C++ STL – Containers

(Day 1, Part 2)

Commonalities within Categories

- Category: Sequence Containers
  - Sequential Access (only)
  - Index-Based Random Access

- Category: Associative Containers
  - Sub-Category: Ordered Associative Containers
  - Sub-Category: Unordered Associative Containers

- Combining Container Classes

Further Comparisons of Associative Containers

- Sets versus Maps
- Commonalities of Sets
- Commonalities of Maps

- Unique Values / Keys
- Non-Unique Values / Keys
- Unique / Non-Unique Keys

Category: Sequence Containers

All sequence containers allow sequential access "front" to "back".

Since C++11 this is most easily programmed with a range-`for` loop:

```
container c;
…                              // with e holding a
for (auto e : c) …             // ... copy of ...
for (const auto &e : c) …      // ... read-only reference to ...
for (auto &e : c) …            // ... modifiable reference to ...
                               // ... another element for each run
```

Sequential Access Only

The sequence containers `std::list` and `std::forward_list` provide sequential access only, with

- *bidirectional iterators* for `std::list`, and
- *forward iterators* for `std::forward_list`.

Therefore their iterators are not compatible with a number of algorithms requiring random access for an efficient operation, like sorting, heap operations etc.

For the common operation of sorting both have their own `sort` member function.

So it is at least possible to use them with algorithms expecting sorted ranges.

[!] Nevertheless they may exhibit much worse performance in these cases – like for a binary search.

Index-Based Random Access

The sequence containers `std::array`, `std::vector`, and `std::deque` provide random access with a numeric index by using

- either the overloaded `operator[]`
- or the member function `at`

with an integral value as argument. Only the latter is

- **required** to check the index at run time and
- will throw an `std::out_of_range` exception

if it does not denote a valid entry.*

> For any index-checking implemented in the program itself, be sure to understand the consequences of the fact that the containers are typically indexed with `unsigned` integral types.

*: Of course – as it was already the case in C – any index-based random access uses 0-origin, i.e. for some `container c` the valid range is `0` ... `c.size()-1`.

Out of Range Indices

For some container supporting index-based access ...

```
container c;                // for container assume any of
                            // std::array, std::deque, std::vector
container::size_type idx;   // defined by all STL containers
```

... indexing may be without ...

```
… c[idx] …  // may expose UB
```

... or with mandatory range check.

```
… c.at(idx) … // may throw
```

Preferring `operator[]` is only acceptable in performance critical code.*

[!] Accessing container elements outside the bounds of an STL container may generally cause **Undefined Behaviour (UB)** – for the consequences see:
http://en.cppreference.com/w/cpp/language/ub

*: Special attention should be paid that the behavior of `operator[]` for sequence containers is not confused with the behavior of the same operation for maps, which *always* returns a reference to an existing (that may have been created as part of the operation).

Unsigned Indexing Expressions

Frequently variables used for indexing will have an unsigned type*

> [!] Evaluating expressions with unsigned operands – more often than not – may surprise the unenlightened ...*

The following code is not as robust as the tests make believe ... it all depends on the types of `offset` and `delta`:

```
container c;
…
assert(offset >= 0 && delta >= 0);
// avoid 'offset' is already beyond container end ...
if (offset >= c.size()) … // ==> leave
// ... and going downwards 'delta' from 'offset' is a valid index
if (offset - delta < 0) … // ==> leave
… c[offset - delta] … // looks OK ... but may cause UB !!!!!!!!!
```

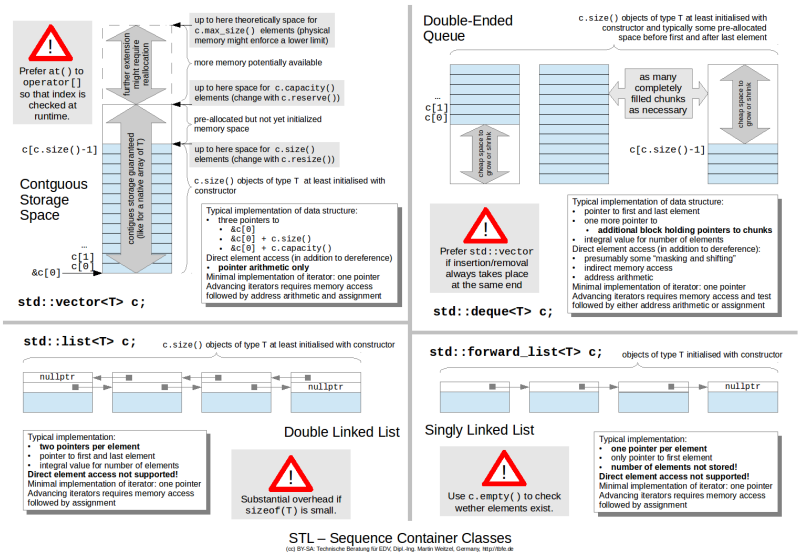
*: ... with the least "surprise" should be they can never be less than zero! Looking back, it may have been a bad decision to require `std::size_t` is an unsigned type in C89, paving the road for all the nested `size_type` definitions in the STL containers. It is only defendable with the argument that in times of "tiny" amounts of main memory, a `char`-array occupying more than half of "the whole range 64KB" – a memory size not unusual around 1989 – with 16 bit `int`-s required an unsigned type.

Sequence Container Data Structures

The data structure of the STL sequence container classes are visualised in the adjoining Info-Graphic.*

Not depicted in the Graphic is `std::array<T, N>`.

It has **exactly the same** memory layout as a native array with `N` elements of type `T`, including any padding (and therefore basically the same as shown for the core content of `std::vector<T>`, though space is not allocated on the heap).



*: Note that this visualisation is only **indirectly derived** from the STL specification the performance guarantees and operations specified for the various container classes.

Category: Associative Containers

Containers of this category are either Sets or Maps and further divided into the sub-categories* of

- Ordered Associative Containers, or
- Unordered Associative Containers.

In both cases there are variants that

- require a Unique Value or Key, and hence reject insertion of another entry comparing equal to an existing one, or
- allow Multiple Values or Keys to coexist, which compare equal to each other.

*: Be aware of the slight difference in comparison for equality when switching between the version with tree-based and hash-based storage.

Combining Container Classes

Compared to other container libraries the STL provides no container classes with advanced or fancy features, like fast lookup and sequential traversal in the order of insertions.

> Nevertheless the STL containers provide a solid foundation on which containers
>
> with special features could be built.

