

Source Listing Advanced Programming 2

Martin Zimmer Kristensen

June 7, 2016

```

1  /*
2   Name: Martin Zimmer Kristensen
3   Compiler: clang version 3.9.0
4       clang++ -std=c++14 -Wall -g -pthread main.cpp -o main.out
5   json.h: detects whether an object contains a parse function and if it does
6   calls it. If it does not contain a parse function we consider it as basic
7   type and parse it to/from the file.
8   type-check.h: used by json.h for detection of types.
9   detection.h: used by type-check.h for detection.
10  hourly_16.json has to be in same directory as executable
11  */
12
13  #include "city-statistics.h"
14  #include "country-statistics.h"
15  #include "element.h"
16  #include "json.h"
17  #include <algorithm> // for_each, find_if
18  #include <chrono>
19  #include <future> // async
20  #include <vector>
21
22  using namespace std::chrono;
23  using Element = typename WeatherData::Element;
24  using CityStatistics = typename Statistics::CityStatistics;
25  using CountryStatistics = typename Statistics::CountryStatistics;
26
27  // 6b - demonstration
28  int main() {
29      auto deserializer =
30          js::Deserializer{"hourly_16.json"};
31      auto elements = std::vector<Element>{};
32      while (!deserializer.eof) {
33          auto element = Element{};
34          deserializer.parse(element);
35          elements.push_back(element);
36      }
37      // 7 - compute city statistics
38      auto city_statistics = std::vector<CityStatistics>{};
39      // function for computing city statistics
40      auto compute_cities = [&city_statistics =
41          city_statistics](const auto &elements) noexcept {
42          auto compute_city = [] (const auto &element) noexcept {
43              auto avg_temp_day = 0.00, avg_temp_night = 0.00;
44              auto daytime_from = "12:00:00", nighttime_from = "00:00:00";
45              auto count = 0;
46              std::for_each(
47                  element.data.begin(), element.data.end(),
48                  [
49                      &avg_temp_night = avg_temp_night, &avg_temp_day = avg_temp_day,
50                      daytime_from = daytime_from, nighttime_from = nighttime_from,
51                      &count = count
52                  ](auto data) {
53                      ++count;
54                      if (daytime_from <= data.dt_txt->substr(11, std::string::npos)) {
55                          avg_temp_day += data.main.temp;
56                      } else {
57                          avg_temp_night += data.main.temp;
58                      }
59                  });
60              auto city_statistic =

```

```

61         CityStatistics{element.city.name, avg_temp_night / (count / 2),
62                     avg_temp_day / (count / 2)};
63     return city_statistic;
64 };
65 for (auto el : elements) {
66     city_statistics.push_back(compute_city(el));
67 }
68 };
69
70 // 7 - compute country statistics
71 auto country_statistics = std::vector<CountryStatistics>{};
72 // function for computing country statistics
73 auto compute_countries = [&country_statistics = country_statistics](
74     const auto &elements) noexcept {
75     auto compute_country = [&country_statistics = country_statistics](
76         const auto &element) noexcept {
77         auto country_statistic = CountryStatistics{};
78         auto it = std::find_if(
79             country_statistics.begin(), country_statistics.end(),
80             [element = element](const auto &country_stat) noexcept {
81                 return element.city.country.compare(country_stat.country) == 0;
82             });
83         int index;
84         if (it == country_statistics.end()) {
85             country_statistic.country = element.city.country;
86             country_statistics.push_back(country_statistic);
87             index = country_statistics.size() - 1;
88         } else {
89             index = std::distance(country_statistics.begin(), it);
90         }
91         std::for_each(element.data.begin(), element.data.end(), [
92             index = index, &country_statistics = country_statistics
93         ](const auto &data) noexcept {
94             if (data.main.temp_max > country_statistics[index].temp_max) {
95                 country_statistics[index].temp_max = data.main.temp_max;
96             } else if (data.main.temp_min < country_statistics[index].temp_min) {
97                 country_statistics[index].temp_min = data.main.temp_min;
98             }
99         });
100     };
101     std::for_each(elements.begin(), elements.end(), compute_country);
102 };
103
104 // 8 - concurrency features
105 auto t1 = high_resolution_clock::now();
106 auto future_countries =
107     std::async(std::launch::async, compute_countries, elements);
108 auto future_cities = std::async(std::launch::async, compute_cities, elements);
109 future_countries.wait();
110 future_cities.wait();
111
112 auto t2 = high_resolution_clock::now();
113 auto duration = duration_cast<microseconds>(t2 - t1).count();
114
115 std::cout << duration << std::endl;
116
117 country_statistics.clear();
118 city_statistics.clear();
119
120 t1 = high_resolution_clock::now();
121 compute_countries(elements);

