Programmentwicklung II

Bachelor of Science

Prof. Dr. Rethmann / Prof. Dr. Brandt

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

Sommersemester 2021

Motivation

Häufig genutzte Container und Algorithmen werden in der STL (Standard Template Library) bereitgestellt und müssen von uns nicht mehr implementiert werden.

Problem: Algorithmen hängen von der Art und Weise ab, wie die Daten gespeichert werden:

- binäre Suche im Array (sortiert)
- lineare Suche in verketteter Liste

Um unsere Algorithmen unabhängig von der Art der Speicherung zu machen, lernen wir das Iterator-Konzept kennen.

Container:

- sind eine Sammlung von Objekten (geordnet oder ungeordnet)
- verfügen unter anderem über Methoden zum
  - Einfügen von Objekten: `insert`, `push_back`, `push_front`
  - Löschen von Objekten: `pop_back`, `pop_front`, `erase` und `clear`
  - Suchen von Objekten: `find`
  - Iterieren über die enthaltenen Objekte: `back`, `front`, `begin`, `end`, `rbegin` und `rend`

Beispiele für

- sequentielle Container: `list`, `vector`, `deque`
- assoziative Container: `set`, `multiset`, `map`, `multimap`
- Algorithmen: `sort`, `binary_search`, `for_each`, `count`, `nth_element` und weitere

Eine gute Übersicht zur Standard Template Library finden Sie bspw. unter:
`https://en.cppreference.com/w/Main_Page`
`http://www.cplusplus.com`

Kanonische Klassen: Die Klassen der Container-Elemente sollen Methoden vorgeben, damit die Container auf den Elementen arbeiten können:

- Konstruktor und Destruktor
- Copy-Konstruktor
- Zuweisungsoperator `operator=`
- Gleichheitsoperator `operator==`
- Kleineroperator `operator<`

In der STL werden die anderen Vergleichsoperatoren aus den gegebenen in `<utility>` definiert. Beispiel: `>` ist äquivalent zu „nicht kleiner und nicht gleich“

Container für unterschiedliche Arten von Objekten:

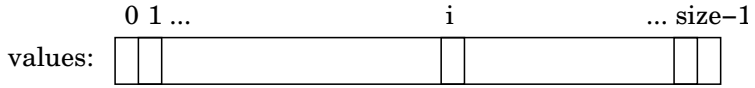
1. Basisklasse für den Container:
   - Für jede neue Objektart, die gespeichert werden soll, wird eine Klasse von der Container-Basisklasse abgeleitet.
   - Nachteil: Hoher Programmieraufwand und sehr ähnliche Lösungen für jede neue Objektart.
2. Basisklasse für die zu speichernden Objekte:
   - Die Container werden für die Basisobjekte formuliert.
   - Neue Objektarten werden durch Vererbung gebildet.
   - Durch dynamische Bindung können auch Objekte des abgeleiteten Typs im Container gespeichert werden.
   - Nachteil: eingeschränkte Typfreiheit
3. Generische Programmierung (Templates):
   - Die Container werden allgemein formuliert.
   - Es werden konkrete Ausprägungen für einzelne Typen von Objekten gebildet.
   - Nachteil: Für jede Ausprägung generiert der Compiler zusätzlichen Objekt-Code und wir können nur Objekte eines Typs und deren Untertypen speichern.

STL verwendet Templates für Container und sehr effiziente Algorithmen!

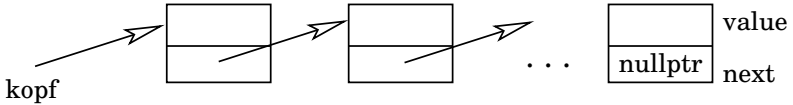
Vektor vs. Liste

Wir können einen Container, in dem Objekte unsortiert gespeichert werden, auf mehrere Arten implementieren: Die Objekte werden

- in einem Array gespeichert



- oder in einer verketteten Liste.



