Source Listing Advanced Programming 2

Martin Zimmer Kristensen

June 7, 2016

Listing 1: main.cpp

```cpp
/*
  Name: Martin Zimmer Kristensen
  Compiler: clang version 3.9.0
        clang++ -std=c++14 -Wall -g -pthread main.cpp -o main.out
  json.h: detects whether an object contains a parse function and if it does
  calls it. If it does not contain a parse function we consider it as basic
  type and parse it to/from the file.
  type-check.h: used by json.h for detection of types.
  detection.h: used by type-check.h for detection.
  hourly_16.json has to be in same directory as executable
 */

#include "city-statistics.h"
#include "country-statistics.h"
#include "element.h"
#include "json.h"
#include <algorithm> // for_each, find_if
#include <chrono>
#include <future> // async
#include <vector>

using namespace std::chrono;
using Element = typename WeatherData::Element;
using CityStatistics = typename Statistics::CityStatistics;
using CountryStatistics = typename Statistics::CountryStatistics;

// 6b - demonstration
int main() {
  auto deserializer =
      js::Deserializer{"hourly_16.json"};
  auto elements = std::vector<Element>{};
  while (!deserializer.eof) {
    auto element = Element{};
    deserializer.parse(element);
    elements.push_back(element);
  }
  // 7 - compute city statistics
  auto city_statistics = std::vector<CityStatistics>{};
  // function for computing city statistics
  auto compute_cities = [&city_statistics =
                             city_statistics](const auto &elements) noexcept {
    auto compute_city = [](const auto &element) noexcept {
      auto avg_temp_day = 0.00, avg_temp_night = 0.00;
      auto daytime_from = "12:00:00", nighttime_from = "00:00:00";
      auto count = 0;
      std::for_each(
          element.data.begin(), element.data.end(),
          [
                &avg_temp_night = avg_temp_night, &avg_temp_day = avg_temp_day,
                daytime_from = daytime_from, nighttime_from = nighttime_from,
                &count = count
          ](auto data) {
            ++count;
            if (daytime_from <= data.dt_txt->substr(11, std::string::npos)) {
              avg_temp_day += data.main.temp;
            } else {
              avg_temp_night += data.main.temp;
            }
          });
      auto city_statistic =
          CityStatistics{element.city.name, avg_temp_night / (count / 2),
                         avg_temp_day / (count / 2)};
      return city_statistic;
    };
    for (auto el : elements) {
      city_statistics.push_back(compute_city(el));
    }
  };

  // 7 - compute country statistics
  auto country_statistics = std::vector<CountryStatistics>{};
  // function for computing country statistics
  auto compute_countries = [&country_statistics = country_statistics](
      const auto &elements) noexcept {
    auto compute_country = [&country_statistics = country_statistics](
        const auto &element) noexcept {
      auto country_statistic = CountryStatistics{};
      auto it = std::find_if(
          country_statistics.begin(), country_statistics.end(),
          [element = element](const auto &country_stat) noexcept {
            return element.city.country.compare(country_stat.country) == 0;
          });
      int index;
      if (it == country_statistics.end()) {
        country_statistic.country = element.city.country;
        country_statistics.push_back(country_statistic);
        index = country_statistics.size() - 1;
      } else {
        index = std::distance(country_statistics.begin(), it);
      }
      std::for_each(element.data.begin(), element.data.end(), [
        index = index, &country_statistics = country_statistics
      ](const auto &data) noexcept {
        if (data.main.temp_max > country_statistics[index].temp_max) {
          country_statistics[index].temp_max = data.main.temp_max;
        } else if (data.main.temp_min < country_statistics[index].temp_min) {
          country_statistics[index].temp_min = data.main.temp_min;
        }
      });
    };
    std::for_each(elements.begin(), elements.end(), compute_country);
  };

  // 8 - concurrency features
  auto t1 = high_resolution_clock::now();
  auto future_countries =
      std::async(std::launch::async, compute_countries, elements);
  auto future_cities = std::async(std::launch::async, compute_cities, elements);
  future_countries.wait();
  future_cities.wait();

  auto t2 = high_resolution_clock::now();
  auto duration = duration_cast<microseconds>(t2 - t1).count();

  std::cout << duration << std::endl;

  country_statistics.clear();
  city_statistics.clear();

  t1 = high_resolution_clock::now();
  compute_countries(elements);
  compute_cities(elements);
  t2 = high_resolution_clock::now();
  duration = duration_cast<microseconds>(t2 - t1).count();

  std::cout << duration;
  /*
    with O2:
    async 7576188
    sync 6841028
    without optimizations:
    async 21995028
    sync 23839428
  */

  auto serializer =
      js::Serializer{"citystat.json"};
  auto serializer2 =
      js::Serializer{"countrystat.json"};
  auto cityparse = std::async(
      std::launch::async,
      [&serializer = serializer, city_statistics = city_statistics ]() {
        for (const auto element : city_statistics) {
          serializer.parse(element);
        }
      });
  auto countryparse = std::async(
      std::launch::async,
      [&serializer = serializer2, country_statistics = country_statistics ]() {
        for (const auto element : country_statistics) {
          serializer.parse(element);
        }
      });
  cityparse.wait();
  countryparse.wait();
  return 0;
}
```