```

```

122 compute_cities(elements);
123 t2 = high_resolution_clock::now();
124 duration = duration_cast<microseconds>(t2 - t1).count();
125
126 std::cout << duration;
127 /*
128     with 02:
129     async 7576188
130     sync 6841028
131     without optimizations:
132     async 21995028
133     sync 23839428
134 */
135
136 auto serializer =
137     js::Serializer{"citystat.json"};
138 auto serializer2 =
139     js::Serializer{"countrystat.json"};
140 auto cityparse = std::async(
141     std::launch::async,
142     [&serializer = serializer, city_statistics = city_statistics]() {
143         for (const auto element : city_statistics) {
144             serializer.parse(element);
145         }
146     });
147 auto countryparse = std::async(
148     std::launch::async,
149     [&serializer = serializer2, country_statistics = country_statistics]() {
150         for (const auto element : country_statistics) {
151             serializer.parse(element);
152         }
153     });
154 cityparse.wait();
155 countryparse.wait();
156 return 0;
157 }

```

Listing 2: json.h

```

1  #ifndef JSON_H
2  #define JSON_H
3
4  #include "type_check.h"
5  #include <fstream>
6  #include <iostream>
7  #include <sstream>
8  #include <tuple>
9
10 namespace js {
11 // 4 - tuple adapter
12 template <typename T, typename... Args> struct tuple_adapter {
13
14     std::array<const char *, sizeof...(Args)> names;
15     std::tuple<Args...> &data;
16     T &serializer;
17
18     template <std::size_t I, typename Last> auto parse_i() const {
19         serializer.parse(std::get<I>(data), names[I]);
20     }
21
22     template <std::size_t I, typename Head, typename Second, typename... Tail>
23     auto parse_i() const {

```

```

24     parse_i<I, Head>();
25     parse_i<I + 1, Second, Tail...>();
26 }
27
28 auto parse() const { parse_i<0, Args...>(); }
29 };
30
31 struct Deserializer {
32     std::ifstream file_stream;
33     std::stringstream string_stream;
34     std::string element;
35     bool eof = false;
36     int pos = 0;
37     Deserializer(std::string file_path) {
38         file_stream.open(file_path, std::ifstream::in);
39     }
40
41     // find the name of the type
42     size_t end_of_type(const std::string &name) {
43         auto end_of_type = element.find("'", pos + 2);
44         auto actual_type = element.substr(pos + 2, end_of_type - pos - 2);
45         if (actual_type.compare(name) == 0) {
46             return end_of_type + 1;
47         }
48         return 0;
49     }
50
51     // root object unnamed
52     template <typename T>
53     typename std::enable_if<
54         t_chk::has_parse<T, decltype(std::declval<Deserializer &>())>::value,
55         void>::type
56     parse(T &obj) {
57         getline(file_stream, element);
58         if (element.size() == 0) {
59             eof = true;
60             file_stream.close();
61         } else {
62             string_stream.str(element);
63             pos = 0;
64             obj.parse(*this);
65             pos++;
66         }
67     }
68     // objects with parse function
69     template <typename T>
70     typename std::enable_if<
71         t_chk::has_parse<T, decltype(std::declval<Deserializer &>())>::value,
72         void>::type
73     parse(T &obj, const std::string name) {
74         auto new_pos = end_of_type(name);
75         if (new_pos != 0) {
76             pos = new_pos + 1;
77             obj.parse(*this);
78             pos++;
79         }
80     }
81
82     // string value types
83     template <typename T>
84     typename std::enable_if<t_chk::is_string<T>::value>::type

```

```

85     parse(T &value, const std::string name) {
86         auto new_pos = end_of_type(name);
87         if (new_pos != 0) {
88             pos = new_pos;
89             auto end_of_value = element.find("'", pos + 2);
90             value = element.substr(pos + 2, end_of_value - pos - 2);
91             pos = end_of_value + 1;
92         }
93     }
94
95     // char value types
96     template <typename T>
97     typename std::enable_if<t_chk::is_character<T>::value>::type
98     parse(T &value, const std::string name) {
99         auto new_pos = end_of_type(name);
100        if (new_pos != 0) {
101            pos = new_pos;
102            size_t end_of_value = element.find("'", pos + 2);
103            value = element[pos + 2];
104            pos = end_of_value + 1;
105        }
106    }
107
108    // numerical value types
109    template <typename T>
110    typename std::enable_if<t_chk::is_numeric<T>::value>::type
111    parse(T &value, const std::string name) {
112        auto new_pos = end_of_type(name);
113        if (new_pos != 0) {
114            pos = new_pos + 1;
115            string_stream.seekg(pos);
116            string_stream >> value;
117            pos = string_stream.tellg();
118        }
119    }
120
121    // container of string value types
122    template <typename T>
123    typename std::enable_if<t_chk::is_string_container<T>::value>::type
124    parse(T &value, const std::string name) {
125        using container_type =
126            typename std::iterator_traits<t_chk::iterated_t<T>>::value_type;
127        size_t new_pos = end_of_type(name) + 2;
128        pos = new_pos;
129        while (element[pos - 1] != ']') {
130            auto el = container_type{};
131            auto end_of_value = element.find("'", pos + 2);
132            el = element.substr(pos + 1, end_of_value - pos - 1);
133            pos = end_of_value + 2;
134            value.insert(value.end(), el);
135        }
136    }
137
138    // container of numerical value types
139    template <typename T>
140    typename std::enable_if<t_chk::is_numeric_container<T>::value>::type
141    parse(T &value, const std::string name) noexcept {
142        using container_type =
143            typename std::iterator_traits<t_chk::iterated_t<T>>::value_type;
144        size_t new_pos = end_of_type(name) + 2;
145        pos = new_pos;