Container mittels unsortiertem Array implementiert:

```
template <typename T>
int MyVector<T>::suchen(const T& x) const {
    for (int i = 0; i < size; i++)
        if (values[i] == x)
            return i;
    return -1;
}
...
int main(void) {
    MyVector<int> v(100);
    ...
    int i = v.suchen(42);
    if (i != -1)
        ... // gefunden
    else ... // nicht gefunden
    ...
```

Container mittels verketteter Liste implementiert:

```
template <typename T>
T * MyList<T>::suchen(const T& x) const {
    for (Elem<T> *p = kopf; p != 0; p = p->next)
        if (p->value == x)
            return &p->value;  // Kopie liefern?
    return 0;
}
...
int main(void) {
    MyList<int> l(100);
    ...
    int *p = l.suchen(42);
    if (p != 0)
        ... // gefunden
    else ... // nicht gefunden
    ...
```

Vektor vs. Liste

Problem:

- Beide Klassen besitzen eine eigene Elementfunktion zum Suchen, die speziell auf die Klasse zugeschnitten ist.
- Bei n Klassen und m Elementfunktionen müssen wir also $n \cdot m$ Elementfunktionen definieren.

Lösung: Standardisierung der Suchfunktion.

- Wie wird der Suchparameter an die Funktion übergeben? (als Referenz, Zeiger, oder Objekt)
- Welchen Rückgabetyp soll die Funktion haben?
- Was wird zurückgegeben, wenn das Gesuchte nicht gefunden wird?

Angleichen der Suchfunktionen des Vektors und der Liste:

```
T * MyVector<T>::suchen(const T& x) const {
    T *p = values;          // start
    while (p != values + size  // nicht am Ende
            && *p != x)        // nicht gefunden
        p++;                // nächstes Element
    return p;
}

T * MyList<T>::suchen(const T& x) const {
    Elem<T> *p = kopf;     // start
    while (p != 0              // nicht am Ende
            && p->value != x)  // nicht gefunden
        p = p->next;       // nächstes Element
    return (p == 0 ? p : &p->value);
}
```

Die `while`-Schleife bricht ab, sobald das Element gefunden wurde oder das „Ende des Containers“ erreicht wurde.

Nichtgefunden bedeutet: Rückgabeposition ist „Ende des Containers“.

Parametrisierte Funktion zum Suchen in `MyVector`:

```
template <typename T>
T * suchen(T *start, T *ende, T x) {
    T *p = start;

    while (p != ende && *p != x)
        p++;

    return p;
}
```

Aufruf der Suchfunktion im Vector-Beispiel:

```
MyVector<int> v(100);
...
int *z = suchen(v.start(), v.ende(), 42);
if (z != v.ende())
    ... // gefunden
else ... // nicht gefunden
```

Die Klasse `MyVector` muss die Funktionen `start` und `ende` bereitstellen! Denn die Funktion `suchen` ist global definiert, nicht als Methode der Klasse `MyVector`, und hat daher keinen Zugriff auf dessen private Attribute zugreifen.

Frage: Warum können wir diese parametrisierte Suchfunktion nicht für unsere Listenklasse nutzen?

Antwort:

- Der Inhaltsoperator funktioniert nicht: In der verketteten Liste müsste es `p->value != x` heißen anstatt `*p != x`.
- Die lokale Variable `start` ist nicht vom Typ `T *` sondern vom Typ `Elem<T> *`.
- Ebenso ist `ende` nicht vom Typ `T *` sondern vom Typ `Elem<T> *`.
- Das nachfolgende Element in einer verketteten Liste ergibt sich nicht aus einem Zeiger auf ein Feldelement +1, sondern der Zeiger auf das nachfolgende Element ist in dem Feldelement selbst enthalten.

Identifiziere die Unterschiede und kapsele diese!

```
template <typename Iterator, typename T>
Iterator suchen(Iterator start, Iterator ende, const T x) {
    while (start != ende && *start != x)
        start++;

    return start;
}
```

Welche Operationen werden beim Iterator benötigt?

Implizite Annahmen über den Iterator:

- Die Parameter `start` und `ende` vom Typ `Iterator` werden als Wert an die Suchfunktion übergeben, es muss also ein Copy-Konstruktor implementiert sein.
- In der Abbruchbedingung der `while`-Schleife
  - werden zwei Iteratoren auf Ungleichheit geprüft und
  - der Iterator `start` wird dereferenziert, um das aktuelle Element mit dem gesuchten Element `x` zu vergleichen.
- Im Rumpf der Schleife wird der Iterator `start` inkrementiert.
- Iterator `start` wird per Copy-Konstruktor zurückgegeben.

