

# Übersicht über die C++-Standard-Library (STL)

Quelle: <http://www.lrz.de/~ebner/C++>,

Folien und Skript der Universitäten München und Stuttgart

## Container-Klassen:

<code>vector&lt;T&gt;</code>	dynamisch wachsendes lineares "Feld"
<code>list&lt;T&gt;</code>	doppelt verkettete Liste, schnelles Einfügen und Entfernen, ohne dass Elemente umkopiert werden müssen
<code>deque&lt;T&gt;</code>	double ended queue: Warteschlangensimulation
<code>set&lt;T&gt;</code>	beliebige Mengen
<code>multiset&lt;T&gt;</code>	wie set, aber Datenelemente können mehrfach vorhanden sein
<code>map&lt;T&gt;</code>	speichert Schlüssel-Wert-Paare. Der Schlüssel muss eindeutig sein und wird nur einfach vorgehalten.
<code>multimap&lt;T&gt;</code>	Jeder Schlüssel kann mit mehreren Werten assoziiert sein (Person mit mehreren Telefonnummern)

```
#include <vector>
void f(){
    // ...
    vector<double> v;
    // iterator ist typedef innerhalb der vector Klasse, also:
    vector<double>::iterator i;

    for ( i = v.begin(); i != v.end(); ++i ){
        std::cout << *i << "\n";
    }
}
```

- Alle Container verwenden **Kopien von Objekten** (copy by value kann teuer werden!).
- Evtl. muss man deshalb **Pointer** auf Objekte in Container eintragen.
- Effiziente Implementierung, aber **nicht sicher** (keine Bereichsprüfung)

## Elementfunktionen, die jeder Container zur Verfügung stellt:

```
CONTAINER<TYPE> m, m2;
TYPE element;
CONTAINER<TYPE>::iterator pos, pos1, pos2;
CONTAINER<TYPE>::const_iterator cpos;

m.size(); // Zahl der gespeicherten Elemente
m.empty(); // bool (true falls leer)
m.swap(m2); // Inhalte von m und m2 vertauschen (schnell)
m.begin(); // Iterator auf erstes Element
m.end(); // Iterator jenseits des letzten Elements
m.rbegin(); // Iterator fuer Rueckwaertsiterator
m.rend();
m.push_front(); // als erstes Element einfügen
m.push_back(); // als letztes Element einfügen
m.insert(pos, element);
m.erase(pos);
m.erase(pos1, pos2);
```

```
m.clear(); // = m.erase(m.begin(),m.end());  
m.resize(size_type anz, TYPE wert); // zusätzliche Elemente
```

## Algorithmen:

### Design:

weitgehend "orthogonal", d.h. die Algorithmen sind zum größten Teil mit allen Containern verwendbar.

### Beispiel:

STL-Algorithmen wenden meist eine Funktion (hier `print_me`), die auf einzelnen Elementen eines Containers an:

```
#include <vector>
#include <algorithm>
void f(){
    std::vector<int> v;
    // ...
    std::for_each( v.begin(), v.end(), print_me );
}
```

### Nicht-modifizierende Algorithmen, z.B.

```
typedef std::vector<int> my_container;

my_container          collect;
my_container::iterator it;

it = std::find( collect.begin()+4, collect.end(), 5 );
if ( it != collect.end() ) // found it
    // ...
```

### und modifizierende Algorithmen, z.B.

```
typedef std::vector<int> my_container;
// ...
std::sort( collect.begin(), collect.end() );
```

Zugriff über **Iteratoren** = verallgemeinerte Pointer

- Entkopplung von Containern und Algorithmen (Algorithmus passt unverändert auf viele Container-Typen)
- Elemente des Containers müssen Mindestsatz von Operatoren implementieren (++, \*)
- Iteratoren des Containers: `v.begin()` und `v.end()`

<b>Iterator-Typ</b>	
Forward Iterator	++, * (deref), ->, ==, !=

Backward Iterator	--, * (deref), ->, ==, !=
Bidirectional Iterator	++, --, * (deref), ->, ==, !=
Random Access Iterator	--, ++, [], * (deref), ->, ==, <, >, !=

... umfangreich (trotzdem noch nicht vollständig!)

→ **modifizierend betrifft hierbei den Container, nicht die Elemente**

<b>nichtmodifizierende Algorithmen:</b>	
for_each	lesender Zugriff auf Elemente durch Fkt./Fkt.Obj.
find(), find_if()	Suche eines bestimmten Elementes
search()	sucht erste Teilfolge bestimmter Werte
find_end()	sucht letzte Teilfolge bestimmter Werte
find_first_of()	sucht eines von mehreren möglichen Elementen
(min/max)_element()	kleinstes/größtes Element
count, count_if	zählt passende Elemente
equal()	testet zwei Bereiche auf Gleichheit
<b>modifizierende Algorithmen:</b>	
copy()	kopiert Bereich
transform()	wie for_each für ggfs. zwei Bereiche, erlaubt Modifikation
fill()	gibt Bereich einen neuen Wert
generate()	wie fill(), aber mit Fkt./Fkt.Obj.
<b>löschende Algorithmen:</b>	
remove(), remove_if()	löscht Bereich
remove_copy()	kopiert Bereich und löscht Elemente
unique()	löscht aufeinanderfolgende Duplikate

<b>spezielle Algorithmen für sortierte Bereiche:</b>	
<code>(lower/upper)_bound()</code>	eines sortierten Bereichs
<code>binary_search()</code>	schnelles Suchen in sortiertem Bereich
<code>merge()</code>	Zusammenfassen zweier sortierter Bereiche
<code>set_union()</code>	Vereinigungsmenge sortierter Bereiche
<code>set_intersection()</code>	Schnittmenge
<code>set_difference()</code>	Differenzmenge
<code>set_union()</code>	Vereinigungsmenge
<b>mutierende Algorithmen:</b>	
<code>reverse()</code>	Umkehr der Reihenfolge
<code>(next/prev)_permutation()</code>	Permutation
<code>random_shuffle()</code>	Durcheinanderwürfeln
<b>modifizierende Algorithmen:</b>	
<code>sort()</code>	Sortieren
<code>partial_sort()</code>	Sortieren der ersten n Elemente
<code>stable_sort()</code>	wie <code>sort()</code> , ändert aber nicht die Reihenfolge von Elementen mit gleichem Wert
<code>make_heap()</code>	Heap aus Bereich erzeugen
<code>push_heap()</code>	integriert ein Element in einem Heap
<code>pop_heap()</code>	löst ein Element aus einem Heap
<code>sort_heap()</code>	sortiert einen Heap komplett