If there is – like the *LinkedHashMap* from Java would satisfy –

- a frequent need for fast (O(1)) look up of elements by their value, and
- a (rarer) need for sequential traversal in insertion order

this could be achieved by combining `std::unordered_set` with `std::vector` or `std::list` (depending on the expected size and whether frequent removals must also be supported with reasonable efficiency).*

*: It can be viewed as an advantage of the STL to concentrate on the basics, as combinations are not pre-build but can be assembled in an optimal way, though not quite without any effort.

Sets versus Maps

Sets and maps have in common that entries may be looked-up efficiently, with

- $O(\log_2 N)$ performance for the hash-based and
- $O(1)$ performance for tree-based variants.

The difference is with respect to entries, which is a

- *single value* (either native type or class instance) for a set,
- *key and value*[*] (both either native type or class instance) for a map.

[*]: Technically an `std::pair` with a `first` component of the key type and a `second` component of the value type.

Commonalities of Set-s

A set consists of entries of either a native type or a class instance.

- New elements are usually added with the `insert` or `emplace` member functions,* accepting as argument
  - an instance of the element type (resp. class) to be inserted,
  - or (any number of) constructor parameters to create one.
- The return value is an `std::pair` of an iterator and a boolean (where and if the new element was inserted – for more information how this may be used see here.

> Inserting new values with `emplace` is typically more performant as it avoids to copy the element.

*: In case of `insert` there is a second optional argument to give after which position the element might be inserted, in case of `emplace` this hint can be given by using `emplace_hint` instead.

Commonalities of Map-s

A map consists of entries which are key-value pairs, with each part either a native type or a class instance.

- New elements are usually added* with the
  - the `insert` member function, accepting as argument an `std:pair` of the key's and the value's type (resp. class) to be inserted, or
  - the `emplace` member function, accepting as argument an `std:pair` as `insert` does, or
  - the `emplace` member function with exactly two arguments, used to construct the key-value pair in place, or
  - an `emplace` member function using `std::piecewise_construct` and `std::forward_as_tuple` to create the key-value pair in place.
- The value type of a map entry can not be `void` – use a set in this case!

> Inserting new key-value pairs with `emplace` is typically more performant as it avoids to copy the element.

*: In case of `insert` there is a second optional argument to give after which position the element might be inserted, in case of `emplace` this hint can be given by using `emplace_hint` instead.

Sub-Category: Ordered Associative Containers

Containers of this category use tree-based storage, hence:

- They need an ordering relation for their elements or keys,
  - i.e. `operator<` for the element or key type,* or
  - a user supplied function returning `true` if its first argument should go first in sort order.
- The ordering relation may impose any criteria, as long as it
  - is not self-contradicting, i.e. from `a < b` follows `!(b <= a)`, and
  - applies transitively, i.e. with `a < b` and `b < c` it follows that `a < c`.

On sequential traversal the elements are visited in ascending sort order, according to the ordering relation.

*: If necessary, comparing for equality can be done as the `!(a < b || b < a)`.

Sub-Category: Unordered Associative Containers

Containers of this category use hash-based storage, hence need

- a hash-function for the type of their elements (set) or keys (map)
- to compare elements or keys for equality
  - with `operator==` for the element or key type, or
  - a user supplied function returning `true` if both of its arguments compare equal.

On sequential traversal the elements are visited in no particular order, except that elements with the same value or key are visited without any other elements interspersed.*

(i) For more information on the category of unordered associative containers see subsection Unordered Associative Container in http://en.cppreference.com/w/cpp/container

*: This is an important guarantee as it allows group values or keys comparing equal during sequential traversal.

Commonalities of Associative "Non-`multi`"-Containers

The non-`multi`-variants restrict the contained value (set) or key (map) to a single entry with no other entries comparing equal.

- The return value of `insert` or `emplace` (including variants) holds two pieces of information:
  - member `first` is an iterator set to the position of the newly inserted or already existing element;
  - member `second` is a `bool` which is `true` if a new entry was added, `false`if this was not the case because one already existed.
- To look-up an element with a given value or key, usually the member function `find` is used, which returns*
  - an iterator to the element if it exists,
  - or the end iterator of the container.

*: Alternatively the member function `count` could be used, which – for non-`multi` containers – will either return `0` or `1`.

Commonalities of Associative "`multi`"-Containers

The `multi`-variants allow any number of entries with the same value (set) or key (map) comparing equal to each other.

- The return value of the `insert` or `emplace` member functions is an iterator, denoting the position where the new element was inserted.
- Looking up elements is usually done with the
  - `find` member function to search for the first element with the given value or key (from which may be iterated further on), or
  - `equal_range` member function to search for the first and last element with the given value or key.
- Alternatively the member function `count` could be used, which returns the number of elements with the given value or key.

Using `count` if the question is whether a given value or key exists at least once or not at all causes unnecessary effort.

Maps with Unique / Non-Unique Keys

For maps there is another difference between the non-`multi` and the `multi` variant:

- Accessing an entry by using the overloaded `operator[]` with an argument of the key type (or convertible to that)
  - returns the value part of the entry for this key by reference ...
  - ... creating it (as part of the operation) if it does not exist, with
    - zero-initialisation for built-in types, or
    - the default constructor for instances of a class.
- Accessing an entry by using the overloaded `at()` member function with an argument of the key type (or convertible to that) will
  - either return the value part of the entry for this key by reference,
  - or throw an `std::out_of_range` exception.
- C++1y will add another function `insert_or_assign` which allows to either insert a new entry or change the value part of an existing one.

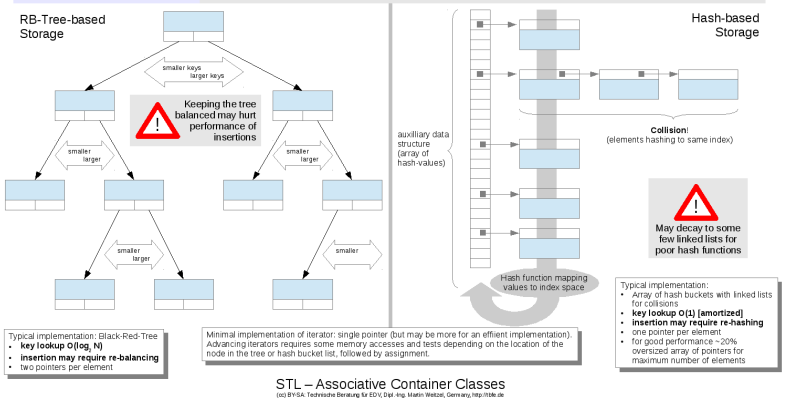
Associative Container Data Structures

The data structure of the STL associative container classes are visualised in the adjoining Info-Graphic.*

For the `unordered_`-variants the implementation shown is effectively demanded, by the lower level API that specifies how to access the linked lists stored per hash bucket.

