6 C++ standard library

This chapter covers the data structures and algorithms in the C++ standard library.

The emphasis is on things that make using the library purposefull and efficient.

Topics that are important from the library implementation point of view are not discussed here.

6.1 General information on the C++ standard library

The standard library, commonly called STL, was standardized together with the C++-language autumn 1998, and it has been somewhat extended in later versions. The latest version of the standard C++17 is from 2017.

It contains the most important basic data structures and algorithms.

- most of the data structures and algorithms covered earlier in this material in one shape or another

It also contains a lot more such as

- input / output: `cin`, `cout`, ...
- processing character strings
- minimum, maximum
- search and modifying operations of queues
- support for functional programming
- complex numbers

- arithmetic functions (e.g. `sin`, `log10`),
- vector arithmetics and support for matrix operations

The interfaces are carefully thought out, flexible, generic and type safe

The efficiency of the operations the interfaces provide has been given with the $\mathcal{O}$-notation.

The compile time C++ template mechanism has been used to implement the genericity.

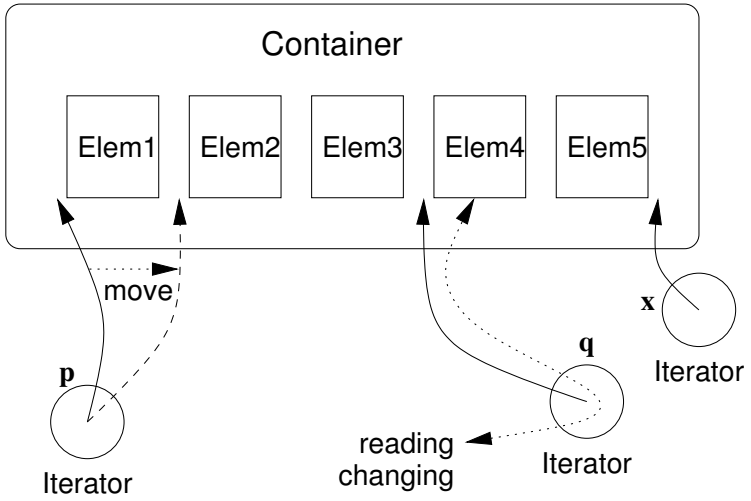
Interesting elements in the standard library from the point of view of a data structures course are the *containers*, i.e. the data structures provided by the library and the *generic algorithms* together with the *iterators*, with which the elements of the containers are manipulated.

Lambdas were introduced with C++11 and play a key role as well.

6.2 Iterators

We see all standard library data structures as black boxes with a lot of common characteristics. The only thing we know is that they contain the elements we've stored in them and that they implement a certain set of interface functions.

We can only handle the contents of the containers through the interface functions and with iterators.



Iterators are handles or "bookmarks" to the elements in the data structure.

- each iterator points to the beginning or the end of the data structure or between two elements.
- the element on the right side of the iterator can be accessed through it, except if the iterator in question is a *reverse iterator* which is used to access the element on the left side.
- the operations of moving the reverse iterator work in reverse, for example ++ moves the iterator one step to the left
- the interface of the containers usually contains the functions **begin()** and **end()**, that return the iterators pointing to the beginning and the end of the container
- functions **rbegin()** and **rend()** return the equivalent reverse iterators
- an iterator can be used to iterate over the elements of the container, as the name suggests
- an iterator can be used for reading and writing