```

```

146     while (element[pos - 1] != ']') {
147         auto el = container_type{};
148         string_stream.seekg(pos);
149         string_stream >> el;
150         pos = string_stream.tellg();
151         this->insert(value, el);
152         pos++;
153     }
154     insert_pos = 0;
155 }
156
157 // call insert if exists
158 template <typename T, typename C>
159 typename std::enable_if<t_chk::is_insertable<T, C>::value>::type
160 insert(T &container, C &val) {
161     container.insert(container.begin(), val);
162 }
163
164 // else use subscript operator
165 int insert_pos = 0;
166 template <typename T, typename C> auto insert(T &container, C &val) {
167     container[insert_pos++] = val;
168 }
169
170 // container of bool value types
171 template <typename T>
172 typename std::enable_if<t_chk::is_bool_container<T>::value>::type
173 parse(T &value, const std::string name) noexcept {
174     using container_type =
175         typename std::iterator_traits<t_chk::iterated_t<T>>::value_type;
176     size_t new_pos = end_of_type(name) + 2;
177     pos = new_pos;
178     while (element[pos - 1] != ']') {
179         auto el = container_type{};
180         el = element[pos] == 't' ? true : false;
181         if (el) {
182             pos += 5;
183         } else {
184             pos += 6;
185         }
186         value.insert(value.end(), el);
187     }
188 }
189
190 // container of objects with parse function
191 template <typename T>
192 typename std::enable_if<
193     t_chk::is_container<T>::value &&
194     t_chk::has_parse<
195         typename std::iterator_traits<t_chk::iterated_t<T>>::value_type,
196         decltype(std::declval<Deserializer &>())>::value,
197         void>::type
198 parse(T &value, const std::string name) {
199     using container_type =
200         typename std::iterator_traits<t_chk::iterated_t<T>>::value_type;
201     size_t new_pos = end_of_type(name) + 2;
202     pos = new_pos;
203     while (element[pos - 1] != ']') {
204         auto el = container_type{};
205         el.parse(*this);
206         value.insert(value.end(), el);

```

```

207     pos += 2;
208 }
209 }
210
211 // tuple object
212 template <typename... Args>
213 auto parse(std::array<const char *, sizeof...(Args)> n,
214           std::tuple<Args...> &t) {
215     tuple_adapter<decltype(*this), Args...>{n, t, *this}.parse();
216 }
217 };
218
219 struct Serializer {
220     std::string element;
221     std::ofstream stream;
222     Serializer(std::string file_path) {
223         stream.open(file_path);
224         element = "";
225     }
226
227 // root object unnamed
228 template <typename T>
229 typename std::enable_if<
230     t_chk::has_parse<T, decltype(std::declval<Serializer &>())>::value,
231     void>::type
232 parse(const T &obj) noexcept {
233     element += "{";
234     obj.parse(*this);
235     element += "}\n";
236     stream << element;
237     element = "";
238 }
239 // objects with parse function
240 template <typename T>
241 typename std::enable_if<
242     t_chk::has_parse<T, decltype(std::declval<Serializer &>())>::value,
243     void>::type
244 parse(const T &obj, const std::string name) noexcept {
245     if (element.back() != '{' && element.back() != '[') {
246         element += ',';
247     }
248     element += "\"" + name + "\":{";
249     obj.parse(*this);
250     element += "}";
251 }
252
253 // string or character value type
254 template <typename T>
255 typename std::enable_if<t_chk::is_string<T>::value ||
256     t_chk::is_character<T>::value>::type
257 parse(const T &value, const std::string name) noexcept {
258     if (element.back() != '{' && element.back() != '[') {
259         element += ',';
260     }
261     element += "\"" + name + "\":\"" + value + "\"";
262 }
263
264 // numerical value type
265 template <typename T>
266 typename std::enable_if<t_chk::is_numeric<T>::value>::type
267 parse(const T &value, const std::string name) noexcept {