A alternative strategy that stores over-runners in adjacent free entries would hence be hard to implement, though it would surely have **much** better cache hit rates.

Contained elements	*STL Class Name*		*Restrictions*
objects of type *T*	`std::set`	`std::unordered_set`	unique elements guaranteed
	`std::multiset`	`std::unordered_multiset`	multiple elements possible (comparing equal to each other)
pairs of objects of type *T1* (key) and type *T2* (associated value)	`std::map`	`std::unordered_map`	unique keys guaranteed
	`std::multimap`	`std::unordered_multimap`	multiple keys possible (comparing equal to each other)



STL – Associative Container Classes

*: Note that this visualisation is only **indirectly derived** from the STL specification the performance guarantees and operations specified for the various container classes.

STL Containers and Adaptors in Detail

Sequence Containers:

- Class `std::array`
- Class `std::vector`
- Class `std::deque`
- Class `std::forward_list`
- Class `std::list`

Container Adaptors:

- Class `std::stack`
- Class `std::queue`
- Class `std::priority_queue`

Ordered Associative Containers:

- Class `std::set`
- Class `std::multiset`
- Class `std::array`
- Class `std::unordered_set`

Unordered Associative Containers:

- Class `std::map`
- Class `std::multimap`
- Class `std::array`
- Class `std::unordered_map`

Class `std::array`

Provides *contiguous storage*

- of compile-time fixed size (so no heap use),
- with efficient index-based random access,
- and the same memory layout as a native array.

See the pages following for

- characterisation,
- special considerations, and
- some example code fragments.

For more information on class `std::array` see
http://en.cppreference.com/w/cpp/container/array

`std::array` Characterisation

Property	
Introduced with	C++11
Header file	`<array>`
Contiguous storage	yes
Direct element lookup by ...	numeric index
... with performance	O(1)
Efficient insertion / removal	not applicable
Iterator category	random access
Dynamic growth ...	no
... may invalidate iterators	not applicable

As memory layout is guaranteed to be the same, instances may also be used with C and (legacy) C++ APIs, expecting native (C-style) arrays.

- Member function `data()` returns a pointer to the first element* and
- member function `size()` returns the number of elements.

---

*: For some `std::array<T, N> c` a difference between `c.data()` and `&c[0]` might exist if `T` overloaded `operator&`.

`std::array` Special Considerations:

- The element type must have a default constructor (either supplied or generated by the compiler).
- There is an assignment operator that
  - either copies all elements in a loop (for lvalue references)
  - or moves all elements in a loop (for rvalue references if the elements have a `nothrow` move assignment.
- In an `std::move`-operation each element will be moved (if elements make a difference between move- and copy-assignment).
- There are comparison operators comparing all elements in a loop*
- For storing 0/1-values space-efficiently rather consider `std::bitset`.

Instances of `std::array<T>` **will not decay** to a pointer (`T*`) to their first element, as it is the case for native (C-style) arrays.

*: In other words: comparisons are simply forwarded to elements, though elements need not implement comparison operations as long as no comparison is requested from the whole array.

std::array Code Examples and Fragments

Code that is not extremely performant sensitive should prefer range-checked indexing:

```
// maybe undefined behaviour
std::array< …  , N> c;
…         // when i out of
… c[i] … // range 0..(N-1)
```

```
// will throw exception
std::array< …  , N> c;
…             // when i out of
… c.at(i) …  // range 0..(N-1)
```

Be aware of the **does not decay to pointer** property when handing over in lambda capture lists, especially in case of C++14 init captures:

```
// assume one of the following:
… c[N]; // native or
std::array< … , N> c;
…                       // handed over by: native  vs. std::array
… [c]( … ) { … c … } …                  //  copy        copy
… [&c]( … ) { … c … } …                 //  reference   reference
… [mine = c]( … ) { … } …               //  pointer     copy!!
… [mine = &c[0]]( … ) { … } …           //  pointer     pointer
```

Class `std::vector`

Provides *contiguous storage*

- that may grow (or shrink) at run-time at one end,
- with efficient index-based random access,
- and the same memory layout as a native array.

See the pages following for

- characterisation,
- special considerations, and
- some example code fragments.

For more information on class `std::vector` see
http://en.cppreference.com/w/cpp/container/vector

`std::vector` Characterisation

Property	
Header file	`<vector>`
Introduced with	C++98
Contiguous storage	yes
Direct element lookup by ...	numeric index
... with performance	O(1)
Efficient insertion / removal	back (end) only
Iterator category	random access
Dynamic growth ...	yes
... may invalidate iterators	yes

As memory layout is guaranteed to be the same, instances may also be used with C and (heritage) C++ APIs, expecting native (C-style) arrays.

- Member function `data()` returns a pointer to the first element* and
- member function `size()` returns the number of elements.

---

*: For some `std::vector<T> c` a difference between `c.data()` and `&c[0] might exist if`Toverloadedoperator&`.

`std::vector` Special Considerations:

- Space is usually allocated when necessary,
  - **with some room to grow**
  - **to avoid frequent copying of the elements**.
- Pre-allocated space is usually chosen by the implementation as a multiple of the space already used.*
- An implementation may take a call to member function `shrink_to_fit` as a hint only and **not actually free any excess space**, or free it at a later time.
- The `insert()` member function will have bad performance, as many elements may have to be moved – even if sufficient pre-allocated space is available, though …
  - for simple types `std::memcpy` will often be used internally, instead of copying each element in a loop;
  - elements may be moved instead of not copied, if they have a `nothrow` move-constructor.

---

*: Typically, if a vector with N elements needs to grow, its memory space will be enlarged to about $2 \times N$.

Replacing `push_back` with `emplace_back` (available since C++11) may improve performance, as it avoids one copy constructor:*

```
std::vector<T> c;
…
… c.push_back(T{ … , … , … }) …
```

```
std::vector<T> c;
…
… c.emplace_back( … , … , … ) …
```

If the size to which an `std::vector` will grow is known **before a large set of values is appended**, it is always recommendable …

to reserve the required space in advance …

```
std::vector<T> c;     // might
c.reserve(1'000'000);  // need
for( … ) // 1 million elements
     c.emplace_back( … );
… c.shrink_to_fit();
```

… or at least outside a time-critical section.

```
// will need N more elements
auto needed = c.size() + N;
if (c.capacity() < needed)
    c.reserve(needed);
… // time critical part
```

*: Note that perfect forwarding will still apply to constructor arguments.

Class `std::deque`

Provides *double ended queue* storage

- that may grow (or shrink) at run-time at both ends,
- with reasonable efficient index-based random access.

