CPPPC: Assignment 03

May 9, 2018

Marius Herget

Contents

3-0: Prerequisites **2**
- 3-0-1: Iterator Concepts 2
- 3-0-2: Sequence Container Concept 2

3-1: List Container Template **3**

3-2 : Measurements<T> Class Template **3**

3-X: Improve Efficiency **4**

3-0: Prerequisites

3-0-1: Iterator Concepts

- Make yourself familiar with the Iterator concepts in the STL:
  `http://en.cppreference.com/w/cpp/concept/Iterator`
- What are the differences between the concepts ForwardIterator and RandomAccessIterator?

  – ForwardIterator reads and writes from the first element to the last element. Hence it follows the order of the data.
  – RandomAccessIterator read and writes obviously on random access. Hence it access data in a non-sequentially (via offsets).
  – As shown in fig. 1 there are different iterator categories. Input is the most restricted one. A ForwardIterator is way more restricted than the RandomAccessIterator.
  – A RandomAccessIterator have similiar functionality as standard pointers.
  – The detailed properties can be seen in the *Iterator categories properties* table in [1].

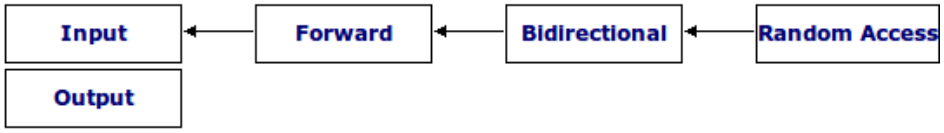

Figure 1: Iterator categories [1]

- What are the differences between the concepts InputIterator and OutputIterator?

  – Input- and OutputIterators are the most restricted Iterator categories. "They can perform sequential single-pass input or output operation" [1].
  – An Input operation is comparable with an `istream / read` operation it it since gets something from the memory and makes it available to your program.
  – An Output operation in comparision puts something from your application back out into the environment (`write`).

3-0-2: Sequence Container Concept

Sequence containers implement data structures which can be accessed sequentially. Methods `begin()` and `end()` define the iteration space of the container elements.

- Make yourself familiar with Sequence Container concept defined in the STL:
  `http://en.cppreference.com/w/cpp/concept/SequenceContainer`

Implement?!?

Excerpt:

Type	**Synopsis**
`typename value_type`	the container's element type T
`typename iterator`	iterator type referencing a container element
`typename const_iterator`	typically defined as const `iterator`
`typename reference`	type definition for `value_type &`
`typename const_reference`	type definition for `const value_type &`

Signature	Synopsis
`iterator begin()`	Iterator referencing the first element in the container or `end()` if container is empty
`const_iterator begin() const`	const iterator referencing the first element in the container or `end()` if container is empty
`iterator end()`	iterator referencing past the final element in the container
`const_iterator end() const`	const iterator referencing past the final element in the container
`size_type size() const`	number of elements in the container, same as `end() - begin()`

3-1: List Container Template

1. Complete the implementation of the List container template discussed in the last lab session.
   - Remember that all template code must be implemented in headers!
   - In your implementation, ensure that `List<T>` and `List<T>::iterator` satisfy `std::list<t>` and the STL iterator concepts, in particular iterator traits which are based on iterator tags.
2. Use the test suite of `Vector` you ported to C++ in assignment 2 to test your implementation of the `List` container.

- Implement `push_back`, `pop_back`
- `bidirectional_iterator`??? -> Nope ForwardIterator is enough.
- Implement as `doubly linked list`?? -> single linked is enough
- Whole Container concept:
  `begin(), end(), size(),`
  `not: max_size(), empty(), swap()`
- testsuite implementation
  - Which concepts to test?

3-2 : Measurements<T> Class Template

Assuming you run a series of benchmarks, each returning a measurement. At the end of the test series, the mean, median, standard deviation (sigma) and variance should be printed.
Implement the class template Measurements<T> representing a sequence container that allows to collect measurement data as single values and provides methods to obtain the mean, median, standard deviation and variance of the container elements.

- Is using the List container recommended? $->$ use `std:vector<>`
- Doppelindex? How?
- Calc `median`, `mean`, `varience` and `sigma` in runtime or at modification?

Measurements Container Concept In addition to the Sequence Container Concept:

Signature	Synopsis
`T median() const`	returns the median of the elements in the container or 0 if the container is empty
`double mean() const`	returns the mean of the elements in the container
`double variance() const`	returns the population variance of the elements in the container
`double sigma() const`	returns the standard deviation of the elements in the container

Example:

```
Measurements<int> m1;
m1.insert(10);
m1.insert(34);

m1.size(); // = 2

Measurements<double> m2;
std::vector<double> v({ 36, 37, 10 });
m2.insert(v.begin(), v.end());

m1.insert(m2.begin(), m2.end())

m1.size(); // = 5

int    median = m1.median();
double mean   = m1.mean();
double sdev   = m1.sigma();
double var    = m1.variance();
```

Define a class template `Measurements<T>` that satisfies the Sequence Container concept (`http://en.cppreference.com/w/cpp/concept/SequenceContainer`) and the Measurements Container concept defined above.
You may ignore the `emplace` methods for now.
The solution uses `std::vector` as a starting point, but you may use any underlying data structure in your implementation of `cpppc::Measurements<T>`.

3-X: Improve Efficiency

- Refactor your implementation of `Measurements<T>` such that all methods in the Measurements concept maintain constant computational complexity $O(c)$
- There are arithmetic solutions, possibly at the cost of numeric stability, and approaches focusing on the underlying data structure

References

[1] <iterator> - C++ Reference, Dec 2013. [Online; accessed 7. May 2018].