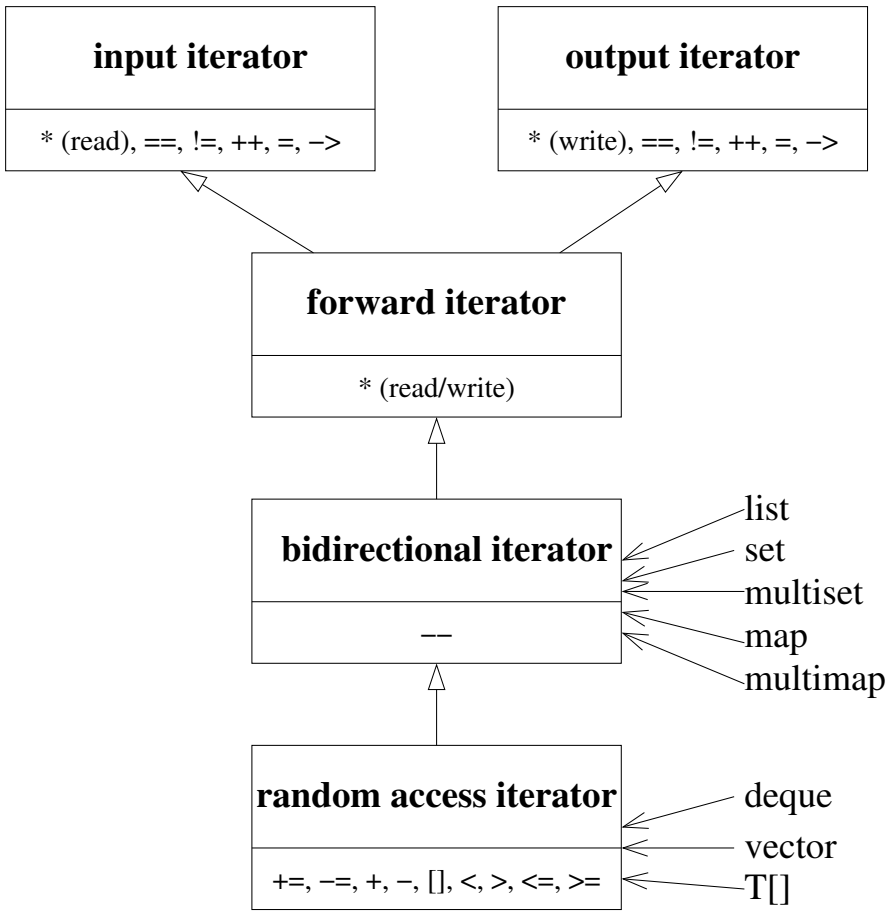
- the location of the elements added to or removed from a container is usually indicated with iterators

Each container has its own iterator type.

- different containers provide different possibilities of moving the iterator from one place to another efficiently (cmp. reading an arbitrary element from a list/array)
- the design principle has been that all iterator operations must be performed in constant time to quarantee that the generic algorithms work with the promised efficiency regardless of the iterator given to them
- iterators can be divided into categories based on which constant time operations they are able to provide

An *input iterator* can only read element values but not change them

- the value of the element the iterator points to can be read (`*p`)
- a field of the element the iterator points to can be read or its member function can be called (`p->`)
- the iterator can be moved one step forwards (`++p` or `p++`)
- iterators can be assigned to and compared with each other (`p=q`, `p==q`, `p!=`)



An *output iterator* is like an input iterator but it can be used only to change the elements (`*p=x`)

A *forward iterator* is a combination of the interfaces of the input- and output iterators.

A *bidirectional iterator* is able to move one step at a time backwards(`--p` or `p--`)

A *random access iterator* is like a bidirectional iterator but it can be moved an arbitrary amount of steps forwards or backwards.

- the iterator can be moved `n` steps forwards or backwards (`p+=n`, `p-=n`, `q=p+n`, `q=p-n`)
- an element `n` steps from the iterator can be read and it can be modified (`p[n]`)
- the distance between two iterators can be determined (`p-q`)
- the difference of two iterators can be compared, an iterator is "smaller" than the other if its location is earlier than the other in the container (`p<q`, `p<=q`, `p>q`, `p>=q`)

The syntax of the iterator operations is obviously similar to the pointer aritmetic of C++.

Iterators can be used with
`#include <iterator>`

An iterator of a correct type can be created with the following syntax for example.

```
container<type stored>::iterator p;
```

The key word `auto` is useful with iterators:

```
auto p = begin( cont );
// → std::vector<std::string>::iterator
```

The additions and removals made to the containers may *invalidate* the iterators already pointing to the container.

- this feature is container specific and the details of it are covered with the containers

In addition to the ordinary iterators STL provides a set of iterator adapters.