See the pages following for

- characterisation,
- special considerations, and
- some example code fragments.

For more information on class `std::deque` see
http://en.cppreference.com/w/cpp/container/deque

`std::deque` Characterisation

Property	
Header file	`<deque>`
Introduced with	C++98
Contiguous storage	partially
Direct element lookup by ...	numeric index
... with performance	O(1)
Efficient insertion / removal	back (end) and front (begin)
Iterator category	random access
Dynamic growth ...	yes
... may invalidate iterators	yes

Though memory layout comes in contiguous chunks and might have reasonable cache performance on sequential traversal, but the details are unspecified and hence even single chunks can not be (portably) used with C and (legacy) C++ APIs expecting native (C-style) arrays.

`std::deque` Special Considerations:

- If insertions and removals are not required *at both ends*, but only random access by index, `std::vector` may be a more reasonable choice.
- If *random access by index* is not required, but only insertions and removals at both ends, `std::deque` competes with `std::list`:
  - the former has more basic overhead but much less per element;
  - the latter has minimal basic overhead but more per element.
- If insertions and removals in the middle are frequent, `std::list` may be a more reasonable choice.

std::deque Code Examples and Fragments

The following example uses an `std::deque` to implement a ring-buffer with a given maximum capacity:

```
template<class ElementType, size_t MaxSize>
class RingBuffer : private std::deque<ElementType> {
    using BaseContainer = std::deque<ElementType>;
public:
    using BaseContainer::size;
    void clear() {
        BaseContainer::clear(); shrink_to_fit();
    }
    using BaseContainer::empty;
    bool full() const { return size() == MaxSize; }
    bool put(const ElementType &e) {
        if (full()) return false;
        push_front(e);
        return true;
    }
    bool get(ElementType &e) {
        if (empty()) return false;
        e = back(); pop_back();
        return true;
    }
};
```

Class `std::forward_list`

Provides a *singly linked list*

- that may grow or shrink at run-time,
- with efficient insertions and removals at its front,
- and also in the middle, once a position is determined.*

For more information on class `std::forward_list` see:
http://en.cppreference.com/w/cpp/container/list

*: I.e. there must be an iterator set to it and the efficiency how that was obtained is not considered here.

`std::forward_list` Characterisation

Property	
Header file	`<forward_list>`
Introduced with	C++11
Contiguous storage	no
Direct element lookup by ...	no
... with performance	not applicable*
Efficient insertion / removal	front (begin)
Iterator category	unidirectional
Dynamic growth ...	yes
... may invalidate iterators	no

Note that insertions and removals are also efficient **in the middle** if the necessity to locate the affected element(s) is not taken into account.

[!] Locating elements in general has O(N) performance and very bad cache performance too.

*: Accessing the *n*-th element (from front) requires incrementing an iterator *n* times, i.e. random element access by a numeric index would have O(N) performance and is not directly supported.

`std::forward_list` Special Considerations:

- There are often better choices than `std::list`, *especially if linear traversal is a frequent usage pattern*.
  - E.g. even if no index based random access is necessary, an `std::vector` may be the better choice,
  - **except** if excess storage should be minimal and free space must be returned reliably as early as possible.
- Forward lists have a different semantics for insertions and removals based on an iterator position:
  - there are no member functions `insert`, `emplace`, `erase`, and `splice` (affecting the element at or directly prior to the iterator position),
  - **instead** there are member functions `insert_after`, `emplace_after`, `erase_after`, and `splice_after`, affecting the element directly succeeding the iterator position.

std::forward_list Code Examples and Fragments

To allow the smallest possible overhead, an `std::forward_list` does **not** hold an element count, therefore the code below left will not compile …

```
std::forward_list<T> c;
…
if (c.size() == 0) …
if (c.size() > 0) …
```

… but it may be easily replaced with this:

```
if (c.empty() == 0) …
if (c.size() > 0) …
```

To sort an `std::forward_list` using the *less-than* (default) or *greater-than* comparisons of `T`, use one of the following:

```
// ascending sort (default)
c.sort();
```

```
// descending sort
c.sort(std::greater<T>());
```

Note that the above works for `std::list` in the same way and vice versa, the sorting examples given there also work for `std::forward_list`.

Class `std::list`

Provides a *doubly linked list*,

- that may grow or shrink at run-time,
- with efficient insertions and removals at both ends,
- and even in the middle, once a position is determined.*

See the pages following for

- characterisation,
- special considerations, and
- some example code fragments.

*: I.e. there must be an iterator set to it and the efficiency how that was obtained is not considered here.

`std::list` Characterisation

Property	
Header file	`<list>`
Introduced with	C++98
Contiguous storage	no
Direct element lookup by ...	no
... with performance	not applicable*
Efficient insertion / removal	front (begin) and back (end)
Iterator category	bidirectional
Dynamic growth ...	yes
... may invalidate iterators	no

Note that insertions and removals are also efficient **in the middle**, if the necessity to locate the affected element(s) is not taken into account.

[!] Locating elements in general has O(N) performance and very bad cache performance too.

*: Accessing the *n*-th element (from front or back) requires incrementing an iterator *n* times, i.e. random element access by a numeric index would have O(N) performance and is not directly supported.

`std::list` Special Considerations:

- There are often better choices than `std::list`, *especially if linear traversal is a frequent usage pattern*.
- If insertions and removals are only necessary at the *same* end **and** in the middle, `std::forward_list` may be considered as an alternative with lesser overhead per element.
- If insertions and removals are only necessary at the *same* end,
  - an `std::vector` will most often be the better choice,
  - **except** if excess storage should be minimal and free space must be returned reliably as early as possible.

`std::list` Code Examples and Fragments (1)

If – for any reason – there is the necessity to access the *n*-th list element, `std::next` and `std::prev` help to avoid writing explicit loops:*

```
template<typename T>
T& get_nth(T& std::list<T> li,
           std::list<T>::size_type n,
           bool from_back = false) {
    if (n > li.size())
        throw std::out_of_range("offset too large);
    return from_back ? std::next(li.begin(), n)
                     : std::prev(li.end(), n+1);
    // or: std::next(from_back ? li.rbegin() : li.begin(), n);
}
```

An index may refer to the front …

```
… get_nth(c, 0) … // first
… get_nth(c, 1) … // second
…
```

… or the back of an `std::list c`:

```
… get_nth(c, 0, true) … // last
… get_nth(c, 1, true) … // etc.
…
```

*: The second argument allows to specify that the offset given should count from the "back" (end).

