = [&result](json\_pointer & ptr, basic\_json val)

{

// adding to the root of the target document means replacing it

if (ptr.empty())

{

result = val;

return;

}

// make sure the top element of the pointer exists

json\_pointer top\_pointer = ptr.top();

if (top\_pointer != ptr)

{

result.at(top\_pointer);

}

// get reference to parent of JSON pointer ptr

const auto last\_path = ptr.back();

ptr.pop\_back();

basic\_json& parent = result[ptr];

switch (parent.m\_type)

{

case value\_t::null:

case value\_t::object:

{

// use operator[] to add value

parent[last\_path] = val;

break;

}

case value\_t::array:

{

if (last\_path == "-")

{

// special case: append to back

parent.push\_back(val);

}

else

{

const auto idx = json\_pointer::array\_index(last\_path);

if (JSON\_HEDLEY\_UNLIKELY(static\_cast<size\_type>(idx) > parent.size()))

{

// avoid undefined behavior

JSON\_THROW(out\_of\_range::create(401, "array index " + std::to\_string(idx) + " is out of range"));

}

// default case: insert add offset

parent.insert(parent.begin() + static\_cast<difference\_type>(idx), val);

}

break;

}

// if there exists a parent it cannot be primitive

default: // LCOV\_EXCL\_LINE

assert(false); // LCOV\_EXCL\_LINE

}

};

// wrapper for "remove" operation; remove value at ptr

const auto operation\_remove = [&result](json\_pointer & ptr)

{

// get reference to parent of JSON pointer ptr

const auto last\_path = ptr.back();

ptr.pop\_back();

basic\_json& parent = result.at(ptr);

// remove child

if (parent.is\_object())

{

// perform range check

auto it = parent.find(last\_path);

if (JSON\_HEDLEY\_LIKELY(it != parent.end()))

{

parent.erase(it);

}

else

{

JSON\_THROW(out\_of\_range::create(403, "key '" + last\_path + "' not found"));

}

}

else if (parent.is\_array())

{

// note erase performs range check

parent.erase(static\_cast<size\_type>(json\_pointer::array\_index(last\_path)));

}

};

// type check: top level value must be an array

if (JSON\_HEDLEY\_UNLIKELY(not json\_patch.is\_array()))

{

JSON\_THROW(parse\_error::create(104, 0, "JSON patch must be an array of objects"));

}

// iterate and apply the operations

for (const auto& val : json\_patch)

{

// wrapper to get a value for an operation

const auto get\_value = [&val](const std::string & op,

const std::string & member,

bool string\_type) -> basic\_json &

{

// find value

auto it = val.m\_value.object->find(member);

// context-sensitive error message

const auto error\_msg = (op == "op") ? "operation" : "operation '" + op + "'";

// check if desired value is present

if (JSON\_HEDLEY\_UNLIKELY(it == val.m\_value.object->end()))

{

JSON\_THROW(parse\_error::create(105, 0, error\_msg + " must have member '" + member + "'"));

}

// check if result is of type string

if (JSON\_HEDLEY\_UNLIKELY(string\_type and not it->second.is\_string()))

{

JSON\_THROW(parse\_error::create(105, 0, error\_msg + " must have string member '" + member + "'"));

}

// no error: return value

return it->second;

};

// type check: every element of the array must be an object

if (JSON\_HEDLEY\_UNLIKELY(not val.is\_object()))

{

JSON\_THROW(parse\_error::create(104, 0, "JSON patch must be an array of objects"));

}

// collect mandatory members

const std::string op = get\_value("op", "op", true);

const std::string path = get\_value(op, "path", true);

json\_pointer ptr(path);

switch (get\_op(op))

{

case patch\_operations::add:

{

operation\_add(ptr, get\_value("add", "value", false));

break;

}

case patch\_operations::remove:

{

operation\_remove(ptr);

break;

}

case patch\_operations::replace:

{

// the "path" location must exist - use at()

result.at(ptr) = get\_value("replace", "value", false);

break;

}

case patch\_operations::move:

{

const std::string from\_path = get\_value("move", "from", true);

json\_pointer from\_ptr(from\_path);

// the "from" location must exist - use at()

basic\_json v = result.at(from\_ptr);

// The move operation is functionally identical to a

// "remove" operation on the "from" location, followed

// immediately by an "add" operation at the target

// location with the value that was just removed.

operation\_remove(from\_ptr);

operation\_add(ptr, v);

break;

}

case patch\_operations::copy:

{

const std::string from\_path = get\_value("copy", "from", true);

const json\_pointer from\_ptr(from\_path);

// the "from" location must exist - use at()

basic\_json v = result.at(from\_ptr);

// The copy is functionally identical to an "add"

// operation at the target location using the value

// specified in the "from" member.

operation\_add(ptr, v);

break;

}

case patch\_operations::test:

{

bool success = false;

JSON\_TRY

{

// check if "value" matches the one at "path"

// the "path" location must exist - use at()

success = (result.at(ptr) == get\_value("test", "value", false));

}

JSON\_INTERNAL\_CATCH (out\_of\_range&)

{

// ignore out of range errors: success remains false

}

// throw an exception if test fails

if (JSON\_HEDLEY\_UNLIKELY(not success))

{

JSON\_THROW(other\_error::create(501, "unsuccessful: " + val.dump()));

}

break;

}

default:

{

// op must be "add", "remove", "replace", "move", "copy", or

// "test"

JSON\_THROW(parse\_error::create(105, 0, "operation value '" + op + "' is invalid"));

}

}

}

return result;

}

/\*!

@brief creates a diff as a JSON patch

Creates a [JSON Patch](http://jsonpatch.com) so that value @a source can

be changed into the value @a target by calling @ref patch function.

@invariant For two JSON values @a source and @a target, the following code

yields always `true`:

@code {.cpp}

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

@endcode

@note Currently, only `remove`, `add`, and `replace` operations are

generated.

@param[in] source JSON value to compare from

@param[in] target JSON value to compare against

@param[in] path helper value to create JSON pointers

@return a JSON patch to convert the @a source to @a target

@complexity Linear in the lengths of @a source and @a target.

@liveexample{The following code shows how a JSON patch is created as a

diff for two JSON values.,diff}

@sa @ref patch -- apply a JSON patch

@sa @ref merge\_patch -- apply a JSON Merge Patch

@sa [RFC 6902 (JSON Patch)](https://tools.ietf.org/html/rfc6902)

@since version 2.0.0

\*/

JSON\_HEDLEY\_WARN\_UNUSED\_RESULT

static basic\_json diff(const basic\_json& source, const basic\_json& target,

const std::string& path = "")

{

// the patch

basic\_json result(value\_t::array);

// if the values are the same, return empty patch

if (source == target)

{

return result;

}

if (source.type() != target.type())

{

// different types: replace value

result.push\_back(

{

{"op", "replace"}, {"path", path}, {"value", target}

});

return result;

}

switch (source.type())

{

case value\_t::array:

{

// first pass: traverse common elements

std::size\_t i = 0;

while (i < source.size() and i < target.size())

{

// recursive call to compare array values at index i

auto temp\_diff = diff(source[i], target[i], path + "/" + std::to\_string(i));

result.insert(result.end(), temp\_diff.begin(), temp\_diff.end());

++i;

}

// i now reached the end of at least one array

// in a second pass, traverse the remaining elements

// remove my remaining elements

const auto end\_index = static\_cast<difference\_type>(result.size());

while (i < source.size())

{

// add operations in reverse order to avoid invalid

// indices

result.insert(result.begin() + end\_index, object(

{

{"op", "remove"},

{"path", path + "/" + std::to\_string(i)}

}));

++i;

}

// add other remaining elements

while (i < target.size())

{

result.push\_back(

{

{"op", "add"},

{"path", path + "/" + std::to\_string(i)},

{"value", target[i]}

});

++i;

}

break;

}

case value\_t::object:

{

// first pass: traverse this object's elements

for (auto it = source.cbegin(); it != source.cend(); ++it)

{

// escape the key name to be used in a JSON patch

const auto key = json\_pointer::escape(it.key());

if (target.find(it.key()) != target.end())

{

// recursive call to compare object values at key it

auto temp\_diff = diff(it.value(), target[it.key()], path + "/" + key);

result.insert(result.end(), temp\_diff.begin(), temp\_diff.end());

}

else

{

// found a key that is not in o -> remove it

result.push\_back(object(

{

{"op", "remove"}, {"path", path + "/" + key}

}));

}

}

// second pass: traverse other object's elements

for (auto it = target.cbegin(); it != target.cend(); ++it)

{

if (source.find(it.key()) == source.end())

{

// found a key that is not in this -> add it

const auto key = json\_pointer::escape(it.key());

result.push\_back(

{

{"op", "add"}, {"path", path + "/" + key},

{"value", it.value()}

});

}

}

break;

}

default:

{

// both primitive type: replace value

result.push\_back(

{

{"op", "replace"}, {"path", path}, {"value", target}

});

break;

}

}

return result;

}

/// @}

////////////////////////////////

// JSON Merge Patch functions //

////////////////////////////////

/// @name JSON Merge Patch functions

/// @{

/\*!

@brief applies a JSON Merge Patch

The merge patch format is primarily intended for use with the HTTP PATCH

method as a means of describing a set of modifications to a target

resource's content. This function applies a merge patch to the current

JSON value.

