5 Die Standard Template Library *(STL)*

5.1 Container
5.2 Iteratoren mit Algorithmen (*)
5.3 Funktionsobjekte
5.4 Algorithmen

(*) Auch in dieser Datei

Lehrbuch: 6.4
Kompendium, 3. Auflage: 8.12, 11.5
Kompendium, 4. Auflage: 11.5

Algorithmus for_each

```
#include <algorithm>
using namespace std;

void MachWas(int i) { cout << i*i; }
int array[100];
 for_each(&array[0], & array[100],
      MachWas);

 sort(&array[10], & array[20]);
```

```
template <typename Iter, typename Func>
void for_each(Iter  anf,  Iter end, Func f){
    while (anf!= end) {
        f(*anf); ++anf;
     }
}
```

Beispiele für Algorithmen

```
vector<int> vec;
. . .

vector<int>::iterator pos;
pos = min_element(vec.begin(), vec.end());
cout << * max_element(vec.begin(), vec.end());
sort(vec.begin(), vec.end());
. . .
```

Beispiele für Algorithmen

```
vector<int> vec;
. . .
vector<int>::iterator pos;
pos = min_element(vec.begin(), vec.end());
cout << * max_element(vec.begin(), vec.end());
sort(vec.begin(), vec.end());
. . .
// Reihenfolge vom zweiten bis vorletzten Element umkehren
reverse(vec.begin()+1, vec.end()-1);
. . .
// Ab dem zweiten Element nach 5 suchen
pos = find(vec.begin()+1, vec.end(), 5);
```

Clicker

```
list<int>  list1;
for (int i=1; i<13; ++i) {
   list1. push_back( i ); }
cout << * max_element(list1.begin(), list1.end());
cout << * max_element(list1.begin()++, list1.end());
cout << * max_element(++list1.begin(), list1.end());
```

Ausgabe?

1. 12 12 12
2. 12 12 11
3. 13 12 12
4. 12 11 12

Clicker

```
list<int>  list1;
for (int i=1; i<13; ++i) {
   list1. push_back( i ); }
cout << * max_element(list1.begin(), list1.end());
cout << * max_element(list1.begin()++, list1.end());
cout << * max_element(++list1.begin(), list1.end());
```

Ausgabe?

1. 12 12 12
2. 12 12 11
3. 13 12 12
4. 12 11 12

Ergebnis:
_-__ 12 12 12 ___12 12 11 __ 13 12 12 __ 12 11 12

Clicker

```
list<int>  list1;
for (int i=1; i<13; ++i) {
   list1. push_front( i );}
cout << * max_element(list1.begin(), list1.end());
cout << * max_element(list1.begin()++, list1.end());
cout << * max_element(++ list1.begin(), list1.end());
```

Ausgabe?

1. 12 12 11
2. 1 1 2
3. 13 12 12
4. 12 11 12

Ausprobieren am Code in Lieferant-02.zip in der auskommentierten Funktion maxElementMitLists() in Aufgaben.cxx

Sie sind dran

Wenn Sie fertig sind, Schauen Sie sich die weiteren Folien hier schon mal an und setzen bei der nächsten Aufgabe fort.

Code auf dieser Seite verstanden?

1. Bin fertig
2. Sie können weitermachen, ich werde es hier und heute nicht implementieren.

Clicker

```
list<int>  list1;
for (int i=1; i<13; ++i) {
   list1. push_front( i );}
cout << * max_element(list1.begin(), list1.end());
cout << * max_element(list1.begin()++, list1.end());
cout << * max_element(++ list1.begin(), list1.end());
```

Ausgabe?

1. 12 12 11
2. 1 1 2
3. 13 12 12
4. 12 11 12

Ergebnis:

_-__ 12 12 11 ___1 1 2 __ 13 12 12 __ 12 11 12

Algorithmen -- Fortsetzung

Beispiel:

```
// vec sei mit Zahlen von 1 bis 20 gefüllt
vector<int>::iterator pos5   = find(vec.begin(), vec.end(), 5);
vector<int>::iterator pos13 = find(vec.begin(), vec.end(), 13);

// 12 wird ausgegeben, da Intervall [pos5, pos13 [        bearbeitet wird
cout << *max_element(pos5, pos13);
// 13 wird ausgegeben.
cout << *max_element(pos5, ++pos13);
```

Der Anwender eines Algorithmus muss selbst darauf achten, dass das übergebene Ende vom Anfang aus erreichbar ist, d.h. die Iteratoren müssen zumindest zum gleichen Container gehören.