Listing 2: json.h

```
#ifndef JSON_H
#define JSON_H

#include "type_check.h"
#include <fstream>
#include <iostream>
#include <sstream>
#include <tuple>

namespace js {
// 4 - tuple adapter
template <typename T, typename... Args> struct tuple_adapter {

  std::array<const char *, sizeof...(Args)> names;
  std::tuple<Args...> &data;
  T &serializer;

  template <std::size_t I, typename Last> auto parse_i() const {
    serializer.parse(std::get<I>(data), names[I]);
  }

  template <std::size_t I, typename Head, typename Second, typename... Tail>
  auto parse_i() const {
    parse_i<I, Head>();
    parse_i<I + 1, Second, Tail...>();
  }

  auto parse() const { parse_i<0, Args...>(); }
};

struct Deserializer {
  std::ifstream file_stream;
  std::istringstream string_stream;
  std::string element;
  bool eof = false;
  int pos = 0;
  Deserializer(std::string file_path) {
    file_stream.open(file_path, std::ifstream::in);
  }

  // find the name of the type
  size_t end_of_type(const std::string &name) {
    auto end_of_type = element.find(’"’, pos + 2);
    auto actual_type = element.substr(pos + 2, end_of_type - pos - 2);
    if (actual_type.compare(name) == 0) {
      return end_of_type + 1;
    }
    return 0;
  }

  // root object unnamed
  template <typename T>
  typename std::enable_if<
      t_chk::has_parse<T, decltype(std::declval<Deserializer &>())>::value,
      void>::type
  parse(T &obj) {
    getline(file_stream, element);
    if (element.size() == 0) {
      eof = true;
      file_stream.close();
    } else {
      string_stream.str(element);
      pos = 0;
      obj.parse(*this);
      pos++;
    }
  }
  // objects with parse function
  template <typename T>
  typename std::enable_if<
      t_chk::has_parse<T, decltype(std::declval<Deserializer &>())>::value,
      void>::type
  parse(T &obj, const std::string name) {
    auto new_pos = end_of_type(name);
    if (new_pos != 0) {
      pos = new_pos + 1;
      obj.parse(*this);
      pos++;
    }
  }

  // string value types
  template <typename T>
  typename std::enable_if<t_chk::is_string<T>::value>::type
```

```
parse(T &value, const std::string name) {
  auto new_pos = end_of_type(name);
  if (new_pos != 0) {
    pos = new_pos;
    auto end_of_value = element.find(’"’, pos + 2);
    value = element.substr(pos + 2, end_of_value - pos - 2);
    pos = end_of_value + 1;
  }
}

// char value types
template <typename T>
typename std::enable_if<t_chk::is_character<T>::value>::type
parse(T &value, const std::string name) {
  auto new_pos = end_of_type(name);
  if (new_pos != 0) {
    pos = new_pos;
    size_t end_of_value = element.find(’"’, pos + 2);
    value = element[pos + 2];
    pos = end_of_value + 1;
  }
}

// numerical value types
template <typename T>
typename std::enable_if<t_chk::is_numeric<T>::value>::type
parse(T &value, const std::string name) {
  auto new_pos = end_of_type(name);
  if (new_pos != 0) {
    pos = new_pos + 1;
    string_stream.seekg(pos);
    string_stream >> value;
    pos = string_stream.tellg();
  }
}

// container of string value types
template <typename T>
typename std::enable_if<t_chk::is_string_container<T>::value>::type
parse(T &value, const std::string name) {
  using container_type =
      typename std::iterator_traits<t_chk::iterated_t<T>>::value_type;
  size_t new_pos = end_of_type(name) + 2;
  pos = new_pos;
  while (element[pos - 1] != ’]’) {
    auto el = container_type{};
    auto end_of_value = element.find(’"’, pos + 2);
    el = element.substr(pos + 1, end_of_value - pos - 1);
    pos = end_of_value + 2;
    value.insert(value.end(), el);
  }
}

// container of numerical value types
template <typename T>
typename std::enable_if<t_chk::is_numeric_container<T>::value>::type
parse(T &value, const std::string name) noexcept {
  using container_type =
      typename std::iterator_traits<t_chk::iterated_t<T>>::value_type;
  size_t new_pos = end_of_type(name) + 2;
  pos = new_pos;
```

```
    while (element[pos - 1] != ’]’) {
      auto el = container_type{};
      string_stream.seekg(pos);
      string_stream >> el;
      pos = string_stream.tellg();
      this->insert(value, el);
      pos++;
    }
    insert_pos = 0;
  }

  // call insert if exists
  template <typename T, typename C>
  typename std::enable_if<t_chk::is_insertable<T, C>::value>::type
  insert(T &container, C &val) {
    container.insert(container.begin(), val);
  }

  // else use subscript operator
  int insert_pos = 0;
  template <typename T, typename C> auto insert(T &container, C &val) {
    container[insert_pos++] = val;
  }

  // container of bool value types
  template <typename T>
  typename std::enable_if<t_chk::is_bool_container<T>::value>::type
  parse(T &value, const std::string name) noexcept {
    using container_type =
        typename std::iterator_traits<t_chk::iterated_t<T>>::value_type;
    size_t new_pos = end_of_type(name) + 2;
    pos = new_pos;
    while (element[pos - 1] != ’]’) {
      auto el = container_type{};
      el = element[pos] == ’t’ ? true : false;
      if (el) {
        pos += 5;
      } else {
        pos += 6;
      }
      value.insert(value.end(), el);