The function implements the following algorithm from Section 2 of

[RFC 7396 (JSON Merge Patch)](https://tools.ietf.org/html/rfc7396):

```

define MergePatch(Target, Patch):

if Patch is an Object:

if Target is not an Object:

Target = {} // Ignore the contents and set it to an empty Object

for each Name/Value pair in Patch:

if Value is null:

if Name exists in Target:

remove the Name/Value pair from Target

else:

Target[Name] = MergePatch(Target[Name], Value)

return Target

else:

return Patch

```

Thereby, `Target` is the current object; that is, the patch is applied to

the current value.

@param[in] apply\_patch the patch to apply

@complexity Linear in the lengths of @a patch.

@liveexample{The following code shows how a JSON Merge Patch is applied to

a JSON document.,merge\_patch}

@sa @ref patch -- apply a JSON patch

@sa [RFC 7396 (JSON Merge Patch)](https://tools.ietf.org/html/rfc7396)

@since version 3.0.0

\*/

void merge\_patch(const basic\_json& apply\_patch)

{

if (apply\_patch.is\_object())

{

if (not is\_object())

{

\*this = object();

}

for (auto it = apply\_patch.begin(); it != apply\_patch.end(); ++it)

{

if (it.value().is\_null())

{

erase(it.key());

}

else

{

operator[](it.key()).merge\_patch(it.value());

}

}

}

else

{

\*this = apply\_patch;

}

}

/// @}

};

/\*!

@brief user-defined to\_string function for JSON values

This function implements a user-defined to\_string for JSON objects.

@param[in] j a JSON object

@return a std::string object

\*/

NLOHMANN\_BASIC\_JSON\_TPL\_DECLARATION

std::string to\_string(const NLOHMANN\_BASIC\_JSON\_TPL& j)

{

return j.dump();

}

} // namespace nlohmann

///////////////////////

// nonmember support //

///////////////////////

// specialization of std::swap, and std::hash

namespace std

{

/// hash value for JSON objects

template<>

struct hash<nlohmann::json>

{

/\*!

@brief return a hash value for a JSON object

@since version 1.0.0

\*/

std::size\_t operator()(const nlohmann::json& j) const

{

// a naive hashing via the string representation

const auto& h = hash<nlohmann::json::string\_t>();

return h(j.dump());

}

};

/// specialization for std::less<value\_t>

/// @note: do not remove the space after '<',

/// see https://github.com/nlohmann/json/pull/679

template<>

struct less<::nlohmann::detail::value\_t>

{

/\*!

@brief compare two value\_t enum values

@since version 3.0.0

\*/

bool operator()(nlohmann::detail::value\_t lhs,

nlohmann::detail::value\_t rhs) const noexcept

{

return nlohmann::detail::operator<(lhs, rhs);

}

};

/\*!

@brief exchanges the values of two JSON objects

@since version 1.0.0

\*/

template<>

inline void swap<nlohmann::json>(nlohmann::json& j1, nlohmann::json& j2) noexcept(

is\_nothrow\_move\_constructible<nlohmann::json>::value and

is\_nothrow\_move\_assignable<nlohmann::json>::value

)

{

j1.swap(j2);

}

} // namespace std

/\*!

@brief user-defined string literal for JSON values

This operator implements a user-defined string literal for JSON objects. It

can be used by adding `"\_json"` to a string literal and returns a JSON object

if no parse error occurred.

@param[in] s a string representation of a JSON object

@param[in] n the length of string @a s

@return a JSON object

@since version 1.0.0

\*/

JSON\_HEDLEY\_NON\_NULL(1)

inline nlohmann::json operator "" \_json(const char\* s, std::size\_t n)

{

return nlohmann::json::parse(s, s + n);

}

/\*!

@brief user-defined string literal for JSON pointer

This operator implements a user-defined string literal for JSON Pointers. It

can be used by adding `"\_json\_pointer"` to a string literal and returns a JSON pointer

object if no parse error occurred.

@param[in] s a string representation of a JSON Pointer

@param[in] n the length of string @a s

@return a JSON pointer object

@since version 2.0.0

\*/

JSON\_HEDLEY\_NON\_NULL(1)

inline nlohmann::json::json\_pointer operator "" \_json\_pointer(const char\* s, std::size\_t n)

{

return nlohmann::json::json\_pointer(std::string(s, n));

}

// #include <nlohmann/detail/macro\_unscope.hpp>

// restore GCC/clang diagnostic settings

#if defined(\_\_clang\_\_) || defined(\_\_GNUC\_\_) || defined(\_\_GNUG\_\_)

#pragma GCC diagnostic pop

#endif

#if defined(\_\_clang\_\_)

