# 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

Summer Semester 2018

# The Standard Template Library (STL)

The Standard Template Library (STL) is a part of the C++ standard that provides classes with often-used functionality.

- It provides algorithms and functors to work with these classes.
- It also formulates interfaces, which must be provided by other collections of classes in order to be used as STL classes, or can be used to write algorithms that work with all STL-like container classes.
- The STL is a new level of abstraction, which frees the programmer from the necessity to write frequently used constructs such as dynamic arrays, lists, binary trees, search algorithms, and so on him/herself.
- STL algorithms are heavily optimized, i.e. if there is an STL algorithm for a problem, one should have a very good reason not to use it.
- Unfortunately, the STL is not self-explanatory.

# STL Components

The main components of the STL are:

**Containers** are used to manage a particular type of object. The various containers have different properties and related advantages and disadvantages.

**Iterators** make it possible to iterate over the contents of a container. They provide a uniform interface for each STL compliant container, regardless of its internal structure.

**Algorithms** work with the elements of a container. They use iterators and therefore must only be written once for an arbitrary number of STL-compliant containers.

**Functors** can be used in STL algorithms to define local operations / comparisons that should be applied to container elements. They can be extended by user-written alternatives.

At first sight, the structure of the STL partially contradicts the original idea of object-oriented programming that algorithms and data belong together.

# Containers

STL container classes, or short containers, manage a collection of elements of the same type. Depending on the type of container, the STL gives assurances on the execution speed of certain operations.

There are two fundamentally different types of container:

**Sequences** are ordered sets of elements with freely selectable arrangement. Each element has its place, which depends on the program execution and not on the value of the element.

**Associative Containers** are ordered sets sorted according to a certain sorting criterion in which the position of an element depends only on its value.

# List of Available Containers

Sequences:

- `array` (C++11), fixed-size array
- `deque`, double ended queue
- `forward_list` (C++11), linked list
- `list`, doubly linked list
- `vector`, variable-size array

Sequence Adaptors:

- `stack`, LIFO (last in, first out)
- `queue`, FIFO (first in, first out)
- `priority_queue`, heap (highest priority out)

Sequence adaptors can use any sequence internally, if the interface matches (compare our `Stack` implementation).

# List of Available Containers

Associative Containers:

- `set`, mathematical set
- `multiset`, entries may appear multiple times
- `map`, mathematical mapping, dictionary
- `multimap`, as above

Unordered Associative Containers (C++11):

- `unordered_set`
- `unordered_multiset`
- `unordered_map`
- `unordered_multimap`

Unordered containers sort elements using an internal hash function instead of element comparison.

# List of Available Containers

Apart from the above containers, there are several container-like structures that don't fulfill the STL container interface (with some of them not being part of the STL), but are often used instead of or in conjunction with STL containers:

- `pair`, heterogeneous collection of two elements
- `tuple` (C++11), heterogeneous collection
- `valarray`, `vector` with math operations
- `bitset`, specialization of `array` for bool

# Vector

STL sequence containers are class templates. There are two template arguments, the type of objects to be stored and a so-called allocator that can be used to change the memory management (this is useful for example when you create many small objects and don't want to pay the operating system overhead every time). The second parameter has a default value where `new()` and `delete()` are used.

**Vector** is a field of variable size.

- Adding and removing elements at the end of a `vector` is fast, i.e. complexity in $O(1)$.
- The element can be accessed directly via an index (random access).
- The memory used by a `vector` is guaranteed to be contiguous (unique among STL containers).
- The memory reserved by a `vector` never shrinks (!).

Abbildung: Structure of a `vector`

# Amortized Complexity

- Typically, adding elements at the end of a `vector` is in $O(1)$.
- In individual cases, however, it may take much longer, especially if the allocated storage is no longer sufficient. Then new storage must be allocated, and often data has to be copied over. This is an $O(N)$ process.
- However, the standard library reserves memory blocks of increasing size for a growing vector. The overhead depends on the length of the vector. This optimizes for speed at the expense of memory usage.
- The $O(N)$ case therefore occurs very rarely. This is called "amortized complexity".
- If it is already known that a certain amount of elements is needed, then one can reserve space with the method `reserve(size_t size)`. This doesn't change the current size of the vector, it only reserves the right amount of memory.