Die Bereiche der Algorithmen werden immer als halboffenene Intervalle ausgewertet: *[anfang, ende)* bedeutet *anfang...ende-1*.

Clicker: Algorithmen auch für Arrays

```
int arr[20];
for (int i=0; i<20; ++i) {
   arr[i]   = i * i;  }
cout << * max_element(&arr[4],   & arr[20]);
cout << * find(&arr[4], & arr[10], 64);
cout << * find(&arr[4], & arr[10], 144);
```

Ausgabe?

1. 19 8 10
2. 361 64 100
3. 361 64 Fehler
4. 361 100 11

Clicker: Algorithmen auch für Arrays

```
int arr[20];
for (int i=0; i<20; ++i) {
   arr[i]   = i * i;  }
cout << * max_element(&arr[4],   & arr[20]);
cout << * find(&arr[4], & arr[10], 64);
cout << * find(&arr[4], & arr[10], 144);
```

Ausgabe?

1. 19 8 10
2. 361 64 100
3. 361 64 Fehler
4. 361 100 11

Ausprobieren am Code in Lieferant-02.zip in der Funktion
maxElementMitLists() in Aufgaben.cxx aus.
Duplizieren die Funktion.
Oder selber „schnell“ den Code eingeben und wundern?

Clicker: Algorithmen auch für Arrays

```
int arr[20];
for (int i=0; i<20; ++i) {
   arr[i]   = i * i;  }
cout << * max_element(&arr[4],   & arr[20]);
cout << * find(&arr[4], & arr[10], 64);
cout << * find(&arr[4], & arr[10], 144);
```

Ausgabe?

1. 19 8 10
2. 361 64 100
3. 361 64 Fehler
4. 361 100 11

Ergebnis:

__ 19 8 10 _-_ 361 64 100 __ 361 64 Fehler _361 100 11

Clicker: Algorithmen auch für STL-Arrays

```
array<int, 20> arr;  // int arr[20];
for (int i=0; i<20; ++i) {
   arr[i]   = i * i; }
cout << * max_element(&arr[4],   & arr[20]);
cout << * find(&arr[4], & arr[10], 64);
cout << * find(&arr[4], & arr[10], 144);
```

Ausgabe?

1. 19 8 10
2. 361 64 100
3. 361 64 Fehler
4. Fehler

Ergebnis:
__ 19 8 10 _-_ 361 64 100 __ 361 64 Fehler _Fehler

Im Release-Modus kann man Glück/Pech haben, aber &arr[20] ist nur für wirkliche C-Arrays, aber nicht für STL-Array definiert

array-Klasse versus C-Array

```
void STL_arrayIter() {
   array<int, 20> arr;  // int arr[20];
   for (int i = 0; i < 20; ++i) {
      arr[i] = i * i;
   }
   // &arr[20] nicht definiert, geht im Release aber nicht im Debug
   cout << *max_element( ++(++(++(++arr.begin()))), arr.end()) << " ";
   cout << *max_element(arr.begin()+4, arr.end()-4) << " ";
   cout << *find(&arr[4], &arr[10], 64) << " ";
   cout << *find(&arr[4], &arr[10], 144) << "\n";
}
```

Ausgabe wäre: 361 225 64 100

Clicker: Algorithmen auch für STL-Vector

```
vector<int> arr(21);
for (int i=0; i<20; ++i) {
   arr[i]   = i * i; }
cout << * max_element(&arr[4],   & arr[20]);
cout << * find(&arr[4], & arr[10], 64);
cout << * find(&arr[4], & arr[10], 144);
```

Ausgabe?

1. 19 8 10
2. 361 64 100
3. 361 64 Fehler
4. 361 100 11

Clicker: Algorithmen auch für STL-Vector

```
vector<int> arr(21);
for (int i=0; i<20; ++i) {
   arr[i]   = i * i; }
cout << * max_element(&arr[4],   & arr[20]);
cout << * find(&arr[4], & arr[10], 64);
cout << * find(&arr[4], & arr[10], 144);
```

Ausgabe?

