Static visitor

Jens Emil Fink Højriis

March 17, 2024

Listing 1: ./include/json_input.hpp

```
#ifndef STATIC_VISITOR_JSON_INPUT_HPP
#define STATIC_VISITOR_JSON_INPUT_HPP

#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include "meta.hpp"

/** TODO: implement json_istream adapter with json input operations
 * The goal is to exercise meta-programming and not have complete JSON (Unicode support is beyond the scope).
 * Parsing should follow the type structure rather than the content of the input stream.
 * Visitor parsing may depend on the order of fields, which is OK for this exercise.
 */

struct json_istream
{
    std::istream& is;
};

/** Visitor pattern support for reading JSON */
class json_reader_t
{
    json_istream& _j;
    bool _first = true;
public:
    json_reader_t(json_istream& j)
        : _j(j) { }

    template <typename Data>
    void visit(const std::string& name, Data& value)
    {
        if (!_first) {
            if (_j.is.get() != ',') {
                throw std::exception("Malformed json, expected ,");
            }
        }
        _first = false;
        std::string s;
        _j.is >> std::quoted(s);
        if (s != name) {
            throw std::exception("Unexpected field ordering");
        }
        if (_j.is.get() != ':') {
            throw std::exception("Malformed json, expected :");
        }
        _j >> value;
    }
```

```
};

template<typename T>
struct SomeReader {
    static json_istream& read(json_istream& j, T& out);
};

template <BoolC B>
struct SomeReader<B>
{
    static json_istream& read(json_istream& j, B& out)
    {
        char buf[5];
        j.is.get(buf, 5);
        if (strcmp(buf, "true") == 0) {
            out = true;
            return j;
        } else if (strcmp(buf, "fals") == 0) {
            if (j.is.get() == 'e') {
                out = false;
                return j;
            }
        }
        throw std::exception("Expected false or true");
    }
};

template<NumberC N>
struct SomeReader<N>
{
    static json_istream& read(json_istream& j, N& out) {
        j.is >> out;
        return j;
    }
};

template <StringC S>
struct SomeReader<S>
{
    static json_istream& read(json_istream& j, S& out) {
        j.is >> std::quoted(out);
        return j;
    }
};

template <WriteableContainer W>
struct SomeReader<W>
{
    static json_istream& read(json_istream& j, W& out) {
        if (j.is.get() != '[') {
            throw std::exception("Malformed json, expected [");
        }
        while (j.is.peek() != ']') {
            j >> out.emplace_back();
            if (j.is.peek() == ',') {
                j.is.get();
            } else if ( j.is.peek() != ']') {
                throw std::exception("Malformed json, expected ] or ,");
            }
        }
        j.is.get();
```

```
110         return j;
111     }
112 };
113 
114 template<DataC<json_reader_t> D>
115 struct SomeReader<D>
116 {
117     static json_istream& read(json_istream& j, D& out)
118     {
119         if (j.is.get() != '{') {
120             throw std::exception("Malformed json, expected {");
121         }
122         json_reader_t reader(j);
123         out.accept(reader);
124         if (j.is.get() != '}') {
125             throw std::exception("Malformed json, expected }");
126         }
127         return j;
128     }
129 };
130 
131 template <typename T>
132 json_istream& operator>>(json_istream& j, T& out)
133 {
134     return SomeReader<T>::read(j, out);
135 }
136 
137 /** Helper for rvalue reference */
138 template <typename T>
139 json_istream& operator>>(json_istream&& j, T& value)
140 {
141     return j >> value;
142 }
143 
144 
145 #endif  // STATIC_VISITOR_JSON_INPUT_HPP
```