#pragma GCC diagnostic pop

#endif

// clean up

#undef JSON\_INTERNAL\_CATCH

#undef JSON\_CATCH

#undef JSON\_THROW

#undef JSON\_TRY

#undef JSON\_HAS\_CPP\_14

#undef JSON\_HAS\_CPP\_17

#undef NLOHMANN\_BASIC\_JSON\_TPL\_DECLARATION

#undef NLOHMANN\_BASIC\_JSON\_TPL

// #include <nlohmann/thirdparty/hedley/hedley\_undef.hpp>

#undef JSON\_HEDLEY\_ALWAYS\_INLINE

#undef JSON\_HEDLEY\_ARM\_VERSION

#undef JSON\_HEDLEY\_ARM\_VERSION\_CHECK

#undef JSON\_HEDLEY\_ARRAY\_PARAM

#undef JSON\_HEDLEY\_ASSUME

#undef JSON\_HEDLEY\_BEGIN\_C\_DECLS

#undef JSON\_HEDLEY\_C\_DECL

#undef JSON\_HEDLEY\_CLANG\_HAS\_ATTRIBUTE

#undef JSON\_HEDLEY\_CLANG\_HAS\_BUILTIN

#undef JSON\_HEDLEY\_CLANG\_HAS\_CPP\_ATTRIBUTE

#undef JSON\_HEDLEY\_CLANG\_HAS\_DECLSPEC\_DECLSPEC\_ATTRIBUTE

#undef JSON\_HEDLEY\_CLANG\_HAS\_EXTENSION

#undef JSON\_HEDLEY\_CLANG\_HAS\_FEATURE

#undef JSON\_HEDLEY\_CLANG\_HAS\_WARNING

#undef JSON\_HEDLEY\_COMPCERT\_VERSION

#undef JSON\_HEDLEY\_COMPCERT\_VERSION\_CHECK

#undef JSON\_HEDLEY\_CONCAT

#undef JSON\_HEDLEY\_CONCAT\_EX

#undef JSON\_HEDLEY\_CONST

#undef JSON\_HEDLEY\_CONST\_CAST

#undef JSON\_HEDLEY\_CONSTEXPR

#undef JSON\_HEDLEY\_CPP\_CAST

#undef JSON\_HEDLEY\_CRAY\_VERSION

#undef JSON\_HEDLEY\_CRAY\_VERSION\_CHECK

#undef JSON\_HEDLEY\_DEPRECATED

#undef JSON\_HEDLEY\_DEPRECATED\_FOR

#undef JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_CAST\_QUAL

#undef JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_CPP98\_COMPAT\_WRAP\_

#undef JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_DEPRECATED

#undef JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_UNKNOWN\_CPP\_ATTRIBUTES