1. 19 8 10
2. 361 64 100
3. 361 64 Fehler
4. 361 100 11

Ergebnis:

__ 19 8 10 _-_ 361 64 100 __ 361 64 Fehler _361 100 11

Clicker: Algorithmen auch für STL-Vector mit push_back

```
vector<int> arr;
for (int i=0; i<20; ++i) {
   arr.push_back(i * i); }
// &arr[20] nicht definiert
cout << * max_element(&arr[4],   & arr[19]);
cout << * find(&arr[4], & arr[10], 64);
cout << * find(&arr[4], & arr[10], 144);
```

Ausgabe?

1. 19 8 10
2. 324 64 100
3. 324 64 Fehler
4. 361 100 11

Ergebnis:

__ 19 8 10 _-_ 324 64 100 __ 361 64 Fehler _361 100 11

Algorithmen – Beispiel: Definition von max_element

```
template <typename ForwardIter>
ForwardIter max_element(ForwardIter first, ForwardIter last) {
   if (first == last) return first;

   ForwardIter result = first;
   while (++first != last) {
      if (*result < *first) {
         result = first;
      }
   }
   return result;
}
```

Trauen Sie sich zu, min_element selbst als Template zu implementieren

1. Versteh nur Bahnhof
2. Ja, mit Blick auf die Folie
3. Sollte ohne Hilfe (evtl. nicht so schön) klappen

Algorithmen – Beispiel: Definition von max_element

```
template <typename ForwardIter>
ForwardIter max_element(ForwardIter first, ForwardIter last) {
   if (first == last) return first;

   ForwardIter result = first;
   while (++first != last) {
      if (*result < *first) {
         result = first;
      }
   }
   return result;
}
```

Clicker-Auswertung

Trauen Sie sich auch zu find selbst als Template zu implementieren
1. Versteh nur Bahnhof
2. Ja, mit Blick auf die Folie zu max_element
3. Sollte ohne Hilfe (evtl. nicht so schön) klappen

Ergebnis:
___ 1 __ 2 __ 3

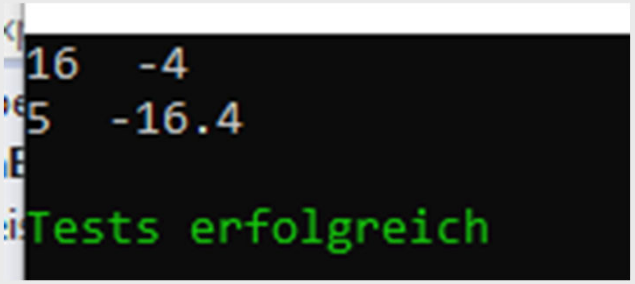
Implementieren Sie MeinMinElement

```
int arr[20];
double arrd[20];
for (int i=0; i<20; ++i) {
   arr[i]   = i * i;
   arrd[i] = i;
}

cout << * MeinMinElement(&arr[4],   & arr[20]);
arr[14] = -4.3; // gibt Warnung
cout <<" " <<  * MeinMinElement(&arr[0],   & arr[20]);

cout << * MeinMinElement(&arrd[5],   & arrd[20]);
arrd[13] = -16.4;
cout <<" " <<  * MeinMinElement(&arrd[0],   & arrd[20]);
```

Gewünschte Ausgabe



Implementieren Sie MeinMinElement

```
int arr[20];
double arrd[20];
for (int i=0; i<20; ++i) {
   arr[i]  = i * i;
   arrd[i] = i;
}
// Suche minimale Element ungleich 16
cout << * MeinMinElementUngleich(&arr[4],   & arr[20], 16);
arr[14] = -4.3;
cout <<" " <<  * MeinMinElementUngleich(&arr[0],   & arr[20],16);

cout << * MeinMinElementUngleich(&arrd[5],   & arrd[20],16);
arrd[13] = -16.4;
cout <<" " <<  * MeinMinElementUngleich(&arrd[0],   & arrd[20], 111);
```

Gewünschte Ausgabe

```
25   -4
5   -16.4
```

Implementieren Sie MeinMinElement

```
int arr[20];
double arrd[20];
for (int i=0; i<20; ++i) {
   arr[i]   = 4;
   arrd[i] = 15;
}

cout << *MeinMinElementUngleich(&arr[4], &arr[19], 16) << " ";
arr[19] = 40000;
cout << *MeinMinElementUngleich(&arr[0], &arr[19], 4) << "\n";

cout << *MeinMinElementUngleich(&arrd[5], &arrd[19], 16) << " ";
arrd[19] = 55555;
cout << *MeinMinElementUngleich(&arrd[0], &arrd[19], 15) << "\n";
```