    }
  }

  // container of objects with parse function
  template <typename T>
  typename std::enable_if<
      t_chk::is_container<T>::value &&
          t_chk::has_parse<
              typename std::iterator_traits<t_chk::iterated_t<T>>::value_type,
              decltype(std::declval<Deserializer &>())>::value,
      void>::type
  parse(T &value, const std::string name) {
    using container_type =
        typename std::iterator_traits<t_chk::iterated_t<T>>::value_type;
    size_t new_pos = end_of_type(name) + 2;
    pos = new_pos;
    while (element[pos - 1] != ’]’) {
      auto el = container_type{};
      el.parse(*this);
      value.insert(value.end(), el);
      pos += 2;
    }
  }

  // tuple object
  template <typename... Args>
  auto parse(std::array<const char *, sizeof...(Args)> n,
             std::tuple<Args...> &t) {
    tuple_adapter<decltype(*this), Args...>{n, t, *this}.parse();
  }
};

struct Serializer {
  std::string element;
  std::ofstream stream;
  Serializer(std::string file_path) {
    stream.open(file_path);
    element = "";
  }

  // root object unnamed
  template <typename T>
  typename std::enable_if<
      t_chk::has_parse<T, decltype(std::declval<Serializer &>())>::value,
      void>::type
  parse(const T &obj) noexcept {
    element += "{";
    obj.parse(*this);
    element += "}\n";
    stream << element;
    element = "";
  }
  // objects with parse function
  template <typename T>
  typename std::enable_if<
      t_chk::has_parse<T, decltype(std::declval<Serializer &>())>::value,
      void>::type
  parse(const T &obj, const std::string name) noexcept {
    if (element.back() != '{' && element.back() != '[') {
      element += ',';
    }
    element += "\"" + name + "\":{";
    obj.parse(*this);
    element += "}";
  }

  // string or character value type
  template <typename T>
  typename std::enable_if<t_chk::is_string<T>::value ||
                          t_chk::is_character<T>::value>::type
  parse(const T &value, const std::string name) noexcept {
    if (element.back() != '{' && element.back() != '[') {
      element += ',';
    }
    element += "\"" + name + "\":\"" + value + "\"";
  }

  // numerical value type
  template <typename T>
  typename std::enable_if<t_chk::is_numeric<T>::value>::type
  parse(const T &value, const std::string name) noexcept {
```

```cpp
    if (element.back() != '{' && element.back() != '[') {
      element += ',';
    }
    element += "\"" + name + "\":" + std::to_string(value);
  }

  // bool value type
  template <typename T>
  typename std::enable_if<t_chk::is_bool<T>::value>::type
  parse(const T &value, const std::string name) {
    if (element.back() != '{' && element.back() != '[') {
      element += ',';
    }
    element += "\"" + name + "\":";
    element += value ? "true" : "false";
  }

  // container of strings
  template <typename T>
  typename std::enable_if<t_chk::is_string_container<T>::value>::type
  parse(const T &value, const std::string name) noexcept {
    if (element.back() != '{' && element.back() != '[') {
      element += ',';
    }
    element += "\"" + name + "\":[";
    for (auto val : value) {
      if (element.back() != '[')
        element += ',';
      element += '"' + val + '"';
    }
    element += "]";
  }

  // container of numerical values
  template <typename T>
  typename std::enable_if<t_chk::is_numeric_container<T>::value>::type
  parse(const T &value, const std::string name) noexcept {
    if (element.back() != '{' && element.back() != '[') {
      element += ',';
    }
    element += "\"" + name + "\":[";
    for (auto val : value) {
      if (element.back() != '[')
        element += ',';
      element += std::to_string(val);
    }
    element += "]";
  }

  // container of bools
  template <typename T>
  typename std::enable_if<t_chk::is_bool_container<T>::value>::type
  parse(const T &value, const std::string name) noexcept {
    if (element.back() != '{' && element.back() != '[') {
      element += ',';
    }
    element += "\"" + name + "\":[";
    for (auto val : value) {
      if (element.back() != '[')
        element += ',';
      element += val ? "true" : "false";
    }
    element += "]";
  }

  // container of objects with parse function
  template <typename T>
  typename std::enable_if<
      t_chk::is_container<T>::value &&
          t_chk::has_parse<
              typename std::iterator_traits<t_chk::iterated_t<T>>::value_type,
              decltype(std::declval<Serializer &>())>::value,
      void>::type
  parse(const T &value, const std::string name) noexcept {
    if (element.back() != '{' && element.back() != '[') {
      element += ',';
    }
    element += "\"" + name + "\":[";
    for (auto val : value) {
      if (element.back() != '[')
        element += ',';
      element += "{";
      val.parse(*this);
      element += "}";
    }
    element += "]";
  }

  // tuple object
  template <typename... Args>
  auto parse(std::array<const char *, sizeof...(Args)> n,
             std::tuple<Args...> t) noexcept {
    tuple_adapter<decltype(*this), Args...>{n, t, *this}.parse();
  }
};

} // js

#endif /* JSON_H */
```