Iterator für einen einfachen Vektor-Typ

```
template <typename T>
class MyVector {
    int _next, _size;
    T *_values;

public:
    MyVector(int size = 10) {
        _next = 0;
        _size = size;
        _values = new T[size];
    }
    void insert(T val) {
        if (_next == _size)
            throw "no space left";
        _values[_next] = val;
        _next += 1;
    }
    .....
```

Iterator für einen einfachen Vektor-Typ

```
class Iterator {
private:
    T *_cursor;
public:
    Iterator(T *cursor = 0) {  // Copy-Konstr.?
        _cursor = cursor;
    }
    bool operator!=(Iterator iter) const {
        return _cursor != iter._cursor;
    }
    T& operator*() const {
        return *_cursor;
    }
    Iterator operator++(int) {  // postfix
        Iterator it = *this;
        _cursor++;
        return it;
    }
}; // Ende des Iterators
```

Iterator für einen einfachen Vektor-Typ

```
    // weiter mit MyVector

    Iterator start() {
        return _values;  // Rückgabe-Typ ok?
    }

    Iterator ende() {
        return _values + _next;
    }
}; // Ende der Klasse MyVector

// globale Funktion: nur 1x implementiert für alle Container!
template <typename Iterator, typename T>
Iterator suchen(Iterator start, Iterator ende, T x) {
    while (start != ende && *start != x)
        start++;

    return start;
}
```

Iterator für einen einfachen Vektor-Typ

```
int main(void) {
    MyVector<int> v;

    for (int i = 1; i < 10; i++)
        v.insert(i);

    MyVector<int>::Iterator iter;

    iter = suchen(v.start(), v.ende(), 6);
    if (iter != v.ende())
        cout << "6 gefunden\n";
    else cout << "6 nicht gefunden\n";
}
```

Postfix- und Präfix-Operator ++

Inkrement- und Dekrementoperatoren:

`++i` erhöhe `i` um 1, *bevor* `i` im Ausdruck weiterverwendet wird (Präfix-Notation)

`i++` erhöhe `i` um 1, *nachdem* `i` im Ausdruck weiterverwendet wird (Postfix-Notation)

Ausdrücke werden unter Umständen schwer einsichtig:

```
int x = 1;
int y = ++x + 1;
        // hier: x = 2, y = 3
x = 1;
y = x++ + 1;
        // hier: x = 2, y = 2
x = 1;
x = ++x + 1;
        // hier: x = 3
```

Diese Operatoren können nur auf Variablen angewendet werden, nicht auf Ausdrücke wie `(i+j)++`.

Postfix- und Präfix-Operator ++

Für die innere Klasse `Iterator` unseres Vektor-Typs können wir definieren:

```
Iterator Iterator::operator++() {
    _cursor++;              // ~~      präfix !!!
    return *this;
}

Iterator Iterator::operator++( int ) {
    Iterator it = *this;    // ~~~~  postfix !!!
    _cursor++;      // cursor weiter setzen aber
    return it;      // alten Wert zurück geben
}
```

Iterator für einen einfachen Listen-Typ

```
template <typename T>
class MyList {
private:
    struct Elem {
        T value;
        Elem *next;

        Elem(const T& val) {
            value = val;
            next = 0;
        }
    } *_head, *_tail;

public:
    MyList() {
        _head = 0;
        _tail = 0;
    }
```

Iterator für einen einfachen Listen-Typ

```
void append(T val) {
    Elem *e = new Elem(val);

    if (_tail == 0)
        _head = e;
    else _tail->next = e;
    _tail = e;
}
```

Iterator für einen einfachen Listen-Typ

```
class Iterator {
private:
    Elem *_cursor;
public:
    Iterator(Elem *cursor = 0) { // Copy-Konst.?
        _cursor = cursor;
    }
    bool operator!=(Iterator iter) const {
        return _cursor != iter._cursor;
    }
    T& operator*() const {
        return _cursor->value;
    }
    Iterator operator++(int) {  // postfix
        Iterator it = *this;
        _cursor = _cursor->next;
        return it;
    }
}; // Ende des Iterators
```

Iterator für einen einfachen Listen-Typ

```
    // weiter mit der Klasse MyList

    Iterator start() {
        return _head;     // Rückgabe-Typ ok?
    }

    Iterator ende() {
        return 0;
    }
}; // Ende der Klasse MyList

// globale Funktion (nur zur Wiederholung aufgeführt)
template <typename Iterator, typename T>
Iterator suchen(Iterator start, Iterator ende, T x) {
    while (start != ende && *start != x)
        start++;

    return start;
}
```

Iterator für einen einfachen Listen-Typ

```
int main(void) {
    MyList<int> l;

    for (int j = 6; j < 26; j++)
        l.append(j);

    MyList<int>::Iterator iter;

    iter = suchen(l.start(), l.ende(), 12);
    if (iter != l.ende())
        cout << "12 gefunden\n";
    else cout << "12 nicht gefunden\n";
}
```