Gewünschte Ausgabe

```
4 40000
15 55555
```

Ihr Stand

1. MeinMinElement geht
2. MeinMInElementUngleich geht
3. Brauche Hilfe

Verwaltung der Lieferanten einer Komponente in einem Container

Eine Komponente Hauptspeicher wird erzeugt.
Zwei verschiedene Lieferanten werden dem Container hinzugefügt und es wird überprüft, ob einer der Lieferanten ein Lieferant des Hauptspeichers ist.
Anschließend wird überprüft, ob ein dritter Lieferant kein Lieferant des Hauptspeichers ist.

Wiederholung

Verwaltung der Lieferanten einer Komponente in einem Container (2)

In einem zweiten Test entfernen wir einen Lieferanten vom Hauptspeicher und überprüfen, ob der Lieferant nun auch wirklich kein Lieferant des Hauptspeichers mehr ist.

Verwaltung der Lieferanten einer Komponente in einem Container

```
bool containerTest() {
    //Eine Komponente Hauptspeicher wird erzeugt.
Komponente hauptspeicher;
   // 2 verschiedene Lieferanten werden dem Container hinzugefügt
Lieferant aa("AA");      hauptspeicher.addLieferant(aa);
Lieferant ab("AB");      hauptspeicher.addLieferant(ab);
   // und es wird überprüft, ob AA ein Lieferant des Hauptspeichers ist.
if (hauptspeicher.hatLieferant(aa) == false) { return false;}
   // Es wird überprüft, ob ein 3. kein Lieferant des Hauptspeichers ist.
Lieferant xx("XX");
if (hauptspeicher.hatLieferant(xx) == true) { return false;}

return true;
}
```

Verwaltung der Lieferanten einer Komponente in einem Container (2)

```
bool containerTest2 () {
Komponente hauptspeicher;
Lieferant aa("AA");      hauptspeicher.addLieferant(aa);
Lieferant ab("AB");      hauptspeicher.addLieferant(ab);

/* In einem zweiten Test entfernen einen Lieferanten vom Hauptspeicher
        und prüfen, dass der Lieferant nun auch wirklich kein Lieferant des
        Hauptspeichers mehr ist.*/
hauptspeicher.entferneLieferant(aa);
if (hauptspeicher.hatLieferant(aa) == true) { return false;}
if (hauptspeicher.hatLieferant(ab) == false) { return false;}

return true;
}
```

Klasse Lieferant seit C++11

```
class Lieferant {
   public:
   Lieferant(string n): name(n) {};
   string getName() const {return name;}
   Lieferant(const Lieferant&) = delete;
   Lieferant& operator=(const Lieferant&) = default;
private:
    string name;
};
```

```
bool operator==(const Lieferant& l1, const Lieferant& l2) {
   return l1.getName() == l2.getName();
}
```

Klasse Komponente

```
class Komponente {
public:
   Komponente() {};
   void addLieferant(const Lieferant& );
   void entferneLieferant(const Lieferant& );
   bool hatLieferant(const Lieferant& li) const;
private:
    Komponente(const Komponente&) = delete; /* verbotener Aufruf */
    Komponente& operator=(const Komponente&) = delete; /* verbotener Aufruf */
   vector<Lieferant> container;
};
```

```
void Komponente::addLieferant(const Lieferant& li) { /*…  */ }
void Komponente:: entferneLieferant(const Lieferant& li ) { /*…  */ }
bool Komponente:: hatLieferant(const Lieferant& li) const { /*…  */ }
```

Algorithmus remove -- Ende der Wiederholung ---

```
#include <iostream>  #include <list> #include<algorithm>  using namespace std;
int main() {
   list<int>  cont;     for (int i=1; i<6; ++i) { cont.push_back(i); }

   cout << "Liste vor remove :"; // Ausgabe ist nun 1 2 3 4 5
   copy(cont.begin(), cont.end(), ostream_iterator<int>(cout, " "));

   // 3 mit remove entfernen
   list<int>::iterator new_end = remove(cont.begin(), cont.end(), 3);

   // Ausgabe ist nun 1 2 4 5 5
   copy(cont.begin(), cont.end(), ostream_iterator<int>(cout, " "));
}
```

Algorithmus versus Methode

```
set<int> s = {11, 2, 3, 5, 7, 8};
vector<int> v = {11, 2, 3, 5, 7, 8};
list<int> li = {11, 2, 3, 5, 7, 8};


…
auto iter = find(s.begin(), s.end(), 32);  // Aufwand O(???)
auto iter = find(v.begin(), v.end(), 32);  // Aufwand O(???)
auto iter = find(li.begin(), li.end(), 32);  // Aufwand O(???)
```

Algorithmus *find*

```
template <class InputIter, class Tp>
inline InputIter find(InputIter first, InputIter last, const Tp& val) {
   while (first != last && *first != val) {
      ++first;
   }
   return first;
}
```

Es wurde der generische „find“ Algorithmus verwendet.
Daher ist die Komplexität immer O(N).
Der generische Algorithmus weiß nicht womit er aufgerufen wird und kann daher auch nicht von der Baumstruktur profitieren.

Algorithmus versus Methode

```
set<int> s = {11, 2, 3, 5, 7, 8};
vector<int> v = {11, 2, 3, 5, 7, 8};
list<int> li = {11, 2, 3, 5, 7, 8};


…
auto iter = find(v.begin(), v.end(), 32);  // Aufwand O(N)
auto iter = find(li.begin(), li.end(), 32);  // Aufwand O(N)
auto iter = s.find(32);  // Aufwand O(???)
```

Bei Verwendung der Methode set::find kann von der Baumstruktur Gebraucht gemacht werden.
Aufwand O(log N).

Algorithmus remove (2)

Warum wird ein Element nicht wirklich gelöscht?

Algorithmen können die Container nicht wirklich verkleinern oder vergrößern, weil ihnen der Container selbst nicht übergeben wird, sondern nur Iteratoren, die irgendwo in den Container zeigen, z.B.

```
list<int>::iterator new_end = cont. end();
--new_end;    --new_end;
remove(cont.begin(), new_end, 3);
```

Damit kann remove auf keinen Fall das *end* von *cont* verändern.
Woher soll es das auch kennen.
Dieses können nur Elementfunktionen, z.B.

```
erase(cont::iterator a, cont::iterator e);
```

Alle Elemente im Intervall *[a..e)* werden wirklich gelöscht.

Elementfunktion erase

```
copy(cont.begin(), cont.end(), ostream_iterator<int>(cout, " "));
// Ausgabe sei vorher 1 2 3 4 5

// 3 mit remove entfernen
cont.erase(remove(cont.begin(),  cont.end(), 3), cont.end());

copy(cont.begin(), cont.end(), ostream_iterator<int>(cout, " "));
// Ausgabe ist nun 1 2 4 5
```

Löschen mit find

```
list<int>  cont;
for (int i=1; i<6; ++i) {
   cont.push_back(i);
}
cont. push_back(3);   cont. push_back(3);

copy(cont.begin(), cont.end(), ostream_iterator<int>(cout, " "));
// Ausgabe ist nun 1 2 3 4 5 3 3

list<int>::iterator iter3= find(cont.begin(), cont.end(), 3);
list<int>::iterator iter3p1=iter3;
++iter3p1;

cont.erase(iter3, iter3p1);
copy(cont.begin(), cont.end(), ostream_iterator<int>(cout, " "));
// Ausgabe ist nun Liste nach erase :1 2 4 5 3 3
```

remove ist aber mächtiger

```
list<int>  cont;  //  Container-List fuer int
for (int i=1; i<6; ++i) {  cont.push_back(i);  }
cont. push_front(3);
cont. push_front(3);
copy(cont.begin(), cont.end(), ostream_iterator<int>(cout, " "));
// Ausgabe ist nun 3 3 1 2 3 4 5

list<int>::iterator new_end =  remove(cont.begin(), cont.end(), 3);
copy(cont.begin(), cont.end(), ostream_iterator<int>(cout, " "));
// Ausgabe ist nun 1 2 4 5 3 4 5

cont.erase(new_end, cont.end());
copy(cont.begin(), cont.end(), ostream_iterator<int>(cout, " "));
// Ausgabe ist nun Liste nach erase :1 2 4 5
```

Clicker

2. Ausgabe?
1. 1 2 3 4 5
2. 1 2 4 5 5
3. 1 2 4 5
4. 1 2 4 5 3

```
list<int>  cont;
for (int i=1; i< 6; ++i) {
   cont. push_back( i );
}
   //  Ausgabe ist 1 2 3 4 5
copy(cont.begin(), cont.end(), ostream_iterator<int>(cout, " "));
remove(cont.begin(), cont.end(), 3);
   // 2. Ausgabe???
copy(cont.begin(), cont.end(), ostream_iterator<int>(cout, " "));
```

Clicker

```
list<int>  cont;
for (int i=1; i< 6; ++i) {
   cont. push_back( i );
}
   //  Ausgabe ist 1 2 3 4 5
copy(cont.begin(), cont.end(), ostream_iterator<int>(cout, " "));
remove(cont.begin(), cont.end(), 3);
   // 2. Ausgabe???
copy(cont.begin(), cont.end(), ostream_iterator<int>(cout, " "));
```

2. Ausgabe?

1. 1 2 3 4 5
2. 1 2 4 5 5
3. 1 2 4 5
4. 1 2 4 5 3

Ergebnis:
__ 1 2 3 4 5 _--_ 1 2 4 5 5 __ 1 2 4 5 __ 1 2 4 5 3

Clicker

```
list<int>  cont;
for (int i=1; i< 6; ++i) {
   cont. push_back( i );
}

   //  Ausgabe ist 1 2 3 4 5
copy(cont.begin(), cont.end(), ostream_iterator<int>(cout, " "));
remove(cont.begin(), cont.end(), 3   );
cont.erase(cont.begin(), cont.end());
copy(cont.begin(), cont.end(), ostream_iterator<int>(cout, " "));
```

2. Ausgabe?
1. 1 2 3 4 5
2.
3. 1 2 4 5
4. 1 2 4 5 3

Clicker

```
list<int>  cont;
for (int i=1; i< 6; ++i) {
   cont. push_back( i );
}

   //  Ausgabe ist 1 2 3 4 5
copy(cont.begin(), cont.end(), ostream_iterator<int>(cout, " "));
remove(cont.begin(), cont.end(), 3    );
cont.erase(cont.begin(), cont.end());
copy(cont.begin(), cont.end(), ostream_iterator<int>(cout, " "));
```

2. Ausgabe?
1. 1 2 3 4 5
2.
3. 1 2 4 5
4. 1 2 4 5 3

Ergebnis:
__ 1 2 3 4 5 _--_ keine Ausgabe __ 1 2 4 5 __ 1 2 4 5 3

Clicker

```
list<int>  cont;
for (int i=1; i< 6; ++i) {
   cont. push_back( i );
}

   //  Ausgabe ist 1 2 3 4 5
copy(cont.begin(), cont.end(), ostream_iterator<int>(cout, " "));
cont.erase(remove(cont.begin(), cont.end(), 3),  cont.end() );
copy(cont.begin(), cont.end(), ostream_iterator<int>(cout, " "));
```

2. Ausgabe?

1. 1 2 3 4 5
2. 1 2 4 5 5
3. 1 2 4 5
4. 1 2 4 5 3

Clicker

```
list<int>  cont;
for (int i=1; i< 6; ++i) {
   cont. push_back( i );
}

   //  Ausgabe ist 1 2 3 4 5
copy(cont.begin(), cont.end(), ostream_iterator<int>(cout, " "));
cont.erase(remove(cont.begin(), cont.end(), 3),  cont.end() );
copy(cont.begin(), cont.end(), ostream_iterator<int>(cout, " "));
```

2. Ausgabe?
1. 1 2 3 4 5
2. 1 2 4 5 5
3. 1 2 4 5
4. 1 2 4 5 3

Ergebnis:
__ 1 2 3 4 5 __ 1 2 4 5 5 _--_ 1 2 4 5 __ 1 2 4 5 3

Komponente::entferneLieferant

```
void Komponente:: entferneLieferant(const Lieferant& li ) {
container.erase(
      remove(container.begin(),  container.end(), li),
      container.end());
}
```

```
void Komponente:: entferneLieferant(const Lieferant& li ) {
vector<Lieferant>::iterator iter =
    find(container.begin(). container.end(), li);
    if (iter != container.end()) {
            vector<Lieferant>::iterator hlp = iter;
             ++hlp;
            container.erase(iter, hlp);
   };
}
```

Wenn man es denn selber „zu Fuß“ implementieren will und dann wird nur das erste gelöscht.
Das obere löscht alle Vorkommen von li

Frage

Warum hat die STL nicht so sehr viel mit objektorientierter Programmierung gemein / zu tun?