Listing 2: ./include/json_output.hpp

```
1  #ifndef STATIC_VISITOR_JSON_OUTPUT_HPP
2  #define STATIC_VISITOR_JSON_OUTPUT_HPP
3  
4  #include "meta.hpp"
5  #include <iostream>
6  #include <iomanip>
7  
8  /** TODO: implement json_ostream adapter with json output operations
9   * The goal is to exercise meta-programming and not have complete JSON (Unicode support is
   ↪beyond the scope).
10  */
11 struct json_ostream
12 {
13     std::ostream& os;
14 };
15 
16 template<typename T>
17 struct SomeWriter {
18     static void write(json_ostream& j, const T& v);
19 };
20 
21 template<BoolC B>
22 struct SomeWriter<B> {
```

```
    static void write(json_ostream& j, const B& v) {
        j.os << (v ? "true" : "false");
    }
};

template <NumberC N>
struct SomeWriter<N> {
    static void write(json_ostream& j, const N& v) {
        j.os << v;
    }
};

template <StringC S>
struct SomeWriter<S> {
    static void write(json_ostream& j, const S& v) {
        j.os << std::quoted(v);
    }
};

template <ContainerC C>
struct SomeWriter<C>
{
    static void write(json_ostream& j, const C& v) {
        auto it = std::cbegin(v);
        j.os << "[";
        if (it != std::cend(v)) {
            j.os << *it;
        }
        for (it++; it != std::cend(v); it++) {
            j.os << "," << *it;
        }
        j.os << "]";
    }
};

/** Visitor pattern support for writing JSON */
class json_writer_t
{
    json_ostream& _jos;
    bool _first = true;
public:
    json_writer_t(json_ostream& jos)
        : _jos(jos) {}

    template <typename Data>
    void visit(const std::string& name, const Data& value)
    {
        if (!_first) {
            _jos.os << ",";
        }
        _first = false;
        _jos.os << std::quoted(name) << ":";
        _jos << value;
    }
};

template <TupleC T>
struct SomeWriter<T>
{
    static void write(json_ostream& j, const T& v) {
        json_writer_t visitor(j);
```

```
        for (size_t i = 0; i < std::tuple_size_v<T>; i++) {
            visitor.visit(std::to_string(i + 1), v);
        }
    }
};

template <DataC<json_writer_t> D>
struct SomeWriter<D>
{
    static json_ostream& write(json_ostream& j, const D& data)
    {
        j.os << "{";
        json_writer_t writer(j);
        data.accept(writer);
        j.os << "}";
        return j;
    }
};

template <typename T>
json_ostream& operator<<(json_ostream& j, const T& v) {
    SomeWriter<T>::write(j, v);
    return j;
}

template <typename T>
json_ostream& operator<<(json_ostream&& j, T& value)
{
    return j << value;
}

#endif  // STATIC_VISITOR_JSON_OUTPUT_HPP
```

Listing 3: ./include/meta.hpp

```
#ifndef STATIC_VISITOR_META_HPP
#define STATIC_VISITOR_META_HPP

/**
 * TODO: implement meta-predicates
 * (template classes/variables taking type and returning bool)
 * is_bool_v
 * is_number_v
 * is_character_v
 * is_string_v
 * is_container_v
 * accepts_v
 *
 * Tips:
 * - read documentation about std::is_same, std::is_integral, std::is_floating_point, ↙
↪std::is_arithmetic.
 * - make your meta-predicates robust against const/volatile/reference combinations (see ↙
↪meta_test.cpp):
 *   see std::remove_reference, std::remove_const, std::remove_cvref, std::decay
 */

#include <type_traits>
#include <string>
#include <tuple>
#include <utility>

template <typename T>
```

```cpp
constexpr auto is_bool_v = std::is_same_v<std::remove_cvref_t<T>, bool>;

template <typename T>
constexpr auto is_number_v =
    !std::is_same_v<std::remove_cvref_t<T>, bool> &&
    (std::is_integral_v<std::remove_cvref_t<T>> ||
↪std::is_floating_point_v<std::remove_cvref_t<T>>);

template <typename T>
constexpr auto is_character_v = std::is_same_v<std::remove_cvref_t<T>, char>;

template <typename T>
constexpr auto is_string_v = std::is_same_v<std::remove_cvref_t<T>, std::string> ||
                             is_character_v<std::remove_extent_t<T>> ||
↪is_character_v<std::remove_pointer_t<T>>;

template <typename T>
constexpr auto is_tuple_v = requires {
    {
        std::tuple_size<T>()
    } -> std::convertible_to<std::size_t>;
};

template <typename T>
constexpr auto is_container_v = requires(T&& c) {
    {
        std::cbegin(c)
    };
    {
        std::cend(c)
    };
} && !is_string_v<T>;

template <typename T>
constexpr auto is_writeable_container_v = requires(T&& c) {
    {
        c.emplace_back()
    };
} && !is_string_v<T>;

template <typename Data, typename Visitor>
constexpr auto accepts_v = requires(Data d, Visitor v) {
    {
        d.accept(v)
    };
};

template <typename T>
concept BoolC = is_bool_v<T>;

template <typename T>
concept NumberC = is_number_v<T>;

template <typename T>
concept StringC = is_string_v<T>;

template <typename T>
concept ContainerC = is_container_v<T>;

template <typename T>
concept WriteableContainer = is_writeable_container_v<T>;
```

```
85
86  template <typename T, typename W>
87  concept DataC = accepts_v<T, W>;
88
89  template <typename T>
90  concept TupleC = is_tuple_v<T>;
91
92  template <typename T>
93  class TD;
94
95  #endif  // STATIC_VISITOR_META_HPP
```