# Example: STL Vector

```
#include <iostream>
#include <vector>
#include <string>

int main()
{
    std::vector<double> a(7);
    std::cout << a.size() << std::endl;
    for (int i = 0; i < 7; ++i)
        a[i] = i * 0.1;
    double d = 4 * a[2];
    std::vector<double> c(a);
    std::cout << a.back() << " " << c.back() << std::endl;
    std::vector<std::string> b;
    b.resize(3);
    for (int i = 2; i >= 0; --i)
        std::cin >> b[i];
    b.resize(4);
    b[3] = "foo";
    b.push_back("bar");
    for (int i = 0; i < b.size(); ++i)
        std::cout << b[i] << std::endl;
}
```

# Deque

Deque is a “double-ended” queue, it is also a field of dynamic size, but:

- The addition and removal of elements is quick also at the beginning of `deque`, that is $O(1)$, in addition to its end.
- Element access can again be achieved using an index, but the index of an element may change when elements are added to the beginning of the container.
- Internally, `deque`s can be implemented using e.g. arrays of arrays, but this is implementation-dependent.

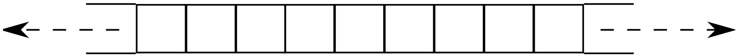


Abbildung: Structure of a `deque`

# List

List is a doubly linked list of elements.

- There is no direct access to list elements.
- To reach the 10th element, one must start at the beginning of the `list` and traverse the first nine elements, access to a specific element is therefore $O(N)$.
- Adding and removing elements is fast in any location in the `list`, i.e. $O(1)$.

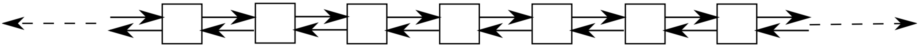


Abbildung: Structure of a `list`

# Example: STL List

```cpp
#include <iostream>
#include <list>
#include <string>

int main()
{
    std::list<double> vals;
    for (int i = 0; i < 7; ++i)
      vals.push_back(i * 0.1);
    vals.push_front(-1);
    std::list<double> copy(vals);
    std::cout << vals.back() << " " << copy.back() << std::endl;
    std::cout << vals.front() << " " << copy.front() << std::endl;
    for (int i = 0; i < vals.size(); ++i)
    {
      std::cout << i << ": " << vals.front() << " " << vals.size() << std::endl;
      vals.pop_front();
    }
    std::cout << std::endl;
    for (int i = 0; i < copy.size(); ++i)
    {
      std::cout << i << ": " << copy.back() << " " << copy.size() << std::endl;
      copy.pop_back();
    }
}
```

# C++11: Array

**Array** is a C++11 replacement for the classical C arrays, i.e. a field of fixed size.

- `array` has two template parameters, the type of the stored objects and the number of elements of the container
- Adding and removing elements isn't possible.
- Element access can be achieved directly via index.
- In contrast to C arrays, an STL `array` knows its own size and can be used like the other STL containers.

Abbildung: Structure of an `array`

# Example: STL Array

```cpp
#include <iostream>
#include <array>
#include <string>

int main()
{
    std::array<double,7> a;
    std::cout << a.size() << std::endl;
    for (int i = 0; i < 7; ++i)
        a[i] = i * 0.1;
    double d = 4 * a[2];
    std::array<double,7> c(a);
    std::cout << a.back() << " " << c.back() << std::endl;
    std::array<std::string,4> b;
    for (int i = 2;i >= 0; --i)
        std::cin >> b[i];
    b[3] = "foo";
    for (int i = 0; i < b.size(); ++i)
        std::cout << b[i] << std::endl;
}
```

# Set/Multiset

- The containers `set` and `multiset` are sorted sets of elements. Internally, these elements are stored in a tree structure.
- 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 (the predicate `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, called concepts.
- 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<`).
- Concepts (constraint sets) like the above can help in writing templates, and have been considered for inclusion in the standard (not just in the context of the STL). However, they haven't been included so far and remain an experimental feature / a way to talk about template design.

# 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 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 value_type. |
| 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 storing 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. number of elements, return type is `size_type`. |
| `max_size()` | Returns the maximum size allowed at the moment (capacity), return type is `size_type`. |
| `empty()` | True if the container is empty. |
| `swap(b)` | Swaps elements with container `b`. |