```



```

268     if (element.back() != '{' && element.back() != '[') {
269         element += ',';
270     }
271     element += "\"" + name + "\": " + std::to_string(value);
272 }
273
274 // bool value type
275 template <typename T>
276 typename std::enable_if<t_chk::is_bool<T>::value>::type
277 parse(const T &value, const std::string name) {
278     if (element.back() != '{' && element.back() != '[') {
279         element += ',';
280     }
281     element += "\"" + name + "\": ";
282     element += value ? "true" : "false";
283 }
284
285 // container of strings
286 template <typename T>
287 typename std::enable_if<t_chk::is_string_container<T>::value>::type
288 parse(const T &value, const std::string name) noexcept {
289     if (element.back() != '{' && element.back() != '[') {
290         element += ',';
291     }
292     element += "\"" + name + "\": [";
293     for (auto val : value) {
294         if (element.back() != '[')
295             element += ',';
296         element += "\"" + val + "\"";
297     }
298     element += "];";
299 }
300
301 // container of numerical values
302 template <typename T>
303 typename std::enable_if<t_chk::is_numeric_container<T>::value>::type
304 parse(const T &value, const std::string name) noexcept {
305     if (element.back() != '{' && element.back() != '[') {
306         element += ',';
307     }
308     element += "\"" + name + "\": [";
309     for (auto val : value) {
310         if (element.back() != '[')
311             element += ',';
312         element += std::to_string(val);
313     }
314     element += "];";
315 }
316
317 // container of bools
318 template <typename T>
319 typename std::enable_if<t_chk::is_bool_container<T>::value>::type
320 parse(const T &value, const std::string name) noexcept {
321     if (element.back() != '{' && element.back() != '[') {
322         element += ',';
323     }
324     element += "\"" + name + "\": [";
325     for (auto val : value) {
326         if (element.back() != '[')
327             element += ',';
328         element += val ? "true" : "false";

```

```

329     }
330     element += "];";
331 }
332
333 // container of objects with parse function
334 template <typename T>
335 typename std::enable_if<
336     t_chk::is_container<T>::value &&
337     t_chk::has_parse<
338         typename std::iterator_traits<t_chk::iterated_t<T>>::value_type,
339         decltype(std::declval<Serializer &>())>::value,
340         void>::type
341 parse(const T &value, const std::string name) noexcept {
342     if (element.back() != '{' && element.back() != '[') {
343         element += ',,';
344     }
345     element += "\"" + name + "\":[";
346     for (auto val : value) {
347         if (element.back() != '[')
348             element += ',,';
349         element += "{";
350         val.parse(*this);
351         element += "}";
352     }
353     element += "];";
354 }
355
356 // tuple object
357 template <typename... Args>
358 auto parse(std::array<const char *, sizeof...(Args)> n,
359     std::tuple<Args...> t) noexcept {
360     tuple_adapter<decltype(*this), Args...>{n, t, *this}.parse();
361 }
362 };
363
364 } // js
365
366 #endif /* JSON_H */

```

Listing 3: type_check.h

```

1  #ifndef TYPE_CHECK_H
2  #define TYPE_CHECK_H
3
4  #include "detection.h"
5  #include <iostream>
6
7  namespace t_chk {
8
9  // has insert function
10 template <typename T, typename C>
11 using insert_t = decltype(
12     std::declval<T &>().insert(std::declval<T &>().end(), std::declval<C &>()));
13
14 template <typename T, typename C>
15 using is_insertable = is_detected<insert_t, T, C>;
16
17 // iterable type
18 template <typename C>
19 using iterated_t = decltype(std::begin(std::declval<C &>()));
20
21 // container types

```

```

22 template <typename C> using is_container = is_detected<iterated_t, C>;
23
24 // enumerate the character types to consider:
25 template <typename T, typename C = typename std::remove_cv<T>::type>
26 struct is_character
27     : std::conditional_t<std::is_same<C, char>::value ||
28                         std::is_same<C, unsigned char>::value ||
29                         std::is_same<C, signed char>::value ||
30                         std::is_same<C, wchar_t>::value,
31                         std::true_type, std::false_type> {};
32
33 // enumerate the numeric types to consider:
34 template <typename T, typename C = typename std::remove_cv<T>::type>
35 struct is_integer
36     : std::conditional_t<
37         std::is_same<C, short int>::value ||
38         std::is_same<C, unsigned short int>::value ||
39         std::is_same<C, int>::value || std::is_same<C, long int>::value ||
40         std::is_same<C, unsigned long int>::value ||
41         std::is_same<C, long long int>::value ||
42         std::is_same<C, unsigned long long int>::value ||
43         std::is_same<C, double>::value || std::is_same<C, float>::value,
44         std::true_type, std::false_type> {};
45
46 // boolean type
47 template <typename T, typename C = typename std::remove_cv<T>::type>
48 struct is_bool : std::conditional_t<std::is_same<C, bool>::value,
49                                     std::true_type, std::false_type> {};
50
51 // numerical types: primary template
52 template <typename T, typename = void> struct is_numeric : std::false_type {};
53
54 // numerical types: specialization
55 template <typename T>
56 struct is_numeric<T, std::enable_if_t<is_integer<T>::value>> : std::true_type {
57 };
58
59 // strings types: primary template
60 template <typename T, typename = void> struct is_string : std::false_type {};
61
62 // pointer types: specialization
63 template <typename T>
64 struct is_string<
65     T, std::enable_if_t<std::is_pointer<std::remove_reference_t<T>>::value>>
66     : is_character<std::remove_pointer_t<std::remove_reference_t<T>>> {};
67
68 // string containers: specialization
69 template <typename T>
70 struct is_string<T, typename std::enable_if<is_container<T>::value>::type>
71     : is_character<typename std::iterator_traits<iterated_t<T>>::value_type> {};
72
73 // container of numerical values: primary template
74 template <typename T, typename = void>
75 struct is_string_container : std::false_type {};
76
77 // container of numerical values: specialization
78 template <typename T>
79 struct is_string_container<
80     T, typename std::enable_if<is_container<T>::value>::type>
81     : is_string<typename std::iterator_traits<iterated_t<T>>::value_type> {};
82