Listing 3: type_check.h

```
#ifndef TYPE_CHECK_H
#define TYPE_CHECK_H

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

namespace t_chk {

// has insert function
template <typename T, typename C>
using insert_t = decltype(
    std::declval<T &>().insert(std::declval<T &>().end(), std::declval<C &>()));

template <typename T, typename C>
using is_insertable = is_detected<insert_t, T, C>;

// iteratable type
template <typename C>
using iterated_t = decltype(std::begin(std::declval<C &>()));

// container types
```

```cpp
template <typename C> using is_container = is_detected<iterated_t, C>;

// enumerate the character types to consider:
template <typename T, typename C = typename std::remove_cv<T>::type>
struct is_character
    : std::conditional_t<std::is_same<C, char>::value ||
                             std::is_same<C, unsigned char>::value ||
                             std::is_same<C, signed char>::value ||
                             std::is_same<C, wchar_t>::value,
                         std::true_type, std::false_type> {};

// enumerate the numeric types to consider:
template <typename T, typename C = typename std::remove_cv<T>::type>
struct is_integer
    : std::conditional_t<
          std::is_same<C, short int>::value ||
              std::is_same<C, unsigned short int>::value ||
              std::is_same<C, int>::value || std::is_same<C, long int>::value ||
              std::is_same<C, unsigned long int>::value ||
              std::is_same<C, long long int>::value ||
              std::is_same<C, unsigned long long int>::value ||
              std::is_same<C, double>::value || std::is_same<C, float>::value,
          std::true_type, std::false_type> {};

// boolean type
template <typename T, typename C = typename std::remove_cv<T>::type>
struct is_bool : std::conditional_t<std::is_same<C, bool>::value,
                                    std::true_type, std::false_type> {};

// numerical types: primary template
template <typename T, typename = void> struct is_numeric : std::false_type {};

// numerical types: specialization
template <typename T>
struct is_numeric<T, std::enable_if_t<is_integer<T>::value>> : std::true_type {
};

// strings types: primary template
template <typename T, typename = void> struct is_string : std::false_type {};

// pointer types: specialization
template <typename T>
struct is_string<
    T, std::enable_if_t<std::is_pointer<std::remove_reference_t<T>>::value>>
    : is_character<std::remove_pointer_t<std::remove_reference_t<T>>> {};

// string containers: specialization
template <typename T>
struct is_string<T, typename std::enable_if<is_container<T>::value>::type>
    : is_character<typename std::iterator_traits<iterated_t<T>>::value_type> {};

// container of numerical values: primary template
template <typename T, typename = void>
struct is_string_container : std::false_type {};

// container of numerical values: specialization
template <typename T>
struct is_string_container<
    T, typename std::enable_if<is_container<T>::value>::type>
    : is_string<typename std::iterator_traits<iterated_t<T>>::value_type> {};