Listing 4: ./src/data.hpp

```
1   #ifndef STATIC_VISITOR_DATA_HPP
2   #define STATIC_VISITOR_DATA_HPP
3
4   #include <string>
5   #include <vector>
6
7   /** custom class to test JSON input/output */
8   struct aggregate_t
9   { /** public access is just for easy structured initialization in tests */
10      bool b;
11      int x;
12      double y;
13      std::string z;
14      std::vector<int> w;
15      /** visitor support with read-only access, e.g. for writing-out */
16      template <typename Visitor>
17      void accept(Visitor&& v) const
18      {
19          v.visit("b", b);
20          v.visit("x", x);
21          v.visit("y", y);
22          v.visit("z", z);
23          v.visit("w", w);
24      }
25      /** visitor support with full access, e.g. for reading-in */
26      template <typename Visitor>
27      void accept(Visitor&& v)
28      {
29          v.visit("b", b);
30          v.visit("x", x);
31          v.visit("y", y);
32          v.visit("z", z);
33          v.visit("w", w);
34      }
35      /** equality operator to support testing */
36      friend bool operator==(const aggregate_t& a1, const aggregate_t& a2)
37      {
38          return (a1.b == a2.b) && (a1.x == a2.x) && (a1.y == a2.y) && (a1.z == a2.z) && (a1.w ==
    ↪a2.w);
39      }
40  };
41
42  /** custom class to test JSON input/output with nesting */
43  struct nested_t
44  {
45      std::string text;
46      aggregate_t agg;
47      /** visitor support with read-only access, e.g. for writing-out */
```

```
48      template <typename Visitor>
49      void accept(Visitor&& v) const
50      {
51          v.visit("text", text);
52          v.visit("agg", agg);
53      }
54      /** visitor support with full access, e.g. for reading-in */
55      template <typename Visitor>
56      void accept(Visitor&& v)
57      {
58          v.visit("text", text);
59          v.visit("agg", agg);
60      }
61      /** equality operator to support testing */
62      friend bool operator==(const nested_t& n1, const nested_t& n2)
63      {
64          return (n1.text == n2.text) && (n1.agg == n2.agg);
65      }
66  };
67
68  #endif  // STATIC_VISITOR_DATA_HPP
```