# Specializations of the Container Concept

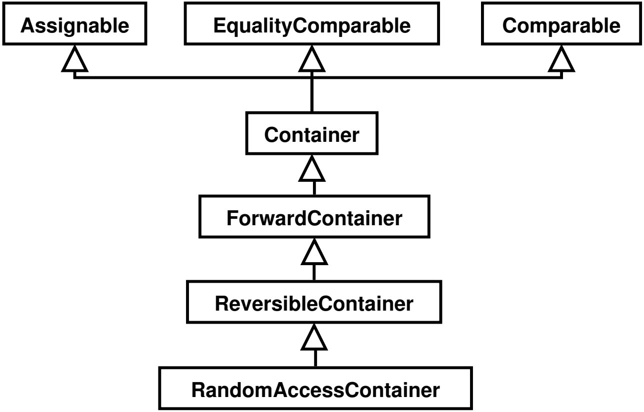



Abbildung: Container concepts

# ForwardContainer

`ForwardContainer` is a specialization of the `Container` concept:

- There is an `iterator` with which one can pass through the container in forward direction (`ForwardIterator`).
- Main addition to vanilla `Container`: elements are arranged in definite order.
- This allows for:
    - element-by-element equality (requires `EqualityComparable` elements)
    - lexicographical ordering (requires `Comparable` elements)

The `Container` concept itself does *not* require an iterator to always produce the same sequence of elements. This is an addition, and formalized in the concept `ForwardContainer`.

Other restrictions and constraints are introduced through similar definitions.

# ReversibleContainer

Concept `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

Concept `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

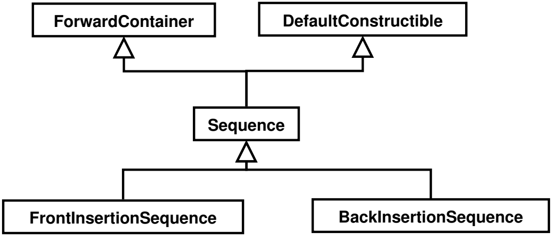



Abbildung: Sequence concepts

# 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 pointing 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 the iterator `p` points to 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())`. |

# C++11: `emplace`

Creating an element and then inserting it creates a copy of the stored element. This is somewhat eleviated with the introduction of C++11 move semantics, but copies are still necessary if the element has to be stored in a specific location (contiguous memory layout of `vector`).

C++11 introduces `emplace` as a substitute for `insert`:

| | |
|---|---|
| `emplace(p,args)` | Inserts an element `t`, constructed using `args`, in front of the one `p` points to, and returns an iterator pointing to the emplaced element. |

The same holds for `emplace_back(args)` (instead of `push_back`) and `emplace_front(args)` (instead of `push_front`) for containers that have that functionality.

Emplacement is possible whenever insertion is possible, and will not be repeated in the following container descriptions.

# Complexity Guarantees for Sequences

The following complexity guarantees are given for sequence containers of the STL:

- The constructors, `X(n,t)` `X(n)` and `X(i,j)` have linear complexity.
- Inserting elements with `insert(p,n,t)` and `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(t)`, 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(t)`, and `pop_front()` have amortized constant complexity.

## Implementations

- `std::list`
- `std::deque`

# STL Sequence Containers

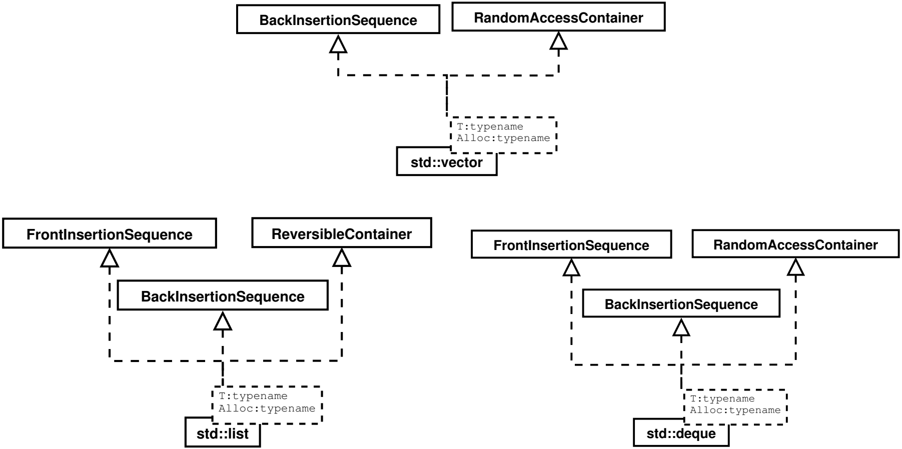


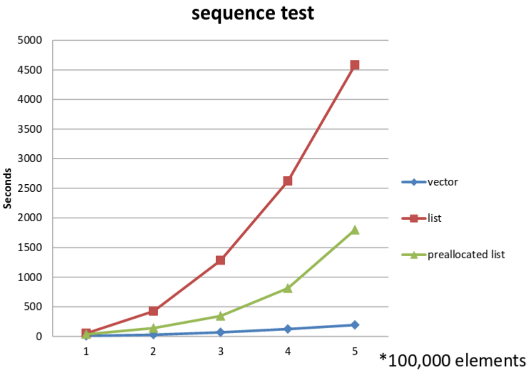

Abbildung: STL sequence containers

# Sequence Test: Vector vs. List

Task by J. Bentley and B. Stroustrup (Element Shuffle):