- they can be used to modify the functionality of the generic algorithms
- the *reverse iterator*s mentioned earlier are iterator adapters
- *insert iterators/inserters* are one of the most important iterator adapters.
  - they are output iterators that insert elements to the desired location instead of copying
  - an iterator that adds to the beginning of the container is given by the function call `front_inserter`(container)
  - an iterator that adds to the end is given by `back_inserter`(container)
  - an iterator that adds to the given location is given by `inserter`(container, location)

- *stream iterators* are input and output iterators that use the C++ streams instead of containers
  - the syntax for an iterator that reads from a stream with the type `cin` is `istream_iterator`<type> (`cin`)
  - the syntax for an iterator that prints the given data to the `cout` stream data separated with a comma is `ostream_iterator`<type> (`cout, ','`)
- *move iterators* replace the copying of an element with a move.

6.3 Containers

The standard library containers are mainly divided into two categories based on their interfaces:

- *sequences*
  - elements can be searched based on their number in the container
  - elements can be added and removed at the desired location
  - elements can be scanned based on their order in the container
- *associative containers*
  - elements are placed in to the container to the location determined by their *key*
  - by default the operator < can be used to compare the key values of the elements in ordered containers
- the interface's member functions indice how the container is supposed to be used.

The containers:

Container type	Library

Sequences	`array`
	`vector`
	`deque`
	`list`
	(`forward_list`)

Assosiative containers	`map` `set`
unordered assosiative	`unordered_map` `unordered_set`
Adapters	`queue` `stack`

Containers are passed by value.

- the container takes a copy of the data stored in it
- the container returns copies of the data it contains

  $\Rightarrow$ changes made outside the container do not effect the data stored in the container
- all elements stored into the containers must implement a copy constructor and an assignment operator.
  - basic types have one automatically
- elements of a type defined by the user should be stored with a pointer pointing to them
  - this is sensible from the efficiency point of view anyway

- the `shared_pointer` is available in C++11 for easier management of memory is situations where several objects share a resource
  - contains an inbuild reference counter and deletes the element once the counter becomes zero
  - there is no need to call delete. Furthermore is must not be called.
  - definition: `auto pi = std::make_shared<Olio>(params);`
  - handy especially if a data structure ordered based on two different keys is needed
    * store the satellite data with a `shared_pointer`
    * store the pointers in two different containers based on two different keys

Sequence containers:

Array `array<type>` is a constant sized array.

- Initialization `std::array<type, size> a = {val, val,...};`
- Can be indexed with `.at()` or with `[]`. Functions `front()` and `back()` access the first and the last element.
- Provides iterators and reverse iterators.
- `empty()`, `size()` and `max_size()`
- The function `data()` accesses the underlying array.

Operations are constant time except `fill()` and `swap()` which are $O(n)$

Vector `vector<type>` is an flexible sized array where additions and removals are efficient at the end

- Initialization `vector<int> v {val, val, ...};`
- Constant time indexing with `.at()`, `[]` and a (amortized) constant time addition with `push_back()` and removal with `pop_back()` at the end of the vector.
- Inserting an element elsewhere with `insert()` and removal with `erase` is linear, $O(n)$
- `emplace_back(args);` builds the element directly into the vector
- The size can be increased with `.resize(size, initial_val);`
  - the initial value is voluntary
  - if needed, the vecor can allocate more memory automatically
  - memory can also be reserved in advance: `.reserve(size)`, `.capacity()`
- iterators are invalidated in the following situations

  - if the vector originally didn't have enough space reserved for it any addition can cause the invalidation of all iterators
  - the removals only invalidate those iterators that point to the elements after the removal point
  - additions to the middle always invalidate the iterators after the addition point
- a special implementation has been given to `vector<bool>` which is different from the general implementation the templates would create in order to save memory
  - the goal: makes 1 bit / element possible where the ordinary implementation would probably use 1 byte / element i.e. 8 bits / element

Deque `deque<type>` is an array open at both ends

- initialization: `deque<type> d {val, val, val...};`
- provides similar services and almost the same interface as vector
- in addition, provides an efficient ($O(1)$ amortized running-time) of addition and removal at *both* ends
  `.push_front(val)`, `.emplace_front(args)`, `.pop_front()`
- iterators are invalidated in the following situations
  - all additions can invalidate the iterators
  - removals at the middle invalidate all iterators
  - all addition and removal operations elsewhere except at the ends can invalidate references and pointers