Listing 5: ./src/json_input_test.cpp

```
1   #include "json_input.hpp"
2   #include "meta.hpp"
3   #include "data.hpp"
4
5   #include <doctest/doctest.h>
6
7   #include <sstream>
8   #include <vector>
9
10  using namespace std::string_literals;
11
12  TEST_CASE("JSON input")
13  {
14      SUBCASE("boolean: true")
15      {
16          auto is = std::istringstream{"true"};
17          auto v = false;
18          json_istream{is} >> v;
19          CHECK(is);
20          CHECK(v == true);
21      }
22      SUBCASE("boolean: false")
23      {
24          auto is = std::istringstream{"false"};
25          auto v = true;
26          json_istream{is} >> v;
27          CHECK(is);
28          CHECK(v == false);
29      }
30      SUBCASE("integer")
31      {
32          auto is = std::istringstream{"7"};
33          auto v = 0;
34          json_istream{is} >> v;
35          CHECK(is);
36          CHECK(v == 7);
37      }
38      SUBCASE("double")
```

```
    {
        auto is = std::istringstream{"3.14"};
        auto v = 0.0;
        json_istream{is} >> v;
        CHECK(is);
        CHECK(v == 3.14);
    }
    SUBCASE("cpp-string")
    {
        auto is = std::istringstream{"\"hello\""};
        auto v = std::string{};
        json_istream{is} >> v;
        CHECK(is);
        CHECK(v == "hello");
    }
    SUBCASE("container")
    {
        auto is = std::istringstream{"[3,7,11]"};
        auto v = std::vector<int>{};
        json_istream{is} >> v;
        CHECK(is);
        CHECK(v == std::vector{3, 7, 11});
    }
    static_assert(accepts_v<aggregate_t&, json_reader_t>, "aggregate should accept reader");
    static_assert(accepts_v<nested_t&, json_reader_t>, "nested should accept reader");
    // TODO: uncomment the following extra tests for meta library and fix accepts_v implementation
    static_assert(!accepts_v<int, double>, "int should not accept double");
    SUBCASE("aggregate")
    {
        auto is = std::istringstream{R"({"b":true,"x":3,"y":3.14,"z":"hello","w":[7,11]})"};
        auto v = aggregate_t{};
        json_istream{is} >> v;
        CHECK(is);
        CHECK(v == aggregate_t{true, 3, 3.14, "hello", {7, 11}});
    }
    SUBCASE("nested")
    {
        auto is =
            std::istringstream{R"({"text":"complicated","agg":{"b":true,"x":3,"y":3.14,"z":"hello","w":[7,11]
        auto v = nested_t{};
        json_istream{is} >> v;
        CHECK(is);
        CHECK(v == nested_t{"complicated", {true, 3, 3.14, "hello", {7, 11}}});
    }
}
```

Listing 6: ./src/json_output_test.cpp

```
#include "json_output.hpp"
#include "meta.hpp"
#include "data.hpp"

#include <doctest/doctest.h>

#include <sstream>
#include <vector>

using namespace std::string_literals;

/** Output operator just for friendly output in tests: */
std::ostream& operator<<(std::ostream& os, const aggregate_t& agg)
{
```

```
15      json_ostream { os } << agg;
16      return os;
17  }
18
19  /** Output operator just for friendly output in tests: */
20  std::ostream& operator<<(std::ostream& os, const nested_t& nested)
21  {
22      json_ostream { os } << nested;
23      return os;
24  }
25
26  TEST_CASE("JSON output")
27  {
28      auto os = std::ostringstream{};
29      auto jos = json_ostream{os};
30      SUBCASE("boolean: lvalue true")
31      {
32          auto v = true;
33          jos << v;
34          CHECK(os.str() == "true");
35      }
36      SUBCASE("boolean: lvalue false")
37      {
38          auto v = false;
39          jos << v;
40          CHECK(os.str() == "false");
41      }
42      SUBCASE("boolean: rvalue")
43      {
44          jos << true;
45          CHECK(os.str() == "true");
46      }
47      SUBCASE("integer")
48      {
49          jos << 7;
50          CHECK(os.str() == "7");
51      }
52      SUBCASE("double")
53      {
54          jos << 3.14;
55          CHECK(os.str() == "3.14");
56      }
57      SUBCASE("c-string")
58      {
59          jos << "hello";
60          CHECK(os.str() == "\"hello\"");
61      }
62      SUBCASE("cpp-string")
63      {
64          jos << "hello"s;
65          CHECK(os.str() == "\"hello\"");
66      }
67      SUBCASE("container")
68      {
69          auto v = std::vector{3, 7, 11};
70          jos << v;
71          CHECK(os.str() == "[3,7,11]");
72      }
73      static_assert(accepts_v<const aggregate_t&, json_writer_t>, "const aggregate should accept
    ↪writer");
74      static_assert(accepts_v<const nested_t&, json_writer_t>, "const nested should accept writer");
```

```
    // TODO: uncomment the following extra tests for meta library and fix accepts_v implementation
    static_assert(!accepts_v<json_writer_t, aggregate_t>, "writer should not accept aggregate");
    static_assert(!accepts_v<int, double>, "int should not accept double");
    SUBCASE("aggregate")
    {
        auto v = aggregate_t{true, 3, 3.14, "hello", {7, 11}};
        jos << v;
        CHECK(os.str() == R"({"b":true,"x":3,"y":3.14,"z":"hello","w":[7,11]})");
    }
    SUBCASE("nested")
    {
        const auto v = nested_t{"complicated", {true, 3, 3.14, "hello", {7, 11}}};
        jos << v;
        CHECK(os.str() ==
↪R"({"text":"complicated","agg":{"b":true,"x":3,"y":3.14,"z":"hello","w":[7,11]}})");
    }
}
```