#undef JSON\_HEDLEY\_DIAGNOSTIC\_DISABLE\_UNKNOWN\_PRAGMAS

#undef JSON\_HEDLEY\_DIAGNOSTIC\_POP

#undef JSON\_HEDLEY\_DIAGNOSTIC\_PUSH

#undef JSON\_HEDLEY\_DMC\_VERSION

#undef JSON\_HEDLEY\_DMC\_VERSION\_CHECK

#undef JSON\_HEDLEY\_EMPTY\_BASES

#undef JSON\_HEDLEY\_EMSCRIPTEN\_VERSION

#undef JSON\_HEDLEY\_EMSCRIPTEN\_VERSION\_CHECK

#undef JSON\_HEDLEY\_END\_C\_DECLS

#undef JSON\_HEDLEY\_FALL\_THROUGH

#undef JSON\_HEDLEY\_FLAGS

#undef JSON\_HEDLEY\_FLAGS\_CAST

#undef JSON\_HEDLEY\_GCC\_HAS\_ATTRIBUTE

#undef JSON\_HEDLEY\_GCC\_HAS\_BUILTIN

#undef JSON\_HEDLEY\_GCC\_HAS\_CPP\_ATTRIBUTE

#undef JSON\_HEDLEY\_GCC\_HAS\_DECLSPEC\_ATTRIBUTE

#undef JSON\_HEDLEY\_GCC\_HAS\_EXTENSION

#undef JSON\_HEDLEY\_GCC\_HAS\_FEATURE

#undef JSON\_HEDLEY\_GCC\_HAS\_WARNING

#undef JSON\_HEDLEY\_GCC\_NOT\_CLANG\_VERSION\_CHECK

#undef JSON\_HEDLEY\_GCC\_VERSION

#undef JSON\_HEDLEY\_GCC\_VERSION\_CHECK

#undef JSON\_HEDLEY\_GNUC\_HAS\_ATTRIBUTE

#undef JSON\_HEDLEY\_GNUC\_HAS\_BUILTIN

#undef JSON\_HEDLEY\_GNUC\_HAS\_CPP\_ATTRIBUTE

#undef JSON\_HEDLEY\_GNUC\_HAS\_DECLSPEC\_ATTRIBUTE

#undef JSON\_HEDLEY\_GNUC\_HAS\_EXTENSION

#undef JSON\_HEDLEY\_GNUC\_HAS\_FEATURE

#undef JSON\_HEDLEY\_GNUC\_HAS\_WARNING

#undef JSON\_HEDLEY\_GNUC\_VERSION

#undef JSON\_HEDLEY\_GNUC\_VERSION\_CHECK

#undef JSON\_HEDLEY\_HAS\_ATTRIBUTE

#undef JSON\_HEDLEY\_HAS\_BUILTIN

#undef JSON\_HEDLEY\_HAS\_CPP\_ATTRIBUTE

#undef JSON\_HEDLEY\_HAS\_CPP\_ATTRIBUTE\_NS

#undef JSON\_HEDLEY\_HAS\_DECLSPEC\_ATTRIBUTE

#undef JSON\_HEDLEY\_HAS\_EXTENSION

#undef JSON\_HEDLEY\_HAS\_FEATURE

#undef JSON\_HEDLEY\_HAS\_WARNING

#undef JSON\_HEDLEY\_IAR\_VERSION

#undef JSON\_HEDLEY\_IAR\_VERSION\_CHECK

#undef JSON\_HEDLEY\_IBM\_VERSION

#undef JSON\_HEDLEY\_IBM\_VERSION\_CHECK

#undef JSON\_HEDLEY\_IMPORT

#undef JSON\_HEDLEY\_INLINE

#undef JSON\_HEDLEY\_INTEL\_VERSION

#undef JSON\_HEDLEY\_INTEL\_VERSION\_CHECK

#undef JSON\_HEDLEY\_IS\_CONSTANT

#undef JSON\_HEDLEY\_IS\_CONSTEXPR\_

#undef JSON\_HEDLEY\_LIKELY

#undef JSON\_HEDLEY\_MALLOC

#undef JSON\_HEDLEY\_MESSAGE

#undef JSON\_HEDLEY\_MSVC\_VERSION

#undef JSON\_HEDLEY\_MSVC\_VERSION\_CHECK

#undef JSON\_HEDLEY\_NEVER\_INLINE

#undef JSON\_HEDLEY\_NO\_ESCAPE

#undef JSON\_HEDLEY\_NON\_NULL

#undef JSON\_HEDLEY\_NO\_RETURN

#undef JSON\_HEDLEY\_NO\_THROW

#undef JSON\_HEDLEY\_NULL

#undef JSON\_HEDLEY\_PELLES\_VERSION

#undef JSON\_HEDLEY\_PELLES\_VERSION\_CHECK

#undef JSON\_HEDLEY\_PGI\_VERSION

#undef JSON\_HEDLEY\_PGI\_VERSION\_CHECK

#undef JSON\_HEDLEY\_PREDICT

#undef JSON\_HEDLEY\_PRINTF\_FORMAT

#undef JSON\_HEDLEY\_PRIVATE

#undef JSON\_HEDLEY\_PUBLIC

#undef JSON\_HEDLEY\_PURE

#undef JSON\_HEDLEY\_REINTERPRET\_CAST

#undef JSON\_HEDLEY\_REQUIRE

#undef JSON\_HEDLEY\_REQUIRE\_CONSTEXPR

#undef JSON\_HEDLEY\_REQUIRE\_MSG

#undef JSON\_HEDLEY\_RESTRICT

#undef JSON\_HEDLEY\_RETURNS\_NON\_NULL

#undef JSON\_HEDLEY\_SENTINEL

#undef JSON\_HEDLEY\_STATIC\_ASSERT

#undef JSON\_HEDLEY\_STATIC\_CAST

#undef JSON\_HEDLEY\_STRINGIFY

#undef JSON\_HEDLEY\_STRINGIFY\_EX

#undef JSON\_HEDLEY\_SUNPRO\_VERSION

#undef JSON\_HEDLEY\_SUNPRO\_VERSION\_CHECK

#undef JSON\_HEDLEY\_TINYC\_VERSION

#undef JSON\_HEDLEY\_TINYC\_VERSION\_CHECK

#undef JSON\_HEDLEY\_TI\_VERSION

#undef JSON\_HEDLEY\_TI\_VERSION\_CHECK

#undef JSON\_HEDLEY\_UNAVAILABLE

#undef JSON\_HEDLEY\_UNLIKELY

#undef JSON\_HEDLEY\_UNPREDICTABLE

#undef JSON\_HEDLEY\_UNREACHABLE

#undef JSON\_HEDLEY\_UNREACHABLE\_RETURN

#undef JSON\_HEDLEY\_VERSION

#undef JSON\_HEDLEY\_VERSION\_DECODE\_MAJOR

#undef JSON\_HEDLEY\_VERSION\_DECODE\_MINOR

#undef JSON\_HEDLEY\_VERSION\_DECODE\_REVISION

#undef JSON\_HEDLEY\_VERSION\_ENCODE

#undef JSON\_HEDLEY\_WARNING

#undef JSON\_HEDLEY\_WARN\_UNUSED\_RESULT

#endif // INCLUDE\_NLOHMANN\_JSON\_HPP\_

#include "Appearance.h"

namespace ATable {

Appearance::Appearance() {

this->left\_top\_corner = ' ';

this->right\_top\_corner = ' ';

this->left\_bottom\_corner = ' ';

this->right\_bottom\_corner = ' ';

this->vertical\_line = ' ';

this->horizontal\_line = ' ';

this->left\_separator = ' ';

this->right\_separator = ' ';

this->top\_separator = ' ';

this->bottom\_separator = ' ';

this->center\_separator = ' ';