```

```cpp
// container of numerical values: primary template
template <typename T, typename = void>
struct is_numeric_container : std::false_type {};

// container of numerical values: specialization
template <typename T>
struct is_numeric_container<
    T, typename std::enable_if<is_container<T>::value>::type>
    : is_numeric<typename std::iterator_traits<iterated_t<T>>::value_type> {};

// container of bool primary template
template <typename T, typename = void>
struct is_bool_container : std::false_type {};

// container of bool specialization
template <typename T>
struct is_bool_container<T,
                         typename std::enable_if<is_container<T>::value>::type>
    : is_bool<typename std::iterator_traits<iterated_t<T>>::value_type> {};

// parse function
template <class T, class C>
using parsable_t = decltype(std::declval<T &>().parse(std::declval<C &>()));

// parse function
template <class T, class C> using has_parse = is_detected<parsable_t, T, C>;
} // t_chk

#endif /* TYPE_CHECK_H */
```

Listing 4: detection.h

```cpp
#ifndef DETECTION_H
#define DETECTION_H

#include <type_traits>

/* Detection idiom toolkit (like in N4502 and lecture 9) */
struct nonesuch {
  nonesuch() = delete;
  ~nonesuch() = delete;
  nonesuch(nonesuch const &) = delete;
  void operator=(nonesuch const &) = delete;
};

template <typename...> using void_t = void;

template <class Default, class, 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;

```

```cpp
template <template <class...> class Op, class... Args>
constexpr bool is_detected_v = is_detected<Op, Args...>::value;

template <template <class...> class Op, class... Args>
using is_detected_t = typename detector<nonesuch, void, Op, Args...>::type;

#endif /* DETECTION_H */
```

Listing 5: element.h

```cpp
#ifndef ELEMENT_H
#define ELEMENT_H

#include "city.h"
#include "data.h"
#include "json.h"
#include <vector>

namespace WeatherData {
struct Element {
  City city;
  int time;
  std::vector<Data> data;

  auto parse(js::Serializer &serializer) const noexcept {
    serializer.parse(city, "city");
    serializer.parse(time, "time");
    serializer.parse(data, "data");
  }

  auto parse(js::Deserializer &serializer) {
    serializer.parse(city, "city");
    serializer.parse(time, "time");
    serializer.parse(data, "data");
  }
};

} // WeatherData

#endif /* ELEMENT_H */
```

Listing 6: city.h

```cpp
#ifndef CITY_H
#define CITY_H

#include "coord.h"
#include "json.h"
#include <string>

namespace WeatherData {
struct City {
  int id;
  std::string name;
  std::string country;
  Coord coord;

  auto parse(js::Serializer &serializer) const noexcept {
    serializer.parse(id, "id");
    serializer.parse(name, "name");
    serializer.parse(country, "country");
    serializer.parse(coord, "coord");
  }

  auto parse(js::Deserializer &serializer) {
    serializer.parse(id, "id");
    serializer.parse(name, "name");
    serializer.parse(country, "country");
    serializer.parse(coord, "coord");
  }
};

} // WeatherData
#endif /* CITY_H */
```