An alternative implementation could use a negative argument value to specify counting from "back" (end):*

```
template<typename T>
T& get_nth(T& std::list<T> li,
           std::list<T>::difference_type n) {
    if (n > li.size()
        throw std::out_of_bounds("offset from 'front' too large");
    if ((-n-1) > li.size())
        throw std::out_of_bounds("offset from 'back' too large");
    auto it = (n > 0) ?  li.begin() : li.end();
    std::advance(it, n); // works backward for negative n
    return *it;
}
```

An index may refer to the front … … or the back of an `std::list c`:

```
… get_nth(c, 0) … // first          … get_nth(c, -1) … // last
… get_nth(c, 1) … // second         … get_nth(c, -2) … // etc.
…                                   …
```

*: [illegible] `const T` [illegible] `const`.

std::list Code Examples and Fragments (3)

To sort an `std::list` with its own `sort` member function, a comparison function may be named …

```
bool compare(const T& lhs,
             const T& rhs) {
    return … // whether lhs is
}            // less-than rhs
…
c.sort(compare);
```

… or a lambda* handed over …

```
struct MyData { int id; … };
…
c.sort([](auto lhs, auto rhs) {
       return lhs.id < rhs.id;
      });
```

… or instantiating a functor, i.e. a class overloading `operator()`:

```
template<typename T> struct IdCompare {
    bool operator() const (const T& lhs, const T& rhs) {
        return lhs.id < rhs.id;
    }
};
…
c.sort(IdCompare<MyData>());
```

*: Actually a C++14 lambda because of `auto`-typed parameters. In C++11 lambda argument types need to be specified (as for `compare` left).

Class `std::stack`

Provides a *last-in/first-out* container

- that may grow or shrink at run-time,
- with efficient insertions and removals at one end.

> The class is an adaptor only, i.e. it builds on some other container class providing the member functions `back`, `push_back`, and `pop_back`.

By default `std::stack` adapts `std::deque`, but it may also be instantiated with `std::vector`.

See the pages following for

- characterisation,
- special considerations, and
- some example code fragments.

> (i) For more information on class `std::stack` see http://en.cppreference.com/w/cpp/container/stack

`std::stack` Characterisation

Property	
Header file	`<stack>`
Introduced with	C++98
Contiguous storage	depending on underlying type
Direct element lookup by ...	no
... with performance	not applicable
Efficient insertion / removal	with `push` or `emplace` / `pop`
Iterator category	none
Dynamic growth ...	yes
... may invalidate iterators	not applicable

Note that accessing the current element with member function `top` and removing it with member function `pop` are separate operations.

[!] There is a race condition in multi-threaded programs if member functions `empty` (or `size`) together with `top` and `pop` are not applied in proper sequence and/or not properly protected in a critical section.

`std::stack` Special Considerations:

- A `std::stack` effectively **reduces** the capabilities of the adapted container, by hiding some of its operations.*
- This makes sense only
  - if a certain semantic is to be enforced – like last-in/first-out in this case,
  - so that inadvertent wrong use is turned into a compile time error.
- As soon as *additional* operations are required, using this adapter is not the appropriate choice.

E.g. if besides last-in/first-out semantics also all the current content must be examined, even if only occasionally, using the underlying container directly is more appropriate.

---

*: Note that as an `std:vector` provides only member functions `push_back` and `pop`back, `but no member functions`push_front`and`pop_front`, it has "natural stack semantics", while in addition all of its content can be examined using sequential traversal or index based random access. (Though strict last-in/first-out behaviour might be bypassed with the`insert` member function.)

`std::stack` Code Examples and Fragments

A typical use of a stack is to reverse a sequence. Of course this can be done with any sequence container that can be traversed in the opposite direction from being filled, but the `std::stack` adapter protects against careless mistakes.

The following program reverses its input (text) data stream, line by line:

```
#include <iostream>
#include <stack>
#include <string>

int main() {
    std::string line;
    std::stack<std::string> data;
    while (std::getline(std::cin, line))
        data.push(line);
    while (!data.empty()) {
        std::cout << data.top() << '\n';
        data.pop();
    }
}
```

Class `std::queue`

Provides a *first-in/first-out* container

- that may grow or shrink at run-time,
- with efficient insertions at one end and efficient removals at the other.

> The class is an adaptor only, i.e. it builds on some other container class providing the member functions `back`, `front`, `push_back`, and `pop_front`.

See the pages following for

- characterisation,
- special considerations, and
- some example code fragments.

By default `std::stack` adapts `std::deque`, but it may also be instantiated with `std::list`.

> (i) For more information on class `std::queue` see http://en.cppreference.com/w/cpp/container/deque

`std::queue` Characterisation

Property	
Header file	`<queue>`
Introduced with	C++98
Contiguous storage	depending on underlying type
Direct element lookup by ...	no
... with performance	not applicable
Efficient insertion / removal	with `push` or `emplace` / `pop`
Iterator category	none
Dynamic growth ...	yes
... may invalidate iterators	not applicable

Note that accessing the current element with member function `front` and removing with member function `pop` are separate operations.

[!] There is a race condition in multi-threaded programs, if the member functions `empty` (or `size`) together with `front` and `pop` are not applied in proper sequence and/or not properly protected in a critical section.

`std::queue` Special Considerations:

- A `std::queue` effectively **reduces** the capabilities of the adapted container, by hiding some of its operations.
- This makes sense only
  - if a certain semantic is to be enforced – like first-in/first-out in this case,
  - so that inadvertent wrong use is turned into a compile time error.
- As soon as *additional* operations are required, using this adapter is not the appropriate choice*

E.g. if besides last-in/first-out semantics also all the current content must be examined, even if only occasionally, using the underlying container directly is more appropriate*

---

*: Note there is **no** *un-adapted* standard container providing insertions and removals at opposite ends **only**. But with a standard container as private base class and by lifting the desired operations into `public:` scope, a customized class may be provided with not too much effort.

std::queue Code Examples and Fragments (1)

A typical use of a queue is a temporary storage of data that needs eventually be reproduced in the order it was received.

The following fragment reproduces its input (text) data stream, printing each sentence in a single line.

First two helpers (that need to be called more than once):

```
std::queue<std::string> sentence;

void print_word(std::ostream &os, char sep) {
    os << sentence.back() << sep;
    sentence.pop();
}

void flush_sentence(std::ostream &os) {
    while (sentence.size() > 1)
        print_word(os, ' ');
    print_word(os, '\n');
}
```

std::queue Code Examples and Fragments (2)

Now the part that reads and temporarily stores word up to one than ends in a full stop:

```
void read_lines(std::istream &is, std::ostream &os) {
    std::string word;
    while (is >> word) {
        sentence.push(word);
        if (word.back() == '.')
            flush_sentence(os);
    }
    if (!sentence.empty())
        flush_sentence(os);
}
```

Note that flushing the sentence once more after the while loop is in case input ands with a word that is not followed by a full stop.