- For a fixed number $N$, generate $N$ random integers and insert them into a sorted sequence.
  Example:
  - 5
  - 1, 5
  - 1, 4, 5
  - 1, 2, 4, 5
- Remove elements at random while keeping the sequence sorted.
  Example:
  - 1, 2, 4, 5
  - 1, 4, 5
  - 1, 4
  - 4
- For which $N$ should a `list` be used, and in which cases a `vector`?

# Sequence Test: Vector vs. List




Source: B. Stroustrup

- Despite random insertion / deletion, `vector` is faster by an order of magnitude
- Linear search for both containers, despite bisection being available for `vector` (!)
- Search completely dominates move required by `vector`
- Non-optimized `list` performs one allocation / deallocation per element (!)

# Associative Containers

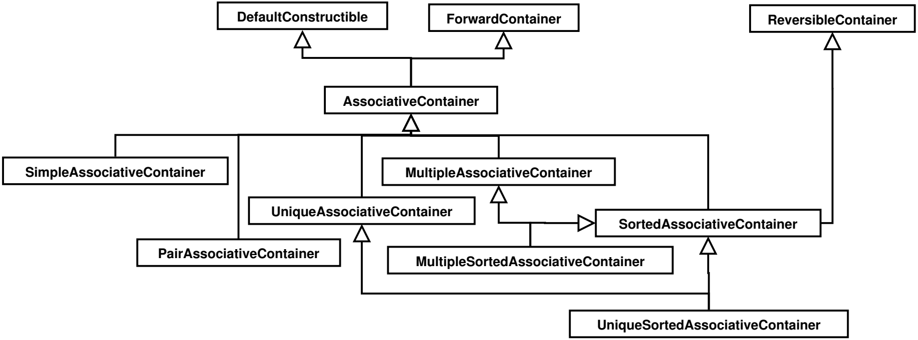



Abbildung: Associative container concepts

# AssociativeContainer

An `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 key `k`. |
| `erase(p)` | Deletes the element that 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 key `k` or `end()` if this key doesn't exist. |
| `count(k)` | Returns the number of elements with key `k`. |
| `equal_range(k)` | Returns a pair `p` of iterators so that [`p.first`,`p.second`) consists of all elements that have 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

## Assurances

value_compare: if `t1` and `t2` have keys `k1` and `k2`, then it is guaranteed that `value_compare()(t1,t2) == key_compare()(k1,k2)` is `true`.

Increasing order of the elements is guaranteed.

## Complexity Guarantees

- `key_compare()(k1,k2)` and `value_compare()(t1,t2)` have constant complexity.
- `lower_bound(k)` and `upper_bound(k)` are logarithmic.

# 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

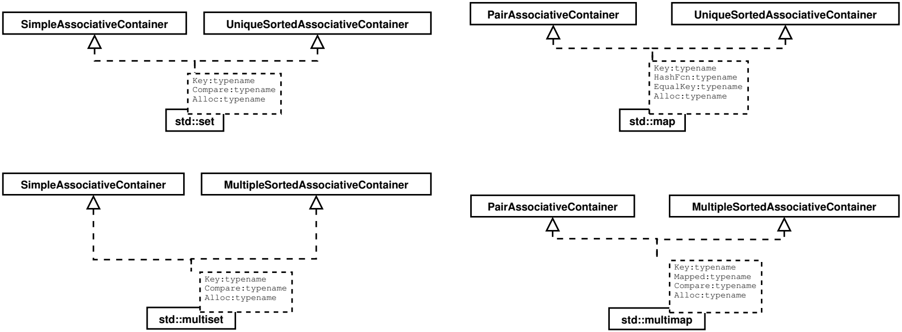



Abbildung: Associative container classes

# 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`.

These are only general recommendations, the "correct" choice depends on specifics! If in doubt, use `vector` if at all possible.