```

```

83 // container of numerical values: primary template
84 template <typename T, typename = void>
85 struct is_numeric_container : std::false_type {};
86
87 // container of numerical values: specialization
88 template <typename T>
89 struct is_numeric_container<
90     T, typename std::enable_if<is_container<T>::value>::type>
91     : is_numeric<typename std::iterator_traits<iterated_t<T>>::value_type> {};
92
93 // container of bool primary template
94 template <typename T, typename = void>
95 struct is_bool_container : std::false_type {};
96
97 // container of bool specialization
98 template <typename T>
99 struct is_bool_container<T,
100     typename std::enable_if<is_container<T>::value>::type>
101     : is_bool<typename std::iterator_traits<iterated_t<T>>::value_type> {};
102
103 // parse function
104 template <class T, class C>
105 using parsable_t = decltype(std::declval<T &>().parse(std::declval<C &>()));
106
107 // parse function
108 template <class T, class C> using has_parse = is_detected<parsable_t, T, C>;
109 } // t_chk
110
111 #endif /* TYPE_CHECK_H */

```

Listing 4: detection.h

```

1 #ifndef DETECTION_H
2 #define DETECTION_H
3
4 #include <type_traits>
5
6 /* Detection idiom toolkit (like in N4502 and lecture 9) */
7 struct nonesuch {
8     nonesuch() = delete;
9     ~nonesuch() = delete;
10     nonesuch(nonesuch const &) = delete;
11     void operator=(nonesuch const &) = delete;
12 };
13
14 template <typename...> using void_t = void;
15
16 template <class Default, class, template <class...> class Op, class... Args>
17 struct detector {
18     using value_t = std::false_type;
19     using type = Default;
20 };
21
22 template <class Default, template <class...> class Op, class... Args>
23 struct detector<Default, void_t<Op<Args...>>, Op, Args...> {
24     using value_t = std::true_type;
25     using type = Op<Args...>;
26 };
27
28 template <template <class...> class Op, class... Args>
29 using is_detected = typename detector<nonesuch, void, Op, Args...>::value_t;
30

```

```

31 template <template <class...> class Op, class... Args>
32 constexpr bool is_detected_v = is_detected<Op, Args...>::value;
33
34 template <template <class...> class Op, class... Args>
35 using is_detected_t = typename detector<nonesuch, void, Op, Args...>::type;
36
37 #endif /* DETECTION_H */

```

Listing 5: element.h

```

1  #ifndef ELEMENT_H
2  #define ELEMENT_H
3
4  #include "city.h"
5  #include "data.h"
6  #include "json.h"
7  #include <vector>
8
9  namespace WeatherData {
10 struct Element {
11     City city;
12     int time;
13     std::vector<Data> data;
14
15     auto parse(js::Serializer &serializer) const noexcept {
16         serializer.parse(city, "city");
17         serializer.parse(time, "time");
18         serializer.parse(data, "data");
19     }
20
21     auto parse(js::Deserializer &serializer) {
22         serializer.parse(city, "city");
23         serializer.parse(time, "time");
24         serializer.parse(data, "data");
25     }
26 };
27
28 } // WeatherData
29
30 #endif /* ELEMENT_H */

```

Listing 6: city.h

```

1  #ifndef CITY_H
2  #define CITY_H
3
4  #include "coord.h"
5  #include "json.h"
6  #include <string>
7
8  namespace WeatherData {
9  struct City {
10     int id;
11     std::string name;
12     std::string country;
13     Coord coord;
14
15     auto parse(js::Serializer &serializer) const noexcept {
16         serializer.parse(id, "id");
17         serializer.parse(name, "name");
18         serializer.parse(country, "country");
19         serializer.parse(coord, "coord");