Class `std::priority_queue`

Provides a *prioritised first-in/first-out* container

- that may grow or shrink at run-time,
- with efficient insertions at one end, efficient removals at the other, and a priority-based extraction order.

> The class is an adaptor only, i.e. it builds on some other container class providing the member functions `front`, `push_back`, and `pop_back`.

See the pages following for

- characterisation,
- special considerations, and
- some example code fragments.

By default `std::priority_queue` adapts `std::vector`, but it may also be instantiated with `std::deque`.

> (i) For more information on class `std::priority_queue` see http://en.cppreference.com/w/cpp/container/priority_queue

`std::priority_queue` Characterisation

Property	
Header file	`<priority_queue>`
Introduced with	C++98
Contiguous storage	depending on underlying type
Direct element lookup by ...	no
... with performance	not applicable
Efficient insertion / removal	with `push` or `emplace` / `pop`
Iterator category	none
Dynamic growth ...	yes
... may invalidate iterators	not applicable

Note that accessing the current element with member function `top` and removing it with member function `pop` are separate operations.

[!] There is a race condition in multi-threaded programs if member functions `empty` (or `size`) together with `top` and `pop` are not applied in proper sequence and/or not properly protected in a critical section.

`std::priority_queue` Special Considerations:

- A `std::priority_queue` mostly **reduces** the capabilities of the adapted container, but also adds new behavior by storing the content in prioritised order.
- More precisely, it arranges the content so that it qualifies as a heap,_[]
  - applying the ordering relation at insertion time, i.e.
  - the ordering **must not** depends on member data that might change for an element that is already part of the priority queue.

For applying an ordering relation *immediately prior* an element is processed (respectively removed), an `std:priority_queue` is not appropriate.

*: Do not confuse this with the area of main memory managed by `new` and `delete`, also commonly called "the heap".

std::priority_queue Code Examples and Fragments

A typical of a priority queue were sorting a `MyTask` class according to its priority.

Assumming the following fragment ...

```
class MyTask {
    … //
};
bool operator<(const MyTask &lhs, const MyTask &rhs) {
    return … ; // return true if lhs has higher priority
}
```

... a (hypothetical) scheduler might manage `MyTask` instances as follows:

```
std::priority_queue<Mytask> ready_to_run;
…
ready_to_run.push( … ); // add newly created or unsuspended task
…
… = ready_to_run.top(); // select task with highest priority
…
ready_to_run.pop();     // withdraw highest-priority task
```

Class `std::set`

Provides a *set-container* that

- may hold elements with a given identity at most once,
- with reasonably fast access by element value,
- and guaranteed ordering on sequential traversal.

See the pages following for

- characterisation,
- special considerations, and
- some example code fragments.

For more information on class `std::set` see
http://en.cppreference.com/w/cpp/container/set

`std::set` Characterisation

Property	
Header file	`<set>`
Introduced with	C++98
Contiguous storage	no
Direct element lookup by ...	unique value
... with performance	$O(\log_2 N)$
Efficient insertion / removal	everywhere
Iterator category	bidirectional
Dynamic growth ...	yes
... may invalidate iterators	no

See also commonalities for sets, tree-based, and non-multi containers.

`std::set` Special Considerations

- If looking up an existing value is frequent and sequential traversal in a specific order is not required, `std::unordered_set` may be a better choice.
- If sequential traversal must happen in a specific sort order, a user specified ordering relation can be used to establish exactly that order.
- During sequential traversal it is not possible to modify the elements visited.
  - The reason is that any change may have an effect on ordering.
  - Therefore, to change a value the element must be deleted, modified and inserted again.

[!] Changing elements in an `std::set` in ways that affect sort order will very possibly corrupt the tree structure.

std::set Code Examples and Fragments

The following code reads words, then prints them sorted without duplicates:

```
std::set<std::string> words;
std::string w;
while (std::cin >> w)
    words.insert(w);
for (auto &e : words)
    std::cout << e << '\n';
```

To print the words in reverse order, either the loop may be changed ...

```
for (auto it = words.crbegin(); it != words.crend(); ++it)
    std::cout << *it << '\n';
```

... or the set could be created with a different ordering relation:

```
std::set<std::string, std::greater<std::string>> words;
```

Class `std::multiset`

Provides a *set-container* that

- may hold multiple elements with a given identity,
- with reasonably fast access by element value,
- and guaranteed ordering on sequential traversal.

See the pages following for

- characterisation,
- special considerations, and
- some example code fragments.

For more information on class `std::multiset` see
http://en.cppreference.com/w/cpp/container/multiset

`std::multiset` Characterisation

Property	
Header file	`<set>`
Introduced with	C++98
Contiguous storage	no
Direct element lookup by ...	by its value
... with performance	$O(\log_2 N)$
Efficient insertion / removal	everywhere
Iterator category	bidirectional
Dynamic growth ...	yes
... may invalidate iterators	no

See also commonalities for sets, tree-based, and multi containers.

`std::multiset` Special Considerations

- If looking up groups of entries with the same value is frequent and except from this traversal in a specific order is not required, `std::unordered_multiset` may be better choice.
- Using an `std::vector` that is sorted before processing should be considered in usage pattern which
  - *first* builds the complete data set, and
  - *then* processes it (without further additions).

Special considerations for `std::set` apply for `std::multiset` too.

std::multiset Code Examples and Fragments

The following code reads words and prints them, with the shortest word first and the longest last:

```
struct WordLengthOrder {
    bool operator()(const std::string &lhs, std::string &rhs) {
        return lhs.length() < rhs.length();
};
std::multiset<std::string, WordLengthOrder> words;
std::string w;
while (std::cin >> w)
    words.insert(w);
for (const auto &e : words)
    std::cout << e < '\n';
```

Alternatively the ordering relation may part of the object instance:*

```
std::multiset<std::string> words(WordLengthOrder());
```

---

*: Or by replacing the functor with a (C++14) lambda:

```
std::multiset<std::string> words([](auto lhs, auto rhs) { return lhs.length() < rhs.length(); });
```

Class `std::map`

Provides a *key-value lookup table* that

- may hold an entry with a given key at most once,
- with reasonably fast access by key,
- and guaranteed ordering on sequential traversal.

See the pages following for

- characterisation,
- special considerations, and
- some example code fragments.