Listing 7: coord.h

```
#ifndef COORD_H
#define COORD_H

#include "json.h"

namespace WeatherData {

struct Coord {
  float lon, lat;

  auto parse(js::Serializer &serializer) const noexcept {
    serializer.parse(lon, "lon");
    serializer.parse(lat, "lat");
  }
  auto parse(js::Deserializer &serializer) {
    serializer.parse(lon, "lon");
    serializer.parse(lat, "lat");
  }
};

} // WeatherData

#endif /* COORD_H */
```

Listing 8: data.h

```
#ifndef DATA_H
#define DATA_H

#include "clouds.h"
#include "json.h"
#include "main.h"
#include "rain.h"
#include "snow.h"
#include "sys.h"
#include "weather.h"
#include "wind.h"
#include <algorithm>
#include <memory>
#include <string>
#include <vector>

using str_shared_ptr = std::shared_ptr<std::string>;
namespace WeatherData {
struct Data {
  int dt;
  Main main;
```

```cpp
std::vector<Weather> weather;
Clouds clouds;
Wind wind;
std::shared_ptr<Snow> snow;
std::shared_ptr<Rain> rain;
Sys sys;
str_shared_ptr dt_txt = std::make_shared<std::string>();
static std::vector<str_shared_ptr> shared_dt_txt;

auto dt_txt_make_shared() {
  auto it = std::find_if(shared_dt_txt.begin(), shared_dt_txt.end(),
                         [&this_dt_txt = dt_txt](const auto &dt_txt) {
                           return *this_dt_txt == *dt_txt;
                         });
  int index;
  if (it == shared_dt_txt.end()) {
    shared_dt_txt.push_back(std::make_shared<std::string>(*this->dt_txt));
    index = shared_dt_txt.size() - 1;
  } else {
    index = std::distance(shared_dt_txt.begin(), it);
  }
  return shared_dt_txt[index];
}

auto parse(js::Serializer &serializer) const noexcept {
  serializer.parse(dt, "dt");
  serializer.parse(main, "main");
  serializer.parse(weather, "weather");
  serializer.parse(clouds, "clouds");
  serializer.parse(wind, "wind");
  if (snow)
    serializer.parse(*snow, "snow");
  if (rain)
    serializer.parse(*rain, "rain");
  serializer.parse(sys, "sys");
  serializer.parse(*dt_txt, "dt_txt");
}

// No shared objects 17.1
auto parse(js::Deserializer &serializer) {
  serializer.parse(dt, "dt");
  serializer.parse(main, "main");
  serializer.parse(weather, "weather");
  serializer.parse(clouds, "clouds");
  serializer.parse(wind, "wind");
  auto temp_snow = Snow{};
  serializer.parse(temp_snow, "snow");
  if (temp_snow.three_h != 0) {
    snow = temp_snow.make_shared();
  }
  auto temp_rain = Rain{};
  serializer.parse(temp_rain, "rain");
  if (temp_rain.three_h != 0) {
    rain = temp_rain.make_shared();
  }
  // 16.9
  serializer.parse(sys, "sys");
  serializer.parse(*dt_txt, "dt_txt");
  this->dt_txt = dt_txt_make_shared();
  // 12
}
```

```
};
std::vector<str_shared_ptr> Data::shared_dt_txt{};
} // WeatherData

#endif /* DATA_H */
```