```

```

20     }
21
22     auto parse(js::Deserializer &serializer) {
23         serializer.parse(id, "id");
24         serializer.parse(name, "name");
25         serializer.parse(country, "country");
26         serializer.parse(coord, "coord");
27     }
28 };
29
30 } // WeatherData
31 #endif /* CITY_H */

```

Listing 7: coord.h

```

1  #ifndef COORD_H
2  #define COORD_H
3
4  #include "json.h"
5
6  namespace WeatherData {
7
8  struct Coord {
9      float lon, lat;
10
11     auto parse(js::Serializer &serializer) const noexcept {
12         serializer.parse(lon, "lon");
13         serializer.parse(lat, "lat");
14     }
15     auto parse(js::Deserializer &serializer) {
16         serializer.parse(lon, "lon");
17         serializer.parse(lat, "lat");
18     }
19 };
20
21 } // WeatherData
22
23 #endif /* COORD_H */

```

Listing 8: data.h

```

1  #ifndef DATA_H
2  #define DATA_H
3
4  #include "clouds.h"
5  #include "json.h"
6  #include "main.h"
7  #include "rain.h"
8  #include "snow.h"
9  #include "sys.h"
10 #include "weather.h"
11 #include "wind.h"
12 #include <algorithm>
13 #include <memory>
14 #include <string>
15 #include <vector>
16
17 using str_shared_ptr = std::shared_ptr<std::string>;
18 namespace WeatherData {
19 struct Data {
20     int dt;
21     Main main;

```

```

22     std::vector<Weather> weather;
23     Clouds clouds;
24     Wind wind;
25     std::shared_ptr<Snow> snow;
26     std::shared_ptr<Rain> rain;
27     Sys sys;
28     str_shared_ptr dt_txt = std::make_shared<std::string>();
29     static std::vector<str_shared_ptr> shared_dt_txt;
30
31     auto dt_txt_make_shared() {
32         auto it = std::find_if(shared_dt_txt.begin(), shared_dt_txt.end(),
33             [&this_dt_txt = dt_txt](const auto &dt_txt) {
34                 return *this_dt_txt == *dt_txt;
35             });
36         int index;
37         if (it == shared_dt_txt.end()) {
38             shared_dt_txt.push_back(std::make_shared<std::string>(*this->dt_txt));
39             index = shared_dt_txt.size() - 1;
40         } else {
41             index = std::distance(shared_dt_txt.begin(), it);
42         }
43         return shared_dt_txt[index];
44     }
45
46     auto parse(js::Serializer &serializer) const noexcept {
47         serializer.parse(dt, "dt");
48         serializer.parse(main, "main");
49         serializer.parse(weather, "weather");
50         serializer.parse(clouds, "clouds");
51         serializer.parse(wind, "wind");
52         if (snow)
53             serializer.parse(*snow, "snow");
54         if (rain)
55             serializer.parse(*rain, "rain");
56         serializer.parse(sys, "sys");
57         serializer.parse(*dt_txt, "dt_txt");
58     }
59
60     // No shared objects 17.1
61     auto parse(js::Deserializer &serializer) {
62         serializer.parse(dt, "dt");
63         serializer.parse(main, "main");
64         serializer.parse(weather, "weather");
65         serializer.parse(clouds, "clouds");
66         serializer.parse(wind, "wind");
67         auto temp_snow = Snow{};
68         serializer.parse(temp_snow, "snow");
69         if (temp_snow.three_h != 0) {
70             snow = temp_snow.make_shared();
71         }
72         auto temp_rain = Rain{};
73         serializer.parse(temp_rain, "rain");
74         if (temp_rain.three_h != 0) {
75             rain = temp_rain.make_shared();
76         }
77         // 16.9
78         serializer.parse(sys, "sys");
79         serializer.parse(*dt_txt, "dt_txt");
80         this->dt_txt = dt_txt_make_shared();
81         // 12
82     }

```

```

83 };
84 std::vector<str_shared_ptr> Data::shared_dt_txt{};
85 } // WeatherData
86
87 #endif /* DATA_H */