Listing 7: ./src/meta_test.cpp

```
#include "meta.hpp"

#include <doctest/doctest.h>

#include <string>
#include <vector>
#include <set>
#include <map>
#include <cstdint>  // uint8_t

/** TODO: fix the meta library to satisfy the assertions below. */

/** is_bool_v tests: */
static_assert(is_bool_v<bool>, "bool is a bool");
static_assert(is_bool_v<bool&>, "reference to bool is a bool");
static_assert(is_bool_v<bool&&>, "rvalue reference to bool is a bool");
static_assert(is_bool_v<const bool>, "const bool is a bool");
static_assert(is_bool_v<const bool&>, "const reference to bool is a bool");
static_assert(is_bool_v<volatile bool>, "volatile bool is a bool");

// TODO: Uncomment
static_assert(!is_bool_v<bool*>, "pointer to bool is not a bool");
static_assert(!is_bool_v<bool[2]>, "bool array is not a bool");
static_assert(!is_bool_v<char>, "char is not a bool");
static_assert(!is_bool_v<int>, "int is not a bool");
static_assert(!is_bool_v<float>, "float is not a bool");
static_assert(!is_bool_v<double>, "double is not a bool");
static_assert(!is_bool_v<std::string>, "string is not a bool");
static_assert(!is_bool_v<std::vector<bool>>, "vector of bool is not a bool");


/** is_number_v tests: */
static_assert(is_number_v<uint8_t>, "uint8_t is a number");
static_assert(is_number_v<int8_t>, "int8_t is a number");
static_assert(is_number_v<int>, "int is a number");
static_assert(is_number_v<int&>, "reference to int is a number");
static_assert(is_number_v<int&&>, "rvalue reference to int is a number");
static_assert(is_number_v<const int>, "const int is a number");
static_assert(is_number_v<const int&>, "const int is a number");
static_assert(is_number_v<volatile int>, "volatile int is a number");
static_assert(is_number_v<long long>, "long long is a number");
static_assert(is_number_v<float>, "float is a number");
```

```
43 static_assert(is_number_v<double>, "double is a number");
44
45 // TODO: Uncomment
46 static_assert(!is_number_v<uint8_t*>, "pointer to uint8_t is not a number");
47 static_assert(!is_number_v<int*>, "pointer to int is not a number");
48 static_assert(!is_number_v<int[2]>, "array of int is not a number");
49 static_assert(!is_number_v<bool>, "bool is not a number");
50 static_assert(!is_number_v<std::string>, "string is not a number");
51 static_assert(!is_number_v<std::vector<int>>, "vector of int is not a number");
52
53
54 /** is_character_v tests: */
55 static_assert(is_character_v<char>, "char is a character");
56 static_assert(is_character_v<char&>, "reference to char is a character");
57 static_assert(is_character_v<char&&>, "rvalue reference to char is a character");
58 static_assert(is_character_v<const char>, "const char is a character");
59 static_assert(is_character_v<const char&>, "const reference to char is a character");
60 static_assert(is_character_v<volatile char>, "volatile char is a character");
61
62 // TODO: Uncomment
63 static_assert(!is_character_v<char[2]>, "array of char is not a character");
64 static_assert(!is_character_v<char*>, "pointer to char is not a character");
65 static_assert(!is_character_v<const char*>, "pointer to const char is not a character");
66 static_assert(!is_character_v<uint8_t>, "uint8_t is not a character");
67 static_assert(!is_character_v<int8_t>, "int8_t is not a character");
68 static_assert(!is_character_v<int>, "int is not a character");
69 static_assert(!is_character_v<float>, "float is not a character");
70 static_assert(!is_character_v<double>, "double is not a character");
71 static_assert(!is_character_v<std::string>, "string is not a character");
72 static_assert(!is_character_v<std::vector<char>>, "vector of char is not a character");
73
74
75 /** is_string_v tests: */
76 static_assert(is_string_v<std::string>, "string is a string");
77 static_assert(is_string_v<std::string&>, "reference to a string is a string");
78 static_assert(is_string_v<std::string&&>, "rvalue reference to a string is a string");