For more information on class `std::map` see
http://en.cppreference.com/w/cpp/container/map

`std::map` Characterisation

Property	
Header file	`<map>`
Introduced with	C++98
Contiguous storage	no
Direct element lookup by ...	unique key
... with performance	$O(\log_2 N)$
Efficient insertion / removal	everywhere
Iterator category	bidirectional
Dynamic growth ...	yes
... may invalidate iterators	no

See also commonalities for maps, tree-based, and non-multi containers.

`std::map` Special Considerations

- If looking up an existing entry by its key is frequent and sequential traversal in a specific key order is not required, `std::unordered_map` may be the more performant alternative.
- If sequential traversal must happen in a specific sort order, a user specified ordering relation can be used to establish exactly that order.
- It is always possible to modify the value part of an entry, but not its key (while leaving the entry in place).

To modify the key of an object, erase and re-insert it with a new key.

`std::map` Code Examples and Fragments

The following code counts how often the words are occur in the input:

```
std::map<std::string, int> words;
std::string w;
while (std::cin >> w)
    ++words[w];
for (const auto &e : words)
    std::cout << e.first << ": " << e.second << '\n';
```

Alternative this could be done also with an `std::multimap`:

```
std::set<std::string> words;
std::string w;
while (std::cin >> w)
    words.insert(w);
for (auto it = words.cbegin(); it != words.cend(); )
    auto cnt = words.count(*it);
    std::cout << *it << ": " cnt << '\n';
    advance(it, cnt);
}
```

Class `std::multimap`

Provides a *key-value lookup table* that

- may hold an entry with a given key more than once,
- with reasonably fast access by key,
- and guaranteed ordering on sequential traversal.

For more information on class `std::multimap` see
http://en.cppreference.com/w/cpp/container/multimap

`std::multimap` Characterisation

Property	
Header file	`<map>`
Introduced with	C++98
Contiguous storage	no
Direct element lookup by ...	key
... with performance	$O(\log_2 N)$
Efficient insertion / removal	everywhere
Iterator category	bidirectional
Dynamic growth ...	yes
... may invalidate iterators	no

See also commonalities for maps, tree-based, and multi-containers.

`std::multimap` Special Considerations

- If looking up an existing entry by its key is frequent and sequential traversal in a specific key order is not required, `std::unordered_multimap` may be the more performant alternative.
- Instead of a multimap also consider a combination of
  - the non-multi variant, with
  - an element type that is another container for the data to store per key.
- If sequential traversal must happen in a specific sort order, a user specified ordering relation can be used to establish exactly that order.
- It is always possible to modify the value part of an entry, but not its key (while leaving the entry in place).

To modify the key of an object, erase and re-insert it with a new key.

std::multimap Code Examples and Fragments

The following fragment reads lines starting with a word and a colon.

The word is then used to store the line in a multimap:

```
std::multimap<std::string, std::string> text_lines;
…
std::string line;
while (std::getline(std::cin, line)) {
    std::istringstream iss(line);
    std::string key;
    if (std::getline(iss, key, ':')
        text_lines.insert(std::make_pair(key, line));
}
```

Iterating over the multimap will reproduce the lines in ascending key order, but **until C++11 not necessarily in insertion order** for keys that compare equal.

Class `std::unordered_set`

Provides a *set-container* that

- may hold elements with a given identity at most once,
- with fast (O(1)) access by element value,
- but no guaranteed ordering on sequential traversal.

See the pages following for

- characterisation,
- special considerations, and
- some example code fragments.

For more information on class `std::unordered_set` see
http://en.cppreference.com/w/cpp/container/unordered_set

`std::unordered_set` Characterisation

Property	
Header file	`<unordered_set>`
Introduced with	C++11
Contiguous storage	no
Direct element lookup by ...	unique value
... with performance	O(1)
Efficient insertion / removal	everywhere
Iterator category	bidirectional
Dynamic growth ...	yes
... may invalidate iterators	yes

See also commonalities for sets, hash-based, and non-multi-containers.

`std::unordered_set` Special Considerations

- If looking up entries is less frequent but traversal in a particular order is desirable, consider to chose `std::set` as an alternative.
- During sequential traversal it is not possible to modify the elements visited.
  - The reason is that any change may have an effect on calculating the hash index.
  - Therefore, to change a value the element must be deleted, modified and inserted again.

[!] Changing elements in an `std::unordered_set` in ways that affect hashing will very possibly corrupt the hash structure.

std::unordered_set Examples and Fragments (1)

The following example combines `std::list` and `std::unordered_set`, providing O(1) performance for existence checks and traversal in insertion order.

At first some data type and the containers:

```
using ValueType = …;
…
std::list<ValueType> sequence;
std::unordered_set<ValueType> lookup;
```

When new elements come in, both containers need to be updated:

```
sequence.push_back(ValueType( … , … ));
// or: sequence.emplace_back( … , … );
lookup.insert(sequence.back());
```

To remove entries with a given value, remove them from both containers:

```
sequence.remove(ValueType( … , … ));
lookup.erase(ValueType( … , … ));
```

std::unordered_set Examples and Fragments (2)

For sequentially traversal in insertion order simply iterate over `sequence`:

```
for (const auto &e : sequence) …
```

Or prior to C++11:

```
for (std::list<ValueType>::const_iterator it = sequence.cbegin();
                                          it != sequence.cend();
                                        ++it) …
```

Looking up if some element exists is done via the other container:*

```
if (lookup.count(Value( …, … )) …
```

*: Note that the `count` member function will give the same performance `find` for the non-`multi`-variant of a set but is somewhat easier to use.

Class `std::unordered_multiset`

Provides a *set-container* that

- may hold multiple elements with a given value,
- with fast (O(1)) access by element value,
- but no guaranteed ordering on sequential traversal.*

*: All elements with the same value are usually visited one after the other.

`std::unordered_multiset` Characterisation

Property	
Header file	`<unordered_set>`
Introduced with	C++11
Contiguous storage	no
Direct element lookup by ...	value
... with performance	O(1)
Efficient insertion / removal	everywhere
Iterator category	bidirectional
Dynamic growth ...	yes
... may invalidate iterators	yes

See also commonalities for sets, hash-based, and multi-containers.

`std::unordered_multiset` Special Considerations

- If looking up values is less frequent but traversal in a particular order is desirable, consider to chose `std::multiset` as an alternative.
- During sequential traversal it is not possible to modify the values of the elements visited.
  - The reason is that any change may have an effect on calculating the hash index.
  - Therefore, to change a value the element must be deleted, modified and re-inserted.

[!] Changing a value in an `std::unordered_set` in ways that affect hashing will very possibly corrupt the hash structure.

std::unordered_multiset Code Examples and Fragments

The following is an alternative to analyse word frequency with a map.

