Assignment 4: Iterators, Function Templates

C++ Programming Course, Summer Term 2018

Source Code

4-0: Watch these Talks

A talk at ACCU has just been published on YouTube that summarizes the relevance of STL algorithms and explains the mental model behind them really well:

- Jonathan Boccara: 105 STL Algorithms in Less Than an Hour

Everything in this talk agrees with the discussion of STL algorithms in our lab sessions, and the presentation is stellar.

> There is a trap that lies right at the beginning of the path to STL algorithms, and this trap is called for_each.

In another talk, Marshall Clow gives an introduction and some hints on how to implement good STL algorithm abstractions:

- Marshall Clow: STL Algorithms - why you should use them, and how to write your own

4-1: Algorithms / Function Templates

In the following, you will implement two algorithms as function templates.

Your implementations should be a combination of functions provided by the STL and must not contain explicit loops like `for` or `while`.

You may use any algorithm interface defined in the C++14 standard. For example, you may use `std::minmax` (since C++11) in your implementations, but not `std::for_each_n` (C++17).

References:

- C++ Algorithm Library: http://en.cppreference.com/w/cpp/algorithm
- C++ Iterator Library: http://en.cppreference.com/w/cpp/header/iterator

4-1-1: find_mean_rep

Implement a function template `cpppc::find_mean_rep` which accepts a range specified by two iterators in the `InputIterator` category and returns an iterator to the element in the range that is closest to the mean of the range.

Your implementation should be automatically more efficient for random access iterator ranges without specialization.

Function interface:

```
template <typename InputIter>
InputIter find_mean_rep(InputIter first, InputIter last);
```

Example:

```
std::vector<int> v { 1, 2, 3, 30, 40, 50 }; // mean: 21
auto closest_to_mean_it = cpppc::find_mean_rep(v.begin(), v.end());
// -> iterator at index 3 (|21-30| = 9)
```

4-1-2: histogram (RandomAccessIterator)

Implement a function template `cpppc::histogram` which accepts a range specified by two iterators in the `InputIterator` category and **replaces** each value by the number of its occurrences in the range.

Note that values of the input range are modified.

Values are unique in the result histogram and ordered by their first occurrence in the range. The function returns an iterator past the final element in the histogram.

Only integer values (int, long, size_t, ...) have to be supported.

Function interface:

```
template <typename RAIt>
RAIt histogram(RAIt first, RAIt last);
```

Example:

```cpp
std::vector<int> v { 1, 5, 5, 3, 4, 1, 5, 7 };
auto hist_end = cpppc::histogram(v.begin(), v.end());
// -> 1: 2 occurences
//    5: 3 occurences
//    5: skipped
//    ...
// -> { 2, 3, 1, 1, 1 }  <- occurrences
//      ^  ^  ^  ^  ^
//      |  |  |  |  |
//      1  5  3  4  7    <- value
//
// -> returns iterator at position 5
```

4-2: Custom Containers / Iterators

4-2-1: Sparse Array

Do we need swap, <, >, <=, >=?

Implement a sparse array (see https://en.wikipedia.org/wiki/Sparse_array) container that satisfies the `std::array` interface.

A sparse array is a usually large array with most elements set to a default value like 0. Instead of allocating memory for all values, only non-default values are actually allocated.

Different from vectors, arrays are static containers so their size does not change after instantiation. The values of its elements can be iterated and changed, but elements cannot be added or removed.

References:

- std::array in the C++ Standard Template Library: http://en.cppreference.com/w/cpp/container/array

```cpp
int default = 0;
cpppc::sparse_array<int, 10000> sa(default);
// no actual values in `sa` yet, all set to default
size_t sa_size = sa.size(); // -> 10000
assert(sa[345] == 0);       // returns `default` if no value
                            // set at position

sa[230] = 23;
assert(sa[230] == 23);
```

```
sa[420] = 42;
// two values stored in `sa`

// Must be compatible with STL algorithms:

auto found_42 = std::find(sa.begin(), sa.end(), 42);
// -> iterator at 'virtual' position 420 (the second 'real' value)
```

Notes:

- Your implementation has to detect assignment of a value reference. Try a temporary proxy: https://en.wikibooks.org/wiki/More_C++_Idioms/Temporary_Proxy.
- Obviously, the `sparse_array` cannot provide direct access to its underlying array (as it doesn't exist) so method `data()` is not provided.

4-2-2: Lazy Sequence

Testing?

Write a class template `lazy_sequence` that implements the sequence concept and is initialized with its size and a generator function.

A lazy sequence does not store elements (unlike containers), instead a generator function is used to create values only when they are accessed. It is not modifiable, similar to `const` sequential containers.

We define the `Sequence` concept, for an instance `s` of sequence type `S`:

Type	Synopsis
`S::value_type`	Values in the sequence
`S::iterator`	Iterator to `S::value_type`, satisfies `RandomAccessIterator`
`S::size_type`	Type to represent sequence size (length, number of values)

Expression	Returns	Synopsis
`s.begin()`	`S::iterator`	Iterator on first value
`s.end()`	`S::const_iterator`	Iterator past final value
`s.size()`	`S::size_type`	`s.end() - s.begin()`
`s[i]`	`const S::value_type &` or `S::value_type`	`*(s.begin() + i)`

As usual, all types have value semantics (assignment, copy, comparison, ...).

Example:

```
lazy_sequence<int> seq(10,
                       [](int i) {
                         return (100 + i * i);
                       });

// `seq` does not contain any elements at this point, but it satisfies the
// STL Sequence Container concept:

std::cout << "sequence size: " << seq.size() << '\n';

auto it = seq.begin();
// still nothing computed

int first = *it; // <-- returns value from generator(it._pos)

for (auto e : seq) {
  // generates values one by one:
  std::cout << e << " ";
}
std::cout << std::endl;
// prints:
// 100 101 104 109 116 125 136 149 164 181
```

Strictly speaking, there is no *Sequence* concept in C++ or the STL.
We will encounter the concept category *Range* for this soon.

4-X: Challenges

4-X-1: Bresenham's Line Sequence

Make yourself familiar with Bresenham's Line Algorithm and implement it as sequential concept:

```
raster_image img(width, height);

raster_point p_from = img[{x0, y0}];
raster_point p_to   = img[{x1, y1}];

raster_line_seq = raster_line_sequence(raster_image, p_from, p_to);

for (auto line_point : raster_line_seq) {
  img[line_point].color = black;
}
```

4-X-2: Skip List

Implement a class template `cpppc::map` that satisfies the `std::map` container interface based on a Skip List data structure.

References:

- Specification of std::map: http://en.cppreference.com/w/cpp/container/map
- *Skip Lists: A Probabilistic Alternative to Balanced Trees*. W. Pugh, 1990. http://cpppc.pub.lab.nm.ifi.lmu.de/docs/skiplist.pdf