List is a container that support bidirectional iteration

- initialization: `list<type> l {val, val, val };`
- addition and removal is everywhere constant time, there is no indexing operation
- addition and removal don't invalidate the iterators or references (except naturally to elements removed)
- list provides several special services
  - `.splice(location, list2)` makes the other list a part of the second list infront of the location
  - `.splice(location, list, val)` moves an element from another list or the same list infront of the location
  - `.splice(location, list, beg, end)`
  - `.merge(list2)` and `.sort()`, stable, $O(nlogn)$ on average
  - `.reverse()`, linear

Amortized running time

Vector is a flexible sized array, i.e. the size increases when needed

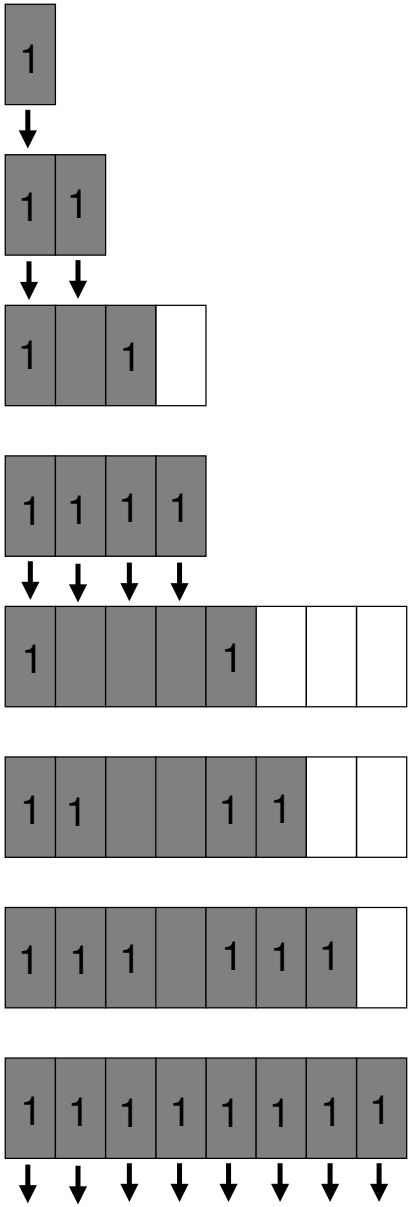
- when a new element no longer fits into the array, a new, larger one is allocated and all the elements are moved there
- the array never gets smaller $\Rightarrow$ the memory allocation is only reduced when a new contents are copied on top of the old one

$\Rightarrow$ Adding an element to the end of the vector was mentioned as *amortized* constant time

- the performance is analyzed as a unit, the efficiency of a sequence of operations is investigated instead of single operations
- each addition operation that requires expensive memory allocation is preceeded by an amount of inexpensive additions relative to the price of the expensive operation

- the cost of the expensive operation can be equally divided to the inexpensive operations
- now the inexpensive operations are still constant time, although they are a constant coefficient more inefficient than in reality
- the savings can be used to pay for the expensive operation

$\Rightarrow$ all addition operations at the end of a vector can be done in amortized constant time



This can be proven with the accounting method:

- charge three credits for each addition
- one credit is used for the real costs of the addition
- one credit is saved with the element added to i
- one credit is saved at the element at $i - \frac{1}{2} \cdot vector.capacity()$
- when the size of the array needs to be increased, each element has one credit saved and the expensive copying can be payed for with them

Associative containers:

`set<type>` and `multiset<type>` is a dynamic set where

- elements can be searched, added and deleted in logarithmic time
- elements can be scanned in ascending order in amortized constant time so that the scan from the beginning to the end is always a linear time operation
- an order of size must be defined for the elements "<"
  - can be implemented separately as a part of the type or as a parameter of the constructor
- determines the equality with $\neg(x<y \vee y<x)$

  $\Rightarrow$ a sensible and efficient definition to "<" needs to be given
- the same element can be in the multiset several times, in the set the elements are unique
- changing the value of the element directly has been prohibited