Listing 9: main.h

```
#ifndef MAIN_H
#define MAIN_H

#include "json.h"
#include <algorithm>
#include <memory>
#include <vector>

namespace WeatherData {
struct Main {
  float temp, temp_min, temp_max, pressure, sea_level, grnd_level;
  int humidity;
  std::shared_ptr<float> temp_kf;
  static std::vector<std::shared_ptr<float>> shared_temp_kf;

  auto temp_kf_make_shared(float &temp_temp_kf) {
    auto it = std::find_if(shared_temp_kf.begin(), shared_temp_kf.end(),
                           [temp_temp_kf = temp_temp_kf](const auto &temp_kf) {
                             return temp_temp_kf == *temp_kf;
                           });
    int index;
    if (it == shared_temp_kf.end()) {
      shared_temp_kf.push_back(std::make_shared<float>(temp_temp_kf));
      index = shared_temp_kf.size() - 1;
    } else {
      index = std::distance(shared_temp_kf.begin(), it);
    }
    return shared_temp_kf[index];
  }

  auto parse(js::Serializer &serializer) const noexcept {
    serializer.parse(temp, "temp");
    serializer.parse(temp_min, "temp_min");
    serializer.parse(temp_max, "temp_max");
    serializer.parse(pressure, "pressure");
    serializer.parse(sea_level, "sea_level");
    serializer.parse(grnd_level, "grnd_level");
    serializer.parse(humidity, "humidity");
    if (temp_kf)
      serializer.parse(*temp_kf, "temp_kf");
  }
  auto parse(js::Deserializer &serializer) {
    serializer.parse(temp, "temp");
    serializer.parse(temp_min, "temp_min");
    serializer.parse(temp_max, "temp_max");
    serializer.parse(pressure, "pressure");
    serializer.parse(sea_level, "sea_level");
    serializer.parse(grnd_level, "grnd_level");
    serializer.parse(humidity, "humidity");
    float temp_temp_kf = 0.0;
    serializer.parse(temp_temp_kf, "temp_kf");
    if (temp_temp_kf != 0.0) {
      temp_kf = temp_kf_make_shared(temp_temp_kf);
    }
  }
};
std::vector<std::shared_ptr<float>> Main::shared_temp_kf{};
} // WeatherData

#endif /* MAIN_H */
```

Listing 10: weather.h

```
#ifndef WEATHER_H
#define WEATHER_H

#include "json.h"
#include <algorithm>
#include <memory>
#include <string>
#include <vector>

namespace WeatherData {
struct Weather {
  static std::vector<std::shared_ptr<Weather>> shared_objects;
  int id;
  std::string main, description, icon;

  auto parse(js::Serializer &serializer) const noexcept {
    serializer.parse(id, "id");
    serializer.parse(main, "main");
    serializer.parse(description, "description");
    serializer.parse(icon, "icon");
  }
  auto parse(js::Deserializer &serializer) {
    serializer.parse(id, "id");
    serializer.parse(main, "main");
    serializer.parse(description, "description");
    serializer.parse(icon, "icon");
  }
  std::shared_ptr<Weather> make_shared() {

    auto it = std::find_if(shared_objects.begin(), shared_objects.end(),
                           [obj = this](const auto weather) {
                             return weather->id == obj->id &&
                                    weather->main == obj->main &&
                                    weather->description == obj->description &&
                                    weather->icon == obj->icon;
                           });
    int index;
    if (it == shared_objects.end()) {
      shared_objects.push_back(std::make_shared<Weather>(*this));
      index = shared_objects.size() - 1;
    } else {
      index = std::distance(shared_objects.begin(), it);
    }
    return shared_objects[index];
  }
};

std::vector<std::shared_ptr<Weather>> shared_objects{};
} // WeatherData

#endif /* WEATHER_H */
```

Listing 11: clouds.h

```
#ifndef CLOUDS_H
#define CLOUDS_H

#include "json.h"
namespace WeatherData {
struct Clouds {
  int all;

  auto parse(js::Serializer &serializer) const noexcept {
    serializer.parse(all, "all");
  }
  auto parse(js::Deserializer &serializer) { serializer.parse(all, "all"); }
};

} // WeatherData

#endif /* CLOUDS_H */
```

Listing 12: wind.h

```
#ifndef WIND_H
#define WIND_H

#include "json.h"

namespace WeatherData {
struct Wind {
  float speed = 0, deg = 0;

  auto parse(js::Serializer &serializer) const noexcept {
    serializer.parse(speed, "speed");
    serializer.parse(deg, "deg");
  }
  auto parse(js::Deserializer &serializer) {
    serializer.parse(speed, "speed");
    serializer.parse(deg, "deg");
  }
};

} // WeatherData

#endif /* WIND_H */
```

