Object-Oriented Programming for Scientific Computing

STL Containers and Iterators

Ole Klein

Interdisciplinary Center for Scientific Computing
Heidelberg University
ole.klein@iwr.uni-heidelberg.de

9. Juni 2015

Evaluation

What is the purpose?

- Anonymous questionnaire from the university administration, used for quality assurance and collection of feedback.
- Chance for you to mention things that were better than in other lectures and should be kept / suggested to others / might be of interest.
- Chance for you to mention things that didn't work out as planned or might be organized in a better way, and give constructive suggestions to make the lecture better.
- Questions as inspiration for the free form fields:
  - Why did you decide to attend this course?
  - Did the content up to now match what you expected?
  - Or did the lectures surprise you (in a good or in a bad way)?
  - How did the unexpected number of participants and the resulting rather spontaneous restructuring work out for you?

Evaluation

Rules for your feedback:

- Time for completing the questionnaire: 15 minutes (at least).
- Use a blue or black ball-point pen if possible, since the sheets will be scanned automatically.
- Please only write into boxes, and use uppercase letters, since otherwise your comments may be lost.
- Common tutorial: *not specified*, since we don't have one.
- I need a student that collects the sheets for me and accompanies me when I hand the envelope in, since I'm not allowed to see the questionnaires after they have been filled out.

Points and Exam

- I have entered your points for the first five exercise sheets in MUESLI. Please check the entries and make sure they are right, since these are the only numbers relevant for admission to the exam.
- If you are missing points, please talk to your tutor to correct your records. Be prepared that you will have to provide the commit that bears your name, the mail where you are mentioned or some other argumentation why you should have a claim.
- As a reminder, the exam will take place on 21.7. during lecture time. Everybody who has the necessary points automatically takes part in the exam unless he or she states otherwise at least a day before.
- If necessary there will be a second exam for those that failed the first or have a good excuse why they couldn't take part ("Attest"). I don't have a date yet.

Set/Multiset

- The containers `set` and `multiset` are sorted sets of elements.
- While in a `set` every element may only appear once, a `multiset` may contain elements several times.
- In a set, it is particularly important to be able to quickly determine whether an element is in the set or not (and in a multiset, how often).
- The search for an element is of optimal complexity $O(\log(N))$.
- `set` and `multiset` have three template parameters: the type of objects, a comparison operator and an allocator. For the last two, there are default values (`less` and the standard allocator).

Map/Multimap

- The containers `map` and `multimap` consist of sorted pairs of two variables, a key and a value. The entries in the map are sorted by the key.
- While each key can only appear once in a `map`, it may exist several times in a `multimap` (independent of the associated value).
- A `map` can be quickly searched for a key and then gives access to the appropriate value.
- The search for a key is of optimal complexity $O(\log(N))$.
- `map` and `multimap` have four template parameters: the type of the keys, the type of the values, a comparison operator and an allocator. For the last two, there are again default values (`less` and `new`/`delete`).

Container Concepts

- The properties of STL containers are divided into specific categories.
- They are, for example, `Assignable`, `EqualityComparable`, `Comparable`, `DefaultConstructible`...
- The objects of a class that are to be stored in a container must be `Assignable` (there is an assignment operator), `Copyable` (there is a copy constructor) , `Destroyable` (there is a public destructor), `EqualityComparable` (there is an `operator==`) and `Comparable` (there is an `operator<`).

Container

- A Container itself is Assignable (there is an assignment operator), EqualityComparable (there is an operator==) and Comparable (there is an operator<).
- Associated types:

value_type	The type of object stored. Needs to be Assignable, but not DefaultConstructible.
iterator	The type of the iterator. Must be an InputIterator and a conversion to const_iterator must exist.
const_iterator	An iterator through which the elements may be read but not changed.
reference	The type of a reference to the value_type of the container.
const_reference	As above, but constant reference.
pointer	As above, but pointer.
const_pointer	As above, but pointer to constant.
difference_type	A type suitable for storage of the difference between two iterators.
size_type	An unsigned integer type that can store the distance between two elements.

Container

In addition to the methods of `Assignable`, `EqualityComparable` and `Comparable`, a container alway has the following methods:

`begin()`	Returns an iterator to the first element. If the container is `const` this is a `const_iterator`.
`end()`	As `begin()`, but points to the location after the last element.
`size()`	Returns the size of the container, i.e. the number of elements, return type `size_type`.
`max_size()`	Returns the maximum size allowed at the moment, return type `size_type`.
`empty()`	True if the container is empty.
`swap(b)`	Swaps elements with container `b`.

Specializations of the Container Concept

Assignable
EqualityComparable
Comparable
Container
ForwardContainer
ReversibleContainer
RandomAccessContainer

Abbildung: `Container` concepts

ForwardContainer

- Specialization of the `Container` concept.
- There is an `iterator` with which one can pass through the container in the forward direction (`ForwardIterator`).