}

Appearance::~Appearance() {

}

void Appearance::print(ostream& stream) {

stream << "left\_top\_corner: " << left\_top\_corner << endl;

stream << "right\_top\_corner: " << right\_top\_corner << endl;

stream << "left\_bottom\_corner: " << left\_bottom\_corner << endl;

stream << "right\_bottom\_corner: " << right\_bottom\_corner << endl;

stream << "vertical\_line: " << vertical\_line << endl;

stream << "horizontal\_line: " << horizontal\_line << endl;

stream << "left\_separator: " << left\_separator << endl;

stream << "right\_separator: " << right\_separator << endl;

stream << "top\_separator: " << top\_separator << endl;

stream << "bottom\_separator: " << bottom\_separator << endl;

stream << "center\_separator: " << center\_separator << endl;

}

ostream& operator<<(ostream& stream, Appearance obj)

{

obj.print(stream);

return stream;

}

ostream& operator<<(ostream& stream, Appearance\* obj) {

return operator<<(stream, \*obj);

}

}

#ifndef COURSEWORK\_APPEAR\_H

#define COURSEWORK\_APPEAR\_H

#include "Object.h"

#include <iostream>

namespace ATable {

class Appearance : public Object {

public:

char left\_top\_corner;

char right\_top\_corner;

char left\_bottom\_corner;

char right\_bottom\_corner;

char vertical\_line;

char horizontal\_line;

char left\_separator;

char right\_separator;

char top\_separator;

char bottom\_separator;

char center\_separator;

public:

Appearance();

~Appearance();

void print(ostream &stream);

friend ostream &operator<<(ostream &stream, Appearance obj);

friend ostream &operator<<(ostream &stream, Appearance \*obj);

};

}

#endif //COURSEWORK\_APPEAR\_H

#ifndef COURSEWORK\_ATABLE\_H

#define COURSEWORK\_ATABLE\_H

#include "Table.h"

#include "SimpleColumn.h"

#include "ConstColumn.h"

#include "DefaultCells.h"

#include "DefaultAppearance.h"

#endif //COURSEWORK\_ATABLE\_H

#include "Cell.h"

namespace ATable {

Cell::Cell() {

this->width = 0;

}

Cell::~Cell() {

}

void Cell::print(ostream& stream) {

stream << this->boundary(this->output());

}

void Cell::setWidth(unsigned int width) {

this->width = width;

}

unsigned int Cell::getWidth() {

return this->width;

}

string Cell::boundary(string value) {

if (value.length() < this->width) {

return value + string(this->width - value.length(), ' ');

}

if (value.length() > this->width) {

string temp = value;

temp.resize(this->width);

return temp;

}

return value;

}

}

#ifndef COURSEWORK\_CELL\_H

#define COURSEWORK\_CELL\_H

#include "Object.h"

#include <string>

#include <iostream>

namespace ATable {

class Cell : public Object {

private:

unsigned int width;

public:

Cell();

~Cell();

virtual string output() = 0;

void print(ostream &stream);

void setWidth(unsigned int width);

unsigned int getWidth();

private:

string boundary(string value);

};

}

#endif //COURSEWORK\_CELL\_H

#include "Column.h"

namespace ATable {

Column::Column(unsigned int width) {

this->width = width;

}

Column::~Column() {

}

unsigned int Column::getWidth() {

return this->width;

}

void Column::setWidth(unsigned int width) {

this->width = width;

}

}

#ifndef COURSEWORK\_COLUMN\_H

#define COURSEWORK\_COLUMN\_H

#include "Object.h"

#include "Cell.h"

#include "TableExceptions.h"

namespace ATable {

class Column : public Object {

public:

unsigned int width;

public:

Column(unsigned int width);

~Column();

unsigned int getWidth();

void setWidth(unsigned int width);

virtual Cell \*getCell(int id) = 0;

};

}

#endif //COURSEWORK\_COLUMN\_H

#include "ConstColumn.h"

namespace ATable {

ConstColumn::ConstColumn(string value) : Column(value.length()){

this->cell = new StringCell(value);

this->cell->setWidth(this->getWidth());