  - the old element needs to be removed and a new one added instead
- interesting operations:
  - `.find(val)` finds the element (first of many in the multiset) or returns `.end()` if it isn't found
  - `.lower_bound(val)` finds the first element $\geq$ element
  - `.upper_bound(val)` finds the first $>$ element
  - `.equal_range(val)`returns `make_pair( .lower_bound(val), .upper_bound(val) )` but needs only one search (the size of the range is 0 or 1 for the set)
  - for sets `insert` returns a pair (location, added), since elements already in the set may not be added
- the standard quarantees that the iterators are not invalidated by the addition or removal (except of course to the elements removed)

`map<key_type, val_type>` and
`multimap<key_type, val_type>`

- store (key, satellite data) pairs
  - the type of the pair is `pair<type1, type2>`
  - a pair can be created with the function `make_pair`
  - the fields of the pair are returned by `.first(), .second()`
- initialization `std::map<key_type, val_type> m {{key1, val1}, {key2, val2}, key3, val3},...};`
  e.g.
  ```
  std::map<std::string,int> anim { {"bear",4}, {"giraffe",2}, {"tiger",7} };
  ```
- `map` can be exceptionally indexed with the key $O(nlogn)$
  - If the key is not found, a new valuepair key-type is added to the container
- the iterators are not invalidated by the addition or removal

Hash tables Unordered set/multiset is a structure that contains a set of elements and unodered map/multimap contains a set of key-value pairs.

- the interfaces of unordered-map/set resemble map and set
- the most significant differences:
  - the elements are not ordered (unordered)
  - addition, removal and searching is on average constant time and in the worst-case linear
  - a set of member function valuable for hashing, such as `rehash(size)`, `load_factor()`, `hash_function()` ja `bucket_size()`.
- the size of the hash table is automatically increased in order to keep the load factor of the buckets under a certain limit
  - changing the size of the hash table *(rehashing)* is on average linear, worst-case quadratic
  - rehashing invalidates all iterators but not pointers or references

Additionally other containers are found in the Standard library:

`bitset<bit_amount>`

- `#include<bitset>`
- for handling fixed sized binary bitsets
- provides typical operations (AND, OR, XOR, NOT)

Strings `string`

- `#include <string>`
- the character string has been optimized for other purposes in C++ and they are usually not percieved as containers. However, they are, in addition to other purposes, also containers.
- store characters but can be made to store other things also
- they provide, among others, iterators, `[...]`, `.at(...)`, `.size()`, `.capacity()` and `swap`
- string can get very large and automatically allocate new memory when necessary

- Care should be taken with the modifying operations of strings (catenation, removal) since they allocate memory and copy elements, which makes them heavy operations for long strings
- it is often sensible anyway to store the strings with a pointer when for example storing them into the containers to avoid needless copying
- for the same reason strings should be passed by reference

In addition to containers, STL provides a group of container adapters, that are not containers themselves but that can be "adapted into a new form" through the interface of the container:

Stack `stack<element_type, container_type>`

- provides in addition to the normal class-operations only
  - stack-operations, `.push(...)`, `.top()`, `.pop()`
  - size queries `.size()` and `.empty()`
  - comparisons "==", "<" etc.
- `.pop()` doesn't return anything, the topmost element is evaluated with `.top()`
- the topmost element of the stack can be changed in place:

```
stack.top() = 35;
```

- what's interesting from the course's point of view is that the user can choose an implementation based on different containers
  - any container that provides `back()`, `push_back()` and `pop_back()` is usable. Especially `vector`, `list` and `deque`.
  - `stack<type> basic_stack;` (`deque`)
  - `stack<type, list<type> > list_stack;`

Queue `queue<element_type, container_type>`

- queue operations `.push(...)`, `.pop()`, `.front()`, `.back()`(!)
- otherwise more or less like the stack