ReversibleContainer

- There is an iterator which allows passing back and forth through the container (`BidirectionalIterator`).
- Additional associated types:

`reverse_iterator`	Iterator in which the `operator++` moves to the previous item instead of the next item.
`const_reverse_iterator`	As above, but `const` version.

- Additional methods:

`rbegin()`	Returns an iterator to the first element of a reverse pass (last element of the container).
`rend()`	As `rbegin()`, but points to the location before the first element.

Implementations

- `std::list`
- `std::set`
- `std::map`

RandomAccessContainer

- Is a specialization of `ReversibleContainer`.
- There is an iterator with which one can gain access to arbitrary elements of the container (`RandomAccessIterator`, uses an index).
- Additional methods: `operator[](size_type)` (and `const` version), access operators for random access.

Implementations

- `std::vector`
- `std::deque`

Sequence

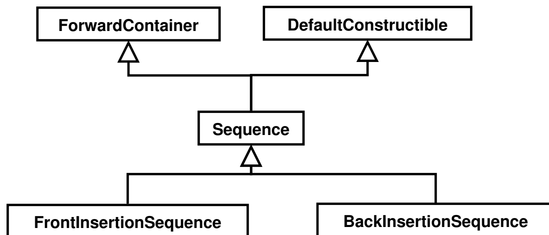

Abbildung: Sequence concepts

Sequence Methods

A `Sequence` is a specialization of the concept of `ForwardContainer` (so one can at least in one direction iterate over the container) and is `DefaultConstructible` (there is a constructor without argument / an empty container).

`X(n,t)`	Generates a sequence with $n \geq 0$ elements initialized with `t`.
`X(n)`	As above, but initialized with the default constructor.
`X(i,j)`	Generates a sequence which is a copy of the range `[i,j)` . Here `i` and `j` are `InputIterators`.
`insert(p, t)`	Inserts the element `t` in front of the one the iterator `p` points to, and returns an iterator that points to the inserted element.
`insert(p, i, j)`	As above, but for the range [i,j).
`insert(p, n, t)`	As above, but inserts `n` copies of `t` and returns an iterator to the last of them.
`erase(p)`	Invokes the destructor for the element to which the iterator `p` points and deletes it from the container.
`erase(p,q)`	As above, but for the range [p,q).
`erase()`	Deletes all elements.
`resize(n,t)`	Shrinks or enlarges the container to size `n` and initializes new elements with `t`.
`resize(n)`	The same as `resize(n, T())`.

Complexity Guarantees for Sequences

- The constructors `X(n,t)` `X(n)` and `X(i,j)` have linear complexity.
- Inserting elements `insert(p, t)`, `insert(p, i, j)` and deleting them with `erase(p,q)` has linear complexity.
- The complexity of inserting and removing single elements depends on the sequence implementation.

BackInsertionSequence

Methods in addition to those from the Sequence concept:

back()	Returns a reference to the last element.
push_back(t)	Inserts a copy of t after the last element.
pop_back()	Deletes the last element of the sequence.

Complexity Guarantees

back, push_back, and pop_back have amortized constant complexity, i.e. in individual cases it may take longer but the average time is independent of the number of elements.

Implementations

- std::vector
- std::list
- std::deque

FrontInsertionSequence

Methods in addition to those from the Sequence concept:

front()	Returns a reference to the first element.
push_front(t)	Inserts a copy of t before the first element.
pop_front()	Removes the first element of the sequence.

Complexity Guarantees

front(), push_front(), and pop_front() have amortized constant complexity.

Implementations

- std::list
- std::deque

STL Sequence Containers

BackInsertionSequence
RandomAccessContainer
T:typename
Alloc:typename
std::vector

FrontInsertionSequence
ReversibleContainer
BackInsertionSequence
T:typename
Alloc:typename
std::list

FrontInsertionSequence
RandomAccessContainer
BackInsertionSequence
T:typename
Alloc:typename
std::deque

Abbildung: STL sequence containers

Associative Containers

DefaultConstructible
ForwardContainer
ReversibleContainer
AssociativeContainer
SimpleAssociativeContainer
MultipleAssociativeContainer
UniqueAssociativeContainer
SortedAssociativeContainer
PairAssociativeContainer
MultipleSortedAssociativeContainer
UniqueSortedAssociativeContainer

Abbildung: Associative container concepts

AssociativeContainer

- Is a spezialisation of `ForwardContainer` and `DefaultConstructible`.
- Additional associated type: `key_type` is the type of a key.
- Additional Methods:

`erase(k)`	Deletes all entries with the key `k`.
`erase(p)`	Deletes the element that the iterator `p` points to.
`erase(p,q)`	As above, but for the range [`p`,`q`).
`clear()`	Deletes all elements.
`find(k)`	Returns an iterator pointing at the item (or one of the items) with the key `k` or `end()` if the key does not exist.
`count(k)`	Returns the number of elements with the key `k`.
`equal_range(k)`	Returns a `pair` `p` of iterators so that [`p.first`,`p.second`) consists of all elements have the key `k`.