}

ConstColumn::~ConstColumn() {

delete cell;

}

Cell\* ConstColumn::getCell(int id) {

return this->cell;

}

}

#include "Column.h"

#include "DefaultCells.h"

namespace ATable {

class ConstColumn : public Column {

protected:

Cell\* cell;

public:

ConstColumn(string value);

~ConstColumn();

Cell\* getCell(int id);

};

}

#pragma once

#include "Appearance.h"

namespace ATable {

class DefaultAppearance : public Appearance {

public:

DefaultAppearance() {

this->left\_top\_corner = 218;

this->right\_top\_corner = 191;

this->left\_bottom\_corner = 192;

this->right\_bottom\_corner = 217;

this->top\_separator = 194;

this->bottom\_separator = 193;

this->left\_separator = 195;

this->right\_separator = 180;

this->center\_separator = 197;

this->vertical\_line = 179;

this->horizontal\_line = 196;

}

};

}

#pragma once

#include <cmath>

#include "Cell.h"

namespace ATable {

class StringCell : public Cell {

public:

string value;

public:

StringCell(string value) {

this->value = value;

}

string output() {

return this->value;

}

};

class IntegerCell : public Cell {

public:

int value;

public:

IntegerCell(int value) {

this->value = value;

}

string output() {

return to\_string(this->value);

}

};

class LongCell : public Cell {

public:

long value;

public:

LongCell(long value) {

this->value = value;

}

string output() {

return to\_string(this->value);

}

};

class FloatCell : public Cell {

public:

float value;

int percision;

public:

FloatCell(float value, int percision) {

this->value = value;

this->percision = percision;

}

FloatCell(float value) : FloatCell(value, 3) {}

string output() {

string integer = to\_string((int) value);

string fractional = to\_string((int) (abs(value - (int) value) \* pow(10, percision)));

string result;

result += integer + "." + fractional;

return result;

}

};

class DoubleCell : public Cell {

public:

float value;

int percision;

public:

DoubleCell(double value, int percision) {

this->value = value;

this->percision = percision;

}

DoubleCell(double value) : DoubleCell(value, 3) {}

string output() {

string integer = to\_string((int) value);

string fractional = to\_string((int) (abs(value - (int) value) \* pow(10, percision)));

string result;

result += integer + "." + fractional;

return result;

}

};

}

#include "Object.h"

Object::Object() = default;

Object::~Object() = default;

#ifndef COURSEWORK\_OBJECT\_H

#define COURSEWORK\_OBJECT\_H

using namespace std;

class Object {

public:

Object();

virtual ~Object();

};

#endif //COURSEWORK\_OBJECT\_H

#include "SimpleColumn.h"

namespace ATable {

SimpleColumn::SimpleColumn(string name, unsigned int widht) : Column(width) {

this->width = widht;

this->name = name;

this->cells = new vector<Cell\*>();

}

SimpleColumn::~SimpleColumn() {

for (unsigned int i = 0; i < this->cells->size(); i++) {

delete this->getCell(i);

}

delete this->cells;

}

string SimpleColumn::getName() {

return this->name;

}

void SimpleColumn::setName(string name) {

this->name = name;

}

void SimpleColumn::addCell(Cell\* cell) {

cell->setWidth(this->width);

this->cells->push\_back(cell);

}

Cell\* SimpleColumn::getCell(int id) {

if (id >= this->cells->size()) {

return NULL;

}

Cell\* cell = (\*(this->cells))[id];

return cell;

}

unsigned int SimpleColumn::getHeight() {

return this->cells->size();

}

}

#ifndef COURSEWORK\_SC\_H

#define COURSEWORK\_SC\_H

#include "Cell.h"

#include "Column.h"

#include <vector>

namespace ATable {

class SimpleColumn : public Column {

public:

string name;

protected:

vector<Cell \*> \*cells;

public:

SimpleColumn(string name, unsigned int width);

~SimpleColumn();

string getName();

void setName(string name);

void addCell(Cell \*cell);

Cell \*getCell(int id);

unsigned int getHeight();

};

}

#endif //COURSEWORK\_SC\_H

#include "Table.h"

namespace ATable {

Table::Table(Appearance preset, string name) {

this->preset = preset;

this->name = name;

this->columns = new vector<Column\*>();

}

Table::~Table() {

for (unsigned int col = 0; col < this->columns->size(); col++) {

Column\* temp = this->getColumn(col);

delete temp;

}

delete this->columns;

}

void Table::addColumn(Column\* column) {

this->columns->push\_back(column);

}

void Table::addCell(int column, Cell\* cell) {

SimpleColumn\* scolumn = dynamic\_cast<SimpleColumn\*>(this->getColumn(column));

if (scolumn) {

scolumn->addCell(cell);