Priority queue `priority_queue<element_type, container\_type>`

- has an almost identical interface as the queue
- implemented with a heap
- any container that provides `front()`, `push_back()` and `pop_back()` and random access iteration can be used. Especially `vector` (default) and `deque`
- elements have a different order: `.top()` returns the largest
- returns any of the equal elements
- the topmost element cannot be changed with `top` in place
- like with associative containers, the sorting criterion can be given as a parameter to <> or as the constructor parameter

data-structure	add to end	add elsewhere	remove 1st elem.	remove elem.	nth elem. (index)	search elem.	remove largest
array					$O(1)$	$O(n)_{[2]}$	
vector	$O(1)$	$O(n)$	$O(n)$	$O(n)_{[1]}$	$O(1)$	$O(n)_{[2]}$	$O(n)_{[3]}$
list	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)_{[3]}$
deque	$O(1)$	$O(n)_{[4]}$	$O(1)$	$O(n)_{[1]}$	$O(1)$	$O(n)_{[2]}$	$O(n)_{[3]}$
stack$_{[9]}$	$O(1)$			$O(1)_{[5]}$			
queue$_{[9]}$		$O(1)_{[6]}$		$O(1)_{[7]}$			
priority queue$_{[9]}$		$O(\log n)$ [10]					$O(\log n)_{[8]}$
set (multiset)		$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(\log n)$	$O(\log n)$
map (multimap)		$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(\log n)$	$O(\log n)$
unordered_ (multi)set		$O(n)$ $\approx \Theta(1)$		$O(n)$ $\approx \Theta(1)$		$O(n)$ $\approx \Theta(1)$	$O(n)$ $O(n)$
unordered_ (multi)map		$O(n)$ $\approx \Theta(1)$		$O(n)$ $\approx \Theta(1)$		$O(n)$ $\approx \Theta(1)$	$O(n)$ $O(n)$

[1] constant-time with the last element, linear otherwise
[2] logarithmic if the container is sorted, else linear
[3] constant time if the data structure is sorted, else linear
[4] constant time with the first element, else linear
[5] possible only with the last element
[6] addition possible only at the beginning
[7] removal possible only at the end
[8] query constant time, removal logarithmic
[9] container adapter
[10] addition is done automatically based on the heap order

6.4 Generic algorithms

The standard library contains most of the algorithms covered so far.

All algorithms have been implemented with function templates that get all the necessary information about the containers through their parameters.

The containers are not however passed directly as a parameter to the algorithms, iterators to the containers are used instead.

- parts of the container can be handled instead of the complete container
- the algorithm can get an iterator to containers of different type which makes it possible to combine the contents of a vector and a list and store the result into a set
- the functionality of the algorithms can be changes with iterator adapters

- the programmer can implement iterators for his/her own data structures which makes using algorithms possible also with them

Not all algorithms can be applied to all data structures efficiently.
$\Rightarrow$ part of the algorithms accept only iterators from one iterator category as parameters.

- this ensures the efficiency of the algorithms since all operations the iterator provides are constant time
- if the iterator is of an incorrect type, a compile-time error is given

  $\Rightarrow$ if the algorithm is given a data structure for which it cannot be implemented efficiently, the program won't even compile

The standard library algorithms are in `algorithm`. In addition the standard defines the C-language library `cstdlib`.

Algorithms are divided into three main groups: Non-modifying sequence operations, modifying sequence operations and sorting and related operations.

A short description on some of the algorithms that are interesting from the course's point of view (in addition there are plenty of straightforward scanning algorithms and such):

Binary search

- `binary_search`$(first, \quad end, \quad value)$ tells if the $value$ is in the sorted sequence
  - $first$ and end are iterators that indicate the beginning and the end of the search area, which is not necessarily the same as the beginning and the end of the data structure
- there can be several successive elements with the same value

  $\Rightarrow$ `lower_bound` and `upper_bound` return the limits of the area where the $value$ is found
  - the lower bound is and the upper bound isn't in the area
- the limits can be combined into a pair with one search: `equal_range`
- cmp. BIN-SEARCH

Sorting algorithms

- `sort(`beg`,` end`)` and `stable_sort(`beg`,` end`)`
- the running-time of `sort` is $O(nlogn)$ and `stable_sort` is $O(nlogn)$ is enough memory is available and $O(nlog^2n)$ otherwise
- the sorting algorithms require random access iterators as parameter

  $\Rightarrow$ cannot be used with lists, but list provides a `sort` of its own (and an non-copying `merge`) as a member function
- there is also a sort that ends once a desired amount of the first elements are sorted: `partial_sort(`beg`,` $middle$`,` end`)`
- in addition `is_sorted(`beg`,` end`)` and `is_sorted_until(`beg`,` end`)`

