Source Listing for Extended Exercise 1

Jonas Sand Madsen and Simon Vandel Sillesen (room 0.2.18)

March 6, 2017

Listing 1: timeseries.h

```cpp
#include <vector>

namespace sw805f17g2 {
class TimeSeries {
public:
  TimeSeries(const std::vector<int> &data);
  static TimeSeries MakeRandom(const int length);
  void operator+=(const TimeSeries &other);

  long amplitude() const;

private:
  std::vector<int> data;
  int minimum;
  int maximum;

  friend bool operator<(const TimeSeries &left, const TimeSeries &right);
};

TimeSeries operator+(const TimeSeries &left, const TimeSeries &right);

inline bool operator<(const TimeSeries &left, const TimeSeries &right) {
  auto this_amp = left.amplitude();
  auto other_amp = right.amplitude();

  return this_amp < other_amp;
}
}
```

Listing 2: timeseries.cpp

```cpp
#include "timeseries.h"

#include <algorithm>
#include <climits>
#include <exception>
#include <functional>
#include <random>

using namespace sw805f17g2;

TimeSeries::TimeSeries(const std::vector<int> &data) : data(data) {
  auto temp_min = std::min_element(data.begin(), data.end());
  if (temp_min != data.end()) {
    minimum = *temp_min;
  } else {
    throw new std::invalid_argument("data");
  }

  // we are sure that the data parameter is not empty
  maximum = *std::max_element(data.begin(), data.end());
}

template <typename T>
void zip_with(std::vector<T> &left, const std::vector<T> &right,
              std::function<void(T &, const T &)> f) {
  // make sure left and right have the same length. Extend the smallest one.
  auto longer = left.size() > right.size() ? left.size() : right.size();
  // resize default initializes the new elements (e.g. 0 for int)
  left.resize(longer);

  auto l_cur = left.begin();
  auto r_cur = right.begin();
  for (; l_cur != left.end() && r_cur != right.end(); ++l_cur, ++r_cur) {
    f(*l_cur, *r_cur);
  }
}

void TimeSeries::operator+=(const TimeSeries &other) {
  zip_with<int>(this->data, other.data, [](int &a, const int &b) { a += b; });
}

TimeSeries operator+(TimeSeries left, const TimeSeries &right) {
  left += right;
  return left;
}

long TimeSeries::amplitude() const {
  return static_cast<long>(maximum) - minimum;
}

TimeSeries TimeSeries::MakeRandom(const int length) {
  std::random_device r;
  std::default_random_engine e1(r());
  std::uniform_int_distribution<int> uniform_dist(INT_MIN, INT_MAX);
  std::vector<int> data_buffer;
  data_buffer.reserve(length);
  for (size_t i = 0; i < length; i++) {
    data_buffer.push_back(uniform_dist(e1));
  }
  return TimeSeries(data_buffer);
}
```

Listing 3: main.cpp

```cpp
#include "timeseries.h"

#include <chrono>
#include <functional>
#include <iostream>
#include <memory>

using namespace sw805f17g2;

void timeit(std::function<void()> f) {
  auto t0 = std::chrono::high_resolution_clock::now();
  f();
  auto t1 = std::chrono::high_resolution_clock::now();
  std::cout
      << std::chrono::duration_cast<std::chrono::nanoseconds>(t1 - t0).count()
      << "nanosec\n";
}

// conclusion:
// without optimizations: raw pointer is the fastest.
// with optimizations: value is faster than raw pointer,
// generally, unique_ptr is the slowest.
int main() {
  // step 2
  const int num_timeseries = 10000;
  const int num_datapoints = 100;

  std::vector<TimeSeries> timeseries;
  timeseries.reserve(num_timeseries);
  for (size_t i = 0; i < num_timeseries; i++) {
    timeseries.emplace_back(TimeSeries::MakeRandom(num_datapoints));
  }

  auto random_timeseries = TimeSeries::MakeRandom(num_datapoints);

  std::vector<TimeSeries> timeseries1;
  timeseries1.reserve(num_timeseries);
  for (auto &elem : timeseries) {
    timeseries1.emplace_back(elem);
  }

  for (auto &ts : timeseries1) {
    ts += random_timeseries;
  }

  // std::for_each(timeseries.begin(), timeseries.end(), [](TimeSeries const&
  // elem) {
  //     std::cout << elem.amplitude() << std::endl;
  // });
  timeit([&]() { std::sort(timeseries1.begin(), timeseries1.end()); });
  // std::cout << std::endl;
  // std::for_each(timeseries.begin(), timeseries.end(), [](TimeSeries const&
  // elem) {
  //     std::cout << elem.amplitude() << std::endl;
  // });

  // step 3
  std::vector<std::unique_ptr<TimeSeries>> timeseries2;
  timeseries2.reserve(num_timeseries);
  for (auto &elem : timeseries) {
    timeseries2.emplace_back(new TimeSeries(elem));
  }

  for (auto &ts : timeseries2) {
    *ts += random_timeseries;
  }

  timeit([&]() {
    std::sort(timeseries2.begin(), timeseries2.end(),
              [](const std::unique_ptr<TimeSeries> &left,
                 const std::unique_ptr<TimeSeries> &right) {
                return *left < *right;
              });
  });

  // step 3.1 (raw pointer)
  // we believe this is faster than step3, because we the overhead from
  // unique_ptr
  std::vector<TimeSeries *> timeseries3;
  timeseries3.reserve(num_timeseries);
  for (auto &elem : timeseries) {
    timeseries3.emplace_back(new TimeSeries(elem));
  }

  for (auto &ts : timeseries3) {
    *ts += random_timeseries;
  }

  timeit([&]() {
    std::sort(timeseries3.begin(), timeseries3.end(),
              [](TimeSeries *const &left, TimeSeries *const &right) {
                return *left < *right;
              });
  });
}
```