79 static_assert(is_string_v<const std::string>, "const string is a string");
80 static_assert(is_string_v<const std::string&>, "const string reference is a string");
81 static_assert(is_string_v<char*>, "mutable C string is a string");
82 static_assert(is_string_v<const char*>, "const C string is a string");
83 static_assert(is_string_v<const char* const>, "const const C string is a string");
84 static_assert(is_string_v<char[2]>, "array of char is a string");
85 static_assert(is_string_v<char[7]>, "array of char is a string");
86
87 // TODO: Uncomment
88 static_assert(!is_string_v<std::string[]>, "array of string is not a string");
89 static_assert(!is_string_v<char**>, "pointer to mutable C string is not a string");
90 static_assert(!is_string_v<int>, "int is not a string");
91 static_assert(!is_string_v<float>, "float is not a string");
92 static_assert(!is_string_v<double>, "double is not a string");
93 static_assert(!is_string_v<std::vector<char>>, "vector of char is not a string");
94
95
96 /** is_container_v tests: */
97 static_assert(is_container_v<std::vector<int>>, "vector of int is a container");
98 static_assert(is_container_v<std::vector<int>&>, "reference to vector of int is a container");
99 static_assert(is_container_v<std::vector<int>&&>, "rvalue reference to vector of int is a
  ↪container");
100 static_assert(is_container_v<const std::vector<int>>, "const vector of int is a container");
101 static_assert(is_container_v<const std::vector<int>&>, "const reference of a vector of int is a
  ↪container");
```

```cpp
static_assert(is_container_v<std::vector<char>>, "vector of char is a container");
static_assert(is_container_v<std::vector<std::string>>, "vector of string is a container");
static_assert(is_container_v<std::initializer_list<int>>, "initializer list of int is a container");
static_assert(is_container_v<std::set<int>>, "set is a container");
static_assert(is_container_v<std::map<int, int>>, "map is a container");
static_assert(is_container_v<int[2]>, "array of int is a container");

// TODO: Uncomment
static_assert(!is_container_v<bool>, "bool is not a container");
static_assert(!is_container_v<char>, "char is not a container");
static_assert(!is_container_v<int>, "int is not a container");
static_assert(!is_container_v<float>, "float is not a container");
static_assert(!is_container_v<double>, "double is not a container");
static_assert(!is_container_v<int*>, "pointer to int is not a container");
static_assert(!is_container_v<std::string>, "string is not a container");

//tuple
static_assert(!is_tuple_v<std::vector<int>>, "vector of int  is not a tuple");
static_assert(!is_tuple_v<std::vector<char>>, "vector of char  is not a tuple");
static_assert(!is_tuple_v<std::vector<std::string>>, "vector of string  is not a tuple");
static_assert(!is_tuple_v<std::initializer_list<int>>, "initializer list of int  is not a tuple");
static_assert(!is_tuple_v<std::set<int>>, "set  is not a tuple");
static_assert(!is_tuple_v<std::map<int, int>>, "map  is not a tuple");
static_assert(!is_tuple_v<int[2]>, "array of int is not a tuple");
static_assert(!is_tuple_v<std::vector<std::tuple<std::tuple<int>, double, int, std::string>>>,
↪"Vector of tuples is not a tuple");

static_assert(!is_tuple_v<bool>, "bool is not a tuple");
static_assert(!is_tuple_v<char>, "char is not a tuple");
static_assert(!is_tuple_v<int>, "int is not a tuple");
static_assert(!is_tuple_v<float>, "float is not a tuple");
static_assert(!is_tuple_v<double>, "double is not a tuple");
static_assert(!is_tuple_v<int*>, "pointer to int is not a tuple");
static_assert(!is_tuple_v<std::string>, "string is not a tuple");

static_assert(is_tuple_v<std::tuple<int>>, "Scalar tuple is a tuple");
static_assert(is_tuple_v<std::tuple<int, int, int, int>>, "N tuple is a tuple");
static_assert(is_tuple_v<std::tuple<std::tuple<int>, double, int, std::string>>, "N tuple is a
↪tuple");

TEST_CASE("Tests for meta library are compile-time only") { CHECK(true); }
```