}

}

void Table::print(ostream & stream) {

try {

//Top line

this->drawLine(stream, this->preset.left\_top\_corner, this->preset.horizontal\_line, this->preset.top\_separator, this->preset.right\_top\_corner);

//Output headers

this->drawColumnsHeaders(stream, this->preset.vertical\_line);

this->drawLine(stream, this->preset.left\_separator, this->preset.horizontal\_line, this->preset.center\_separator, this->preset.right\_separator);

//Rows

unsigned int maxRowHeight = this->getMaxRowHeight();

for (int row = 0; row < maxRowHeight; row++) {

this->drawRow(stream, row, this->preset.vertical\_line);

if (row < maxRowHeight - 1) {

this->drawLine(stream, this->preset.left\_separator, this->preset.horizontal\_line, this->preset.center\_separator, this->preset.right\_separator);

}

}

//Bottom line

this->drawLine(stream, this->preset.left\_bottom\_corner, this->preset.horizontal\_line, this->preset.bottom\_separator, this->preset.right\_bottom\_corner);

}

catch (exception e) {

stream << "Unable to print table [" << this->name << "]" << endl;

}

}

Column\* Table::getColumn(int id) {

if (!(id < this->columns->size())) {

throw new WrongIDException();

}

return (\*(this->columns))[id];

}

void Table::drawLine(ostream& stream, char left\_corner, char horizontal\_line, char center\_separator, char right\_corner) {

stream << left\_corner;

for (unsigned int col = 0; col < this->columns->size(); col++) {

stream << string(this->getColumn(col)->getWidth(), horizontal\_line);

if (col < this->columns->size() - 1) {

stream << center\_separator;

}

}

stream << right\_corner << endl;

}

void Table::drawRow(ostream& stream, unsigned int row, char vertical\_line) {

stream << vertical\_line;

for (unsigned int col = 0; col < this->columns->size(); col++) {

Cell\* cell = this->getColumn(col)->getCell(row);

if (cell) {

cell->print(stream);

}

else {

stream << string(this->getColumn(col)->getWidth(), ' ');

}

stream << vertical\_line;

}

stream << endl;

}

void Table::drawColumnsHeaders(ostream& stream, char vertical\_line) {

stream << vertical\_line;

for (unsigned int col = 0; col < this->columns->size(); col++) {

Column\* column = this->getColumn(col);

SimpleColumn\* scolumn = dynamic\_cast<SimpleColumn\*>(column);

if (scolumn) {

stream << boundary(scolumn->getName(), scolumn->getWidth());

}

else {

stream << string(column->getWidth(),' ');

}

stream << vertical\_line;

}

stream << endl;

}

unsigned int Table::getMaxRowHeight() {

unsigned int maxHeight = 0;

for (int i = 0; i < this->columns->size(); i++) {

SimpleColumn\* scolumn = dynamic\_cast<SimpleColumn\*>(this->getColumn(i));

if (scolumn) {

unsigned int temp = scolumn->getHeight();

if (maxHeight < temp) {

maxHeight = temp;

}

}

}

return maxHeight;

}

string Table::boundary(string value, int width) {

if (value.length() < width) {

return value + string(width - value.length(), ' ');

}

if (value.length() > width) {

string temp = value;

temp.resize(width);

return temp;

}

return value;

}

}

#ifndef COURSEWORK\_TABLE\_H

#define COURSEWORK\_TABLE\_H

#include "Object.h"

#include "SimpleColumn.h"

#include "TableExceptions.h"

#include "Appearance.h"

#include <vector>

#include <iostream>

namespace ATable {

class Table : public Object {

public:

vector<Column \*> \*columns;

Appearance preset;

string name;

public:

Table(Appearance preset, string name);

~Table();

void addColumn(Column \*column);

void addCell(int column, Cell \*cell);

virtual void print(ostream &stream);

private:

Column \*getColumn(int id);

void

drawLine(ostream &stream, char left\_corner, char horizontal\_line, char center\_separator, char right\_corner);

void drawRow(ostream &stream, unsigned int row, char vertical\_line);

void drawColumnsHeaders(ostream &stream, char vertical\_line);

unsigned int getMaxRowHeight();

public:

static string boundary(string value, int width);

};

}

#endif //COURSEWORK\_TABLE\_H

#ifndef COURSEWORK\_TE\_H

#define COURSEWORK\_TE\_H

#include "Object.h"

#include <string>

namespace ATable {

class Exception : public Object {

protected:

string message;

public:

Exception(string message) {

this->message = message;

}

~Exception() {}

string getMessage() {

return this->message;

}

};

class WrongIDException : public Exception {

public:

WrongIDException() : Exception("Exception: Wrong id exception") {}

};

class WrongColumnTypeException : public Exception {

public:

WrongColumnTypeException() : Exception("Exception: Wrong column type exception") {}

};

}

#endif //COURSEWORK\_TE\_H