AssociativeContainer

- Assurances:

  Continuous memory: all elements with the same key directly follow one another.

  Immutability of the key: The key of each element of an associative container is unchangeable.

Complexity Guarantees

`erase(k)`	Average complexity at most $O(\log(size() + count(k))$
`erase(p)`	Average complexity constant
`erase(p,q)`	Average complexity at most $O(\log(size() + N))$
`count(k)`	Average complexity at most $O(\log(size() + count(k))$
`find(k)`	Average complexity at most logarithmic
`equal_range(k)`	Average complexity at most logarithmic

These are just average complexities, and the worst case can be significantly more expensive!

SimpleAssociativeContainer and PairAssociativeContainer

These are specializations of the AssociativeContainer.

SimpleAssociativeContainer

has the following restrictions:

- key_type and value_type must be the same.
- iterator and const_iterator must have the same type.

PairAssociativeContainer

- introduces the associated data type mapped_type. The container maps key_type to mapped_type.
- The value_type is std::pair<key_type,mapped_type>.

SortedAssociativeContainer

This specialization uses a sorting criterion for the key. Two keys are equivalent if none is smaller than the other.

Additional associated types

`key_compare`	The type implementing `StrictWeakOrdering` to compare two keys.
`value_compare`	The type implementing `StrictWeakOrdering` to compare two values. Compares two objects of type `value_type` by handing their keys over to `key_compare`.

Additional Methods

`key_compare()`	Returns the key comparison object.
`value_compare()`	Returns the value comparison object.
`lower_bound(k)`	Returns an iterator to the first element whose key is not less than `k`, or `end()` if there is no such element.
`upper_bound(k)`	Returns an iterator to the first element whose key is greater than `k`, or `end()` if there is no such element.

SortedAssociativeContainer

Complexity Guarantees

- `key_comp`, `value_comp` and `erase(p)` have constant complexity.
- `erase(k)` is $O(\log(size()) + count(k))$
- `erase(p,q)` is $O(\log(size() + N))$
- `find` is logarithmic.
- `count(k)` is $O(\log(size() + count(k))$
- `lower_bound`, `upper_bound`, and `equal_range` are logarithmic.

Assurances

value_compare: if `t1` and `t2` have the associated keys `k1` and `k2`, then `value_compare()(t1,t2)==key_compare(k1,k2)` is guaranteed to be `true`.

Increasing order of the elements is guaranteed.

UniqueAssociativeContainer and MultipleAssociativeContainer

A UniqueAssociativeContainer is an AssociativeContainer with the additional property that each key occurs at most once.
A MultipleAssociativeContainer is an AssociativeContainer in which each key can appear several times.

Additional Methods

`X(i,j)`	Creates an associative container from the items in the range [i,j).
`insert(t)`	Inserts the value_type t and returns a std:pair containing an iterator to the copy of t and a bool (true if the copy has just been inserted)
`insert(i,j)`	Inserts all elements in the range [i,j) .

UniqueAssociativeContainer and MultipleAssociativeContainer

Complexity Guarantees

- The average complexity of `insert(t)` is at most logarithmic.
- The average complexity of `insert(i,j)` is at most $O(N * \log(size()) + N)$, where `N=j-i`

Associative Container Classes

SimpleAssociativeContainer
UniqueSortedAssociativeContainer
Key:typename
Compare:typename
Alloc:typename
std::set

PairAssociativeContainer
UniqueSortedAssociativeContainer
Key:typename
HashFcn:typename
EqualKey:typename
Alloc:typename
std::map

SimpleAssociativeContainer
MultipleSortedAssociativeContainer
Key:typename
Compare:typename
Alloc:typename
std::multiset

PairAssociativeContainer
MultipleSortedAssociativeContainer
Key:typename
Mapped:typename
Compare:typename
Alloc:typename
std::multimap

Abbildung: Associative container classes

Properties of the Different Container Classes

	`vector`	`deque`	`list`	`set`	`map`
Typical internal data structure	Dynamic array	Array of arrays	Doubly linked list	Binary tree	Binary tree
Elements	values	values	values	values	keys/values
Search	slow	slow	very slow	fast	fast (key)
Insert/delete fast	end	beginning and end	everywhere	—	—
Frees memory of removed elements	never	sometimes	always	always	always
Allows preallocation	yes	no	—	—	—

Tabelle: Properties of the different container classes

Which Container Should Be Used?

- If there is no reason to use a specific container, then `vector` should be used, because it is the simplest data structure and allows random access.
- If elements often have to be inserted/removed at the beginning or at the end, then a `deque` should be used. This container will shrink again when items are removed.
- If elements have to be inserted/removed/moved at arbitrary locations, then a `list` is the container of choice. Even moving all elements from one `list` into another can be done in constant time. But there is no random access.
- If it should be possible to repeatedly search for items in a fast way, one should use a `set` or `multiset`.
- If it is necessary to manage pairs of keys and values (as in a dictionary or phone book) then one uses a `map` or `multimap`.