Listing 8: ./CMakeLists.txt

```
project(static_visitor CXX)

set(CMAKE_EXPORT_COMPILE_COMMANDS ON)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

include(sanitizers.cmake)
include(doctest.cmake)

include_directories(include)

enable_testing()

add_subdirectory(src)
```

Listing 9: ./README.md

```
# Static Visitor for JSON input/output

Motivation: we often want to serialize and load data from our datastructures with as little overhead and maintenance as possible, thus a generic solution is preferred. There are many libraries for performing reflection over generic data structures, but they are often intrusive and C++ standard still does not have a solution. Instead of reflection, we can use the static visitor pattern to provide low level access to data members. Static visitor uses compile-time polymorphism, it does not rely on central inheritance, adding support is trivial and it does not penalize runtime if unused.

JSON format is chosen as it is very popular among many modern frameworks.

The library consists of:
- Meta predicates for checking the basic types that JSON supports (bool, number, text, array/container, object).
- Customizing adapters wrapping input/output streams: `json_ostream` and `json_istream`.
- Output and input operators coupling `json_iostream`s with JSON data types.
- User-defined types (`struct`s and `class`es) need to have `accept` member function template calling arbitrary visitor with the content of their member fields.
- `json_writer_t` and `json_reader_t` implement the visitor pattern with member function templates `visit(std::string name, Value value)` to handle individual fields with given `name` and `value`. In principle, `json_writer_t` functionality can be part of `json_ostream` adapter, but they are separated here in order to separate visitor implementation from I/O operations.

Proposed plan:

1. Focus on one unit test section at a time and implement one feature at a time.
2. Start with [meta_test.cpp](src/meta_test.cpp): uncomment static assertions and implement the corresponding meta predicates until all assertions pass.
3. Implement the output functionality completely first ([json_output.hpp](include/json_output.hpp)) to satisfy tests in [json_output_test.cpp](src/json_output_test.cpp).

4. Compare `json_writer_t` and `printer` visitor developed for DSEL calculator during previous extended exercise.

5. Implement the input support ([json_input.hpp](include/json_input.hpp), similar to [json_output.hpp](include/json_output.hpp)) to satisfy tests in [json_input_test.cpp](src/json_input_test.cpp).

6. If time permits, extend the library to support `std::tuple` types: add a meta predicate to check if the type is created using `std::tuple` template, add a unit test for tuple output, implement tuple output using variadic function templates (either recursion or [tag-dispatch](https://en.cppreference.com/w/cpp/utility/integer_sequence) and [fold expressions](https://en.cppreference.com/w/cpp/language/fold)). Reuse as much as possible (e.g. call `json_writer_t::visit` method for each element in the tuple).
   For example, given `auto t = std::make_tuple(3,3.14,"pi");` then `json_ostream{std::cout} << t;` should produce `{"1":3,"2":3.14,"3":"pi"}`.
```

Listing 10: ./src/CMakeLists.txt

```
add_executable(meta_test meta_test.cpp)
target_link_libraries(meta_test PRIVATE doctest::doctest_with_main)

add_executable(json_output_test json_output_test.cpp)
target_link_libraries(json_output_test PRIVATE doctest::doctest_with_main)

add_executable(json_input_test json_input_test.cpp)
target_link_libraries(json_input_test PRIVATE doctest::doctest_with_main)

```

```
add_test(NAME meta_test COMMAND meta_test)
add_test(NAME json_output_test COMMAND json_output_test)
add_test(NAME json_input_test COMMAND json_input_test)
```