```

Listing 9: main.h

```

1  #ifndef MAIN_H
2  #define MAIN_H
3
4  #include "json.h"
5  #include <algorithm>
6  #include <memory>
7  #include <vector>
8
9  namespace WeatherData {
10 struct Main {
11     float temp, temp_min, temp_max, pressure, sea_level, grnd_level;
12     int humidity;
13     std::shared_ptr<float> temp_kf;
14     static std::vector<std::shared_ptr<float>> shared_temp_kf;
15
16     auto temp_kf_make_shared(float &temp_temp_kf) {
17         auto it = std::find_if(shared_temp_kf.begin(), shared_temp_kf.end(),
18             [temp_temp_kf = temp_temp_kf](const auto &temp_kf) {
19             return temp_temp_kf == *temp_kf;
20         });
21         int index;
22         if (it == shared_temp_kf.end()) {
23             shared_temp_kf.push_back(std::make_shared<float>(temp_temp_kf));
24             index = shared_temp_kf.size() - 1;
25         } else {
26             index = std::distance(shared_temp_kf.begin(), it);
27         }
28         return shared_temp_kf[index];
29     }
30
31     auto parse(js::Serializer &serializer) const noexcept {
32         serializer.parse(temp, "temp");
33         serializer.parse(temp_min, "temp_min");
34         serializer.parse(temp_max, "temp_max");
35         serializer.parse(pressure, "pressure");
36         serializer.parse(sea_level, "sea_level");
37         serializer.parse(grnd_level, "grnd_level");
38         serializer.parse(humidity, "humidity");
39         if (temp_kf)
40             serializer.parse(*temp_kf, "temp_kf");
41     }
42     auto parse(js::Deserializer &serializer) {
43         serializer.parse(temp, "temp");
44         serializer.parse(temp_min, "temp_min");
45         serializer.parse(temp_max, "temp_max");
46         serializer.parse(pressure, "pressure");
47         serializer.parse(sea_level, "sea_level");
48         serializer.parse(grnd_level, "grnd_level");
49         serializer.parse(humidity, "humidity");
50         float temp_temp_kf = 0.0;
51         serializer.parse(temp_temp_kf, "temp_kf");
52         if (temp_temp_kf != 0.0) {
53             temp_kf = temp_kf_make_shared(temp_temp_kf);
54         }

```



```

55     }
56 };
57 std::vector<std::shared_ptr<float>> Main::shared_temp_kf{};
58 } // WeatherData
59
60 #endif /* MAIN_H */

```

Listing 10: weather.h

```

1  #ifndef WEATHER_H
2  #define WEATHER_H
3
4  #include "json.h"
5  #include <algorithm>
6  #include <memory>
7  #include <string>
8  #include <vector>
9
10 namespace WeatherData {
11 struct Weather {
12     static std::vector<std::shared_ptr<Weather>> shared_objects;
13     int id;
14     std::string main, description, icon;
15
16     auto parse(js::Serializer &serializer) const noexcept {
17         serializer.parse(id, "id");
18         serializer.parse(main, "main");
19         serializer.parse(description, "description");
20         serializer.parse(icon, "icon");
21     }
22     auto parse(js::Deserializer &serializer) {
23         serializer.parse(id, "id");
24         serializer.parse(main, "main");
25         serializer.parse(description, "description");
26         serializer.parse(icon, "icon");
27     }
28     std::shared_ptr<Weather> make_shared() {
29
30         auto it = std::find_if(shared_objects.begin(), shared_objects.end(),
31                               [obj = this](const auto weather) {
32                                   return weather->id == obj->id &&
33                                         weather->main == obj->main &&
34                                         weather->description == obj->description &&
35                                         weather->icon == obj->icon;
36                               });
37         int index;
38         if (it == shared_objects.end()) {
39             shared_objects.push_back(std::make_shared<Weather>(*this));
40             index = shared_objects.size() - 1;
41         } else {
42             index = std::distance(shared_objects.begin(), it);
43         }
44         return shared_objects[index];
45     }
46 };
47
48 std::vector<std::shared_ptr<Weather>> shared_objects{};
49 } // WeatherData
50
51 #endif /* WEATHER_H */

```

Listing 11: clouds.h

```

1  #ifndef CLOUDS_H
2  #define CLOUDS_H
3
4  #include "json.h"
5  namespace WeatherData {
6  struct Clouds {
7      int all;
8
9      auto parse(js::Serializer &serializer) const noexcept {
10         serializer.parse(all, "all");
11     }
12     auto parse(js::Deserializer &serializer) { serializer.parse(all, "all"); }
13 };
14
15 } // WeatherData
16
17 #endif /* CLOUDS_H */

```

Listing 12: wind.h

```

1  #ifndef WIND_H
2  #define WIND_H
3
4  #include "json.h"
5
6  namespace WeatherData {
7  struct Wind {
8      float speed = 0, deg = 0;
9
10     auto parse(js::Serializer &serializer) const noexcept {
11         serializer.parse(speed, "speed");
12         serializer.parse(deg, "deg");
13     }
14     auto parse(js::Deserializer &serializer) {
15         serializer.parse(speed, "speed");
16         serializer.parse(deg, "deg");
17     }
18 };
19
20 } // WeatherData
21
22 #endif /* WIND_H */