Note that the correctness depends on the guarantee – *which is explicitly stated in the C++ standard* – that values comparing equal to each other are visited consecutively:

```
#include <iostream>
#include <string>
#include <unordered_set> // also defines the multi_-variant

int main() {
    std::string word;
    std::unordered_multiset<std::string> all_words;
    while (std::cin >> word)
        all_words.insert(word);
    auto it = all_words.cbegin();
    while (it != all_words.cend()) {
        const auto wcount = all_words.count(word);
        std::cout << word << ": " << wcount << '\n';
        std::advance(it, wcount);
    }
}
```

Class `std::unordered_map`

Provides a *key-value lookup table* that

- may hold an entry with a given key at most once,
- with fast (O(1)) access by key,
- but no guaranteed ordering on sequential traversal.

For more information on class `std::unordered_map` see
http://en.cppreference.com/w/cpp/container/unordered_map

`std::unordered_map` Characterisation

Property	
Header file	`<unordered_map>`
Introduced with	C++11
Contiguous storage	no
Direct element lookup by ...	unique key
... with performance	O(1)
Efficient insertion / removal	everywhere
Iterator category	bidirectional
Dynamic growth ...	yes
... may invalidate iterators	yes

See also commonalities for maps, hash-based, and non-multi-containers.

`std::unordered_map` Special Considerations

- If looking up keys is less frequent but traversal in a particular order is desirable, consider to chose `std::map` as an alternative.
- During sequential traversal it is not possible to modify the keys of the elements visited.
  - The reason is that any change may have an effect on calculating the hash index.
  - Therefore, to change a key the element must be deleted, modified and re-inserted with the new key.

[!] Changing a key in an `std::unordered_set` in ways that affect hashing will very possibly corrupt the hash structure.

`std::unordered_map` Examples and Fragments

The following extends the principle of combining two containers to provide O(1) lookup performance with sequential access in insertion order.

> The example assumes that keys and values are separate and the keys are not of interest for sequential traversal.

Therefore `lookup` associates keys with iterators into `sequence`.

First some type definitions and the two containers:

```
using KType = …;   // or until C++11: typedef … KType;
using VType = …; // or until C++11: typedef … VType;
…
std::list<VType> sequence;
std::unordered_multimap<KType, std::list<VType>::iterator>> lookup;
```

std::unordered_map Examples and Fragments (2)

Insertion needs to take place for both containers:*

```
if (sequence.find(KType( … )) == sequence.end())
    sequence.push_back(VType( … , … ));
// or: sequence.emplace_back( … , … );
auto sequence_end = sequence.end();
lookup.insert(std::make_pair(KType( … ), --sequence_end));
```

Sequential traversal is as simple as a range-`for` over `sequence`:

```
for (auto e : sequence)
    … // access VType elements via e (copy) ...
for (auto &e : sequence)
    … // ... or efficiently via reference (modifiable) ...
for (const auto &e : sequence)
    … // ... or efficiently via non-modifiable reference
```

---

*: With `std::prev` introduced in C++11, the last two lines can be easily combined to one:

```
loopup.insert(std::make_pair(KType( … ), std::prev(sequence.end())));
```

`std::unordered_map` Examples and Fragments (3)

Or, if traversal needs to include keys and no particular is order required:

```
for (auto &e : lookup) {
   … // access KType key via e.first (always non-modifiable)
   … // access Vtype element via e.second (add 'const' if
                 // compiler shall guarantee non-modifiable)
}
```

Finally, looking up a value for a specific key can be done as follows:*

```
auto entry_it = lookup.find(KType( … ));
if (entry_it != lookup.end()) {
    auto &key = entry_it->first;
    auto &value = *entry_it->second;
               // ^--- double dereference required,
               // since lookup holds iterators only
}
else … // no such key
```

---

*: Still simpler for keys that are expected or guaranteed to exist:
`… *lookup.at(key) …` – (since C++11) may throw `std::out_of_range`;
`… *lookup[key] …` – but entry for `key` is created if it doesn't exist
and **access via dereferencing causes Undefined Behaviour!**

Class `std::unordered_multimap`

Provides a *key-value lookup table* that

- may hold multiple entries with a given key,
- with fast (O(1)) access by key,
- but no guaranteed ordering on sequential traversal.

For more information on class `std::unordered_map` see
http://en.cppreference.com/w/cpp/container/unordered_map

`std::unordered_multimap` Characterisation

Property	
Header file	`<unordered_map>`
Introduced with	C++11
Contiguous storage	no
Direct element lookup by ...	key
... with performance	O(1)
Efficient insertion / removal	everywhere
Iterator category	bidirectional
Dynamic growth ...	yes
... may invalidate iterators	yes

See also commonalities for maps, hash-based, and multi-containers.

`std::unordered_multimap` Special Considerations

- If looking up entries by key is less frequent but traversal in a particular key order is desirable, consider to chose `std::multimap` as an alternative.
- To store several values per key, also the non-`multi`-variant may be used with an appropriate (sub-) container as value part.
- It is always possible to modify the value part of an entry, but not its key (while leaving the entry in place).

> To modify the key of an object, erase and re-insert it with a new key.

`std::unordered_multimap` Examples and Fragments (1)

The following extends the principle of combining two containers to provide O(1) lookup performance with sequential access in insertion order.

> The example assumes that keys and values are separate and the keys are not of interest for sequential traversal.

Therefore `lookup` associates keys with iterators into `sequence`.

First some type definitions and the two containers:

```
using KType = …;   // or until C++11: typedef … KType;
using VType = …; // or until C++11: typedef … VType;
…
std::list<VType> sequence;
std::unordered_multimap<KType, std::list<VType>::iterator>> lookup;
```

std::unordered_multimap Examples and Fragments (2)

Insertion needs to take place for both containers:

```
sequence.push_back(VType( … , … ));
// or: sequence.emplace_back( … , … );
lookup.insert(std::make_pair(KType( … ),
              std::prev(sequence.end())));
```

Sequential traversal is as simple as a range-`for` over `sequence`:

```
for (auto e : sequence) … // access VType elements via e (copy)
for (auto &e : sequence) … // ... or via reference (modifiable)
for (const auto &e : sequence) … // ... or non-modifiable
```

Finally, a range of keys that compare equal to each other may be found that way:

```
auto const range = lookup.equal_range(KType( … ));
… range.first …  // iterator to first entry with given key
… range.second … // iterator one beyond last entry with given key
```

Practically Working with Containers

For a practical approach to gather experience with STL containers visit the Micro Projects focussing on STL containers.

For more information on the STL Algorithm Library see:
http://en.cppreference.com/w/cpp/algorithm