`nth_element(` $first$`,` nth`,` end `)`

- finds the element that would be at index nth in a sorted container
- resembles RANDOMIZED-SELECT
- iterators must be random-access

Partition

- `partition(`$first$`,` end`,` $condition$`)` unstable
- `stable_partition(`$first$`,` end`,` $condition$`)` stable but slower and/or reserves more memory
- sorts the elements in the range $first$ - end so that the elements for which the $condition$-function returns true come first and then the ones for which $condition$ is false.
- cmp. QUICK-SORT's PARTITION
- the efficiency of partition is linear
- the iterators must be bidirectional
- in addition `is_partitioned` and `partition_point`

`merge(` $beg1$ `,` $end1$ `,` $beg2$ `,` $end2$ `,` $target$`)`

- The algorithm merges the elements in the ranges $beg1$ - $end1$ and $beg2$ - $end2$ and copies them in an ascending order to the end of the iterator $target$
- the algorithm requires that the elements in the two ranges are sorted
- cmp. MERGE
- the algorithm is linear
- beg- and end-iterators are input iterators and $target$ is an output iterator

Heaps

- Heap algorithms equivalent to those described in chapter 3.1. can be found in STL
- `push_heap(` $first$ `,` end`)` HEAP-INSERT

- `pop_heap(` $first$`,` end `)` makes the topmost element the last (i.e. to the location $end - 1$) and executes HEAPIFY to the range $first \ldots end - 1$
  - cmp. HEAP-EXTRACT-MAX
- `make_heap(` $first$`,` end`)` BUILD-HEAP
- `sort_heap(` $first$`,` end `)` HEAPSORT
- the iterators must be random-access
- in addition `is_heap` and `is_heap_until`

Set operations

- The C++ standard library contains functions that support this
- `includes(` $first1$, $end1$, $first2$, $end2$ `)` subset $\subseteq$
- `set_union(` $first1$, $end1$, $first2$, $end2$, $result$ `)` union $\cup$
- `set_intersection(...)` intersection $\cap$
- `set_difference(...)` difference -
- `set_symmetric_difference(...)`
- $first$- and end-iterators are input iterators and $result$ is an output iterator

`find_first_of(` $first1$, $end1$, $first2$, $end2$ `)`

- there is a condition in the end that limits the elements investigated
- finds the first element from the first queue that is also in the second queue
- the queue can be an array, a list, a set , . . .

- a simple implementation is $\Theta(nm)$ in the worst case where n and m are the lengths of the queues
- the second queue is scanned for each element in the first queue
  – the slowest case is when nothing is found
  $\Rightarrow$ slow if both queues are long
- the implementation could be made simple, efficient and memorysaving by requiring that the queues are sorted

NOTE! None of the STL algorithms automatically make additions or removals to the containers but only modify the elements in them

- for example `merge` doesn't work if it is given an output iterator into the beginning of an empty container
- if the output iterator is expected to make additions instead of copying the iterator adapter insert iterator must be used (chapter 6.2)

6.5 Lambdas: ()(){}

Situations where a need to pass functionality on to the functions arise often with the algorithm library

- e.g. `find_if`, `for_each`, `sort`

Lambdas are nameless functions of an undefined type. They take parameters, return a value and can refer to the creation environment and change it.

Syntax: `[environment](parameters)->returntype {body}`

- `environment` is empty if the lamba does not refer to its environment
- `parameter` can be left out
- if there is no `->returntype` it is void. It can also be deducted from a simple return statement.
- e.g. `[](int x, int y){ return x+y;}`
  `for_each( v.begin(), v.end(),[] (int val) {cout<<val<<endl;});`
  `std::cin >> lim;` //local var
  `std:find_if(v.begin(), v.end(),[lim](int a){return a<lim;});`

STL algorithms can be seen as named special loops whose body is given in the lambda.

```
bool all = true;
for (auto i : v)
{
    if (i%10 != 0) {
        all = false;
        break;
    }
}
if (all) {...}

if (std::all_of(v.begin(), v.end(), [](int i){return i%10==0;}){...}
```