```

Listing 13: snow.h

```

1  #ifndef SNOW_H
2  #define SNOW_H
3
4  #include "json.h"
5  #include <algorithm>
6  #include <memory>
7  #include <vector>
8
9  namespace WeatherData {
10 struct Snow {
11     static std::vector<std::shared_ptr<Snow>> shared_objects;
12     float three_h = 0;
13     auto parse(js::Serializer &serializer) const noexcept {
14         serializer.parse(three_h, "3h");
15     }
16     auto parse(js::Deserializer &serializer) { serializer.parse(three_h, "3h"); }

```

```

17 auto make_shared() {
18     auto it = std::find_if(shared_objects.begin(), shared_objects.end(),
19                             [obj = this](const auto &snow) {
20                                 return obj->three_h == snow->three_h;
21                             });
22     int index;
23     if (it == shared_objects.end()) {
24         shared_objects.push_back(std::make_shared<Snow>(*this));
25         index = shared_objects.size() - 1;
26     } else {
27         index = std::distance(shared_objects.begin(), it);
28     }
29     return shared_objects[index];
30 }
31 };
32 std::vector<std::shared_ptr<Snow>> Snow::shared_objects{};
33 } // WeatherData
34
35 #endif /* SNOW_H */

```

Listing 14: rain.h

```

1 #ifndef RAIN_H
2 #define RAIN_H
3
4 #include "json.h"
5 #include <algorithm>
6 #include <memory>
7 #include <vector>
8
9 namespace WeatherData {
10 struct Rain {
11     static std::vector<std::shared_ptr<Rain>> shared_objects;
12     float three_h = 0;
13     auto parse(js::Serializer &serializer) const noexcept {
14         serializer.parse(three_h, "3h");
15     }
16     auto parse(js::Deserializer &serializer) { serializer.parse(three_h, "3h"); }
17     auto make_shared() {
18         auto it = std::find_if(shared_objects.begin(), shared_objects.end(),
19                                 [obj = this](const auto &el_rain) {
20                                     return obj->three_h == el_rain->three_h;
21                                 });
22         int index;
23         if (it == shared_objects.end()) {
24             shared_objects.push_back(std::make_shared<Rain>(*this));
25             index = shared_objects.size() - 1;
26         } else {
27             index = std::distance(shared_objects.begin(), it);
28         }
29         return shared_objects[index];
30     }
31 };
32
33 std::vector<std::shared_ptr<Rain>> Rain::shared_objects{};
34 } // WeatherData
35 #endif /* RAIN_H */

```

Listing 15: sys.h

```

1 #ifndef SYS_H
2 #define SYS_H

```

```

3
4 #include "json.h"
5 #include <string.h>
6
7 namespace WeatherData {
8 struct Sys {
9     std::string pod;
10
11     auto parse(js::Serializer &serializer) const noexcept {
12         serializer.parse(pod, "pod");
13     }
14     auto parse(js::Deserializer &serializer) { serializer.parse(pod, "pod"); }
15 };
16
17 } // WeatherData
18
19 #endif /* SYS_H */

```

Listing 16: city-statistics.h

```

1 #ifndef CITY_STATISTICS_H
2 #define CITY_STATISTICS_H
3
4 #include "element.h"
5 #include "json.h"
6 #include <string>
7
8 using Element = WeatherData::Element;
9 namespace Statistics {
10 struct CityStatistics {
11     std::string city;
12     float avg_temp_night, avg_temp_day;
13     CityStatistics(){};
14     CityStatistics(std::string city, double avg_temp_night, double avg_temp_day)
15         : city(city), avg_temp_night(avg_temp_night),
16         avg_temp_day(avg_temp_day){};
17     void parse(js::Serializer &serializer) const noexcept {
18         serializer.parse(city, "city");
19         serializer.parse(avg_temp_night, "avg_temp_night");
20         serializer.parse(avg_temp_day, "avg_temp_day");
21     }
22     void parse(js::Deserializer &serializer) {
23         serializer.parse(city, "city");
24         serializer.parse(avg_temp_night, "avg_temp_night");
25         serializer.parse(avg_temp_day, "avg_temp_day");
26     }
27 };
28 } // Statistics
29
30 #endif /* CITY-STATISTICS_H */

```

Listing 17: country-statistics.h

```

1 #ifndef COUNTRY_STATISTICS_H
2 #define COUNTRY_STATISTICS_H
3
4 #include "json.h"
5 #include <string>
6
7 namespace Statistics {
8
9 struct CountryStatistics {

```

```
10     std::string country;
11     double temp_min = 9999.99, temp_max = 00.00;
12
13     void parse(js::Serializer &serializer) const noexcept {
14         serializer.parse(country, "country");
15         serializer.parse(temp_max, "temp_max");
16         serializer.parse(temp_min, "temp_min");
17     }
18     void parse(js::Deserializer &serializer) {
19         serializer.parse(country, "country");
20         serializer.parse(temp_max, "temp_max");
21         serializer.parse(temp_min, "temp_min");
22     }
23 };
24 } // Statistics
25
26 #endif /* COUNTRY-STATISTICS_H */
```