Listing 13: snow.h

```
#ifndef SNOW_H
#define SNOW_H

#include "json.h"
#include <algorithm>
#include <memory>
#include <vector>

namespace WeatherData {
struct Snow {
  static std::vector<std::shared_ptr<Snow>> shared_objects;
  float three_h = 0;
  auto parse(js::Serializer &serializer) const noexcept {
    serializer.parse(three_h, "3h");
  }
  auto parse(js::Deserializer &serializer) { serializer.parse(three_h, "3h"); }
  auto make_shared() {
    auto it = std::find_if(shared_objects.begin(), shared_objects.end(),
                           [obj = this](const auto &snow) {
                             return obj->three_h == snow->three_h;
                           });
    int index;
    if (it == shared_objects.end()) {
      shared_objects.push_back(std::make_shared<Snow>(*this));
      index = shared_objects.size() - 1;
    } else {
      index = std::distance(shared_objects.begin(), it);
    }
    return shared_objects[index];
  }
};
std::vector<std::shared_ptr<Snow>> Snow::shared_objects{};
} // WeatherData

#endif /* SNOW_H */
```

Listing 14: rain.h

```
#ifndef RAIN_H
#define RAIN_H

#include "json.h"
#include <algorithm>
#include <memory>
#include <vector>

namespace WeatherData {
struct Rain {
  static std::vector<std::shared_ptr<Rain>> shared_objects;
  float three_h = 0;
  auto parse(js::Serializer &serializer) const noexcept {
    serializer.parse(three_h, "3h");
  }
  auto parse(js::Deserializer &serializer) { serializer.parse(three_h, "3h"); }
  auto make_shared() {
    auto it = std::find_if(shared_objects.begin(), shared_objects.end(),
                           [obj = this](const auto &el_rain) {
                             return obj->three_h == el_rain->three_h;
                           });
    int index;
    if (it == shared_objects.end()) {
      shared_objects.push_back(std::make_shared<Rain>(*this));
      index = shared_objects.size() - 1;
    } else {
      index = std::distance(shared_objects.begin(), it);
    }
    return shared_objects[index];
  }
};

std::vector<std::shared_ptr<Rain>> Rain::shared_objects{};
} // WeatherData
#endif /* RAIN_H */
```

Listing 15: sys.h

```
#ifndef SYS_H
#define SYS_H
```

```

#include "json.h"
#include <string.h>

namespace WeatherData {
struct Sys {
  std::string pod;

  auto parse(js::Serializer &serializer) const noexcept {
    serializer.parse(pod, "pod");
  }
  auto parse(js::Deserializer &serializer) { serializer.parse(pod, "pod"); }
};

} // WeatherData

#endif /* SYS_H */
```

Listing 16: city-statistics.h

```
#ifndef CITY_STATISTICS_H
#define CITY_STATISTICS_H

#include "element.h"
#include "json.h"
#include <string>

using Element = WeatherData::Element;
namespace Statistics {
struct CityStatistics {
  std::string city;
  float avg_temp_night, avg_temp_day;
  CityStatistics(){};
  CityStatistics(std::string city, double avg_temp_night, double avg_temp_day)
      : city(city), avg_temp_night(avg_temp_night),
        avg_temp_day(avg_temp_day){};
  void parse(js::Serializer &serializer) const noexcept {
    serializer.parse(city, "city");
    serializer.parse(avg_temp_night, "avg_temp_night");
    serializer.parse(avg_temp_day, "avg_temp_day");
  }
  void parse(js::Deserializer &serializer) {
    serializer.parse(city, "city");
    serializer.parse(avg_temp_night, "avg_temp_night");
    serializer.parse(avg_temp_day, "avg_temp_day");
  }
};
} // Statistics

#endif /* CITY-STATISTICS_H */
```

Listing 17: country-statistics.h

```
#ifndef COUNTRY_STATISTICS_H
#define COUNTRY_STATISTICS_H

#include "json.h"
#include <string>

namespace Statistics {

struct CountryStatistics {
  std::string country;
  double temp_min = 9999.99, temp_max = 00.00;

  void parse(js::Serializer &serializer) const noexcept {
    serializer.parse(country, "country");
    serializer.parse(temp_max, "temp_max");
    serializer.parse(temp_min, "temp_min");
  }
  void parse(js::Deserializer &serializer) {
    serializer.parse(country, "country");
    serializer.parse(temp_max, "temp_max");
    serializer.parse(temp_min, "temp_min");
  }
};
} // Statistics

#endif /* COUNTRY-STATISTICS_H */
```