Auf den folgenden Folien finden Sie einige Programme, die exemplarisch die Anwendung von Containern und den Umgang mit Iteratoren zeigen:

- `set` und `multiset`
- `vector`
- `map` und `multimap`
- `iterator` und `reverse_iterator`

Set[1] und Multiset[2]

```
#include <iostream>
#include <set>
#include <cstdlib>
using namespace std;

#define MAX 30

int main(void) {
    set<char> s;
    multiset<char> ms;

    for (int i = 0; i < MAX; i++) {
        char r = 'a' + rand() % 26;
        cout << r;
        s.insert(r);
        ms.insert(r);
    }
```

[1] https://en.cppreference.com/w/cpp/container/set
[2] https://en.cppreference.com/w/cpp/container/multiset

```
    cout << endl;
    set<char>::iterator iter;
    for (iter = s.begin(); iter != s.end(); iter++)
        cout << *iter;
    cout << endl;

    set<char>::reverse_iterator riter;
    for (riter = s.rbegin(); riter != s.rend(); riter++)
        cout << *riter;
    cout << endl;

    multiset<char>::iterator iter2;
    for (iter2 = ms.begin(); iter2 != ms.end(); iter2++)
        cout << *iter2;
    cout << endl;
}
```

Ausgabe:

```
nwlrbbmqbhcdarzowkkyhiddqscdxr
abcdhiklmnoqrswxyz
zyxwsrqonmlkihdcba
abbbccddddhhikklmnoqqrrrswwxyz
```

Vector[3]

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <algorithm>
using namespace std;

int main(void) {
    vector<int> v;

    for (int i = 1; i < 20; i++)
        v.push_back(i);

    vector<int>::iterator it;
    for (it = v.begin(); it != v.end(); it++) {
        if (it != v.begin())
            cout << ",";
        cout << *it;
    }
```

[3] https://en.cppreference.com/w/cpp/container/vector

```
    cout << endl;
    vector<int>::reverse_iterator rit;
    for (rit = v.rbegin(); rit != v.rend(); rit++) {
        if (rit != v.rbegin())
            cout << ",";
        cout << *rit;
    }
    cout << endl;

    it = find(v.begin(), v.end(), 10);
    v.erase(it);
    for (it = v.begin(); it != v.end(); it++) {
        if (it != v.begin())
            cout << ",";
        cout << *it;
    }
    cout << endl;
}
```

Ausgabe:

```
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19
19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1
1,2,3,4,5,6,7,8,9,11,12,13,14,15,16,17,18,19
```

`std::vector::at`[4]

```
      reference at (size_type n);
const_reference at (size_type n) const;
```

Returns a reference to the element at position `n` in the vector.

The function automatically checks whether `n` is within the bounds of valid elements in the vector, throwing an `out_of_range` exception if it is not (i.e., if `n` is greater than, or equal to, its size). This is in contrast with member `operator[]`, that does not check against bounds.

[4] `http://www.cplusplus.com/reference/vector/vector/at/`

std::vector::operator[][5]

```
      reference operator[] (size_type n);
const_reference operator[] (size_type n) const;
```

Returns a reference to the element at position n in the vector container.

A similar member function, `vector::at`, has the same behavior as this operator function, except that `vector::at` is bound-checked and signals if the requested position is out of range by throwing an `out_of_range` exception.

Portable programs should never call this function with an argument `n` that is out of range, since this causes undefined behavior.

[5] http://www.cplusplus.com/reference/vector/vector/operator[]/

Map[6] und Multimap[7]

```
#include <iostream>
#include <string>
#include <map>
#include <cstdlib>
using namespace std;

string randomString() {
    string res;

    for (int i = 0; i < 4; i++)
        res += rand() % 26 + 'a';

    return res;
}
```

[6] https://en.cppreference.com/w/cpp/container/map
[7] https://en.cppreference.com/w/cpp/container/multimap

Map und Multimap

```
int main(void) {
    map<int, string> m1;
    multimap<int, string> m2;

    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 3; j++) {
            string s = randomString();
            // m1.insert(pair<int, string>(i, s));
            m1.insert({i, s});     // seit C++11
            // m2.insert(pair<int, string>(i, s));
            m2.insert({i, s});     // seit C++11
        }

    map<int, string>::iterator it1;
    for (it1 = m1.begin(); it1 != m1.end(); it1++)
        cout << (*it1).first << ": "
             << (*it1).second << endl;
    ...
```

```
    multimap<int, string>::iterator it2;
    for (it2 = m2.begin(); it2 != m2.end(); it2++)
        cout << (*it2).first << ": "
             << (*it2).second << endl;

    return 0;
}
```

Ausgabe:

```
0: nwlr
1: arzo
2: qscd
3: xsjy
```

Ausgabe:

```
0: nwlr
0: bbmq
0: bhcd
1: arzo
1: wkky
1: hidd
2: qscd
2: xrjm
2: owfr
3: xsjy
3: bldb
3: efsa
```

In C wird beim Aufruf von `bsearch` oder `qsort` ein Zeiger auf eine Funktion `compar` übergeben, die eine Vergleichsoperation für zwei Feldelemente implementiert.

- ```
  void qsort(void *base, size_t nmemb, size_t size,
             int (*compar)(void *, void *))
  ```

  sortiert ein Array mit `nmemb` Elementen der Größe `size` (erstes Element bei `base`).

- ```
  void *bsearch(void *key, void *base, size_t nmemb,
                size_t size, int (*compar)(void *, void *))
  ```

  durchsucht ein Array mit `nmemb` Elementen der Größe `size` (erstes Element bei `base`) nach einem Element `key`.

Der Aufruf der Vergleichsfunktion erfolgt aus `bsearch` bzw. aus `qsort` heraus *zurück* in das Anwendungsprogramm, deshalb wird diese Technik als *Callback* bezeichnet.

Jede Funktion besitzt eine Adresse:

```
int min(int a, int b) {
    return (a < b) ? a : b;
}
int max(int a, int b) {
    return (a > b) ? a : b;
}

int main(void) {
    int (*fp)(int, int);   // fp: Zeiger auf Funktion

    fp = min;              // fp zeigt auf min
    printf("min(5, 7) = %d\n", (*fp)(5, 7));

    fp = max;              // fp zeigt auf max
    printf("max(5, 7) = %d\n", (*fp)(5, 7));
    return 0;
}
```

Erklärung:

- `fp` ist ein Zeiger auf eine Funktion, die einen `int`-Wert liefert und zwei `int`-Werte als Parameter verlangt.
- `*fp` kann für die Funktionen `min` und `max` benutzt werden.
- nicht verwechseln mit `int *fp(int, int)`: Funktion, die zwei `int`-Werte als Parameter hat und einen Zeiger auf einen `int`-Wert als Ergebnis liefert!
- `fp` kann zugewiesen, in Vektor eingetragen, an Funktion übergeben werden usw..

```
#include <stdio.h>
void main(void) {
    int zahl;
    int (*fptr[])(const char *, ...) = { scanf, printf };

    (*fptr[1])("Zahl? ");
    (*fptr[0])("%d", &zahl);
    (*fptr[1])("Die Zahl lautet %d\n", zahl);
}
```

In C++ tritt an die Stelle von Zeigern auf Funktionen das allgemeinere Konzept des Funktionsobjekts, welches auch als Functor bezeichnet wird:

> Ein Functor ist ein Objekt, welches `operator()` definiert.

Im folgenden Beispiel:

- Der Functor erhält das Element per Referenz und kann deshalb dessen Wert ändern.
- Innerhalb der `for_each`-Parameterklammer wird das Funktionsobjekt definiert, indem der Konstruktor `Limiter` mit der unteren Grenze `lower` und der oberen Grenze `upper` aufgerufen wird.
- Der überladene Operator `()` wird intern von `for_each` aufgerufen.

```
#include <iostream>
#include <vector>
#include <algorithm>

class Limiter {
    int _lowerBound, _upperBound;
public:
    Limiter(int l, int u) {
        _lowerBound = l;
        _upperBound = u;
    }

    void operator()(int& value) {
        if (value < _lowerBound)
            value = _lowerBound;
        else if (value > _upperBound)
            value = _upperBound;
    }
};
```

```
int main(void) {
    int lower, upper;

    std::cout << "untere Grenze: ";
    std::cin >> lower;

    std::cout << "obere Grenze: ";
    std::cin >> upper;

    std::vector<int> v;
    std::vector<int>::iterator it;

    for (int i = -5; i <= 5; i++)
        v.push_back(i);

    ...
```

```
    for (it = v.begin(); it != v.end(); it++)
        std::cout << *it << std::endl;
    std::cout << std::endl;

    std::for_each(v.begin(), v.end(), Limiter(lower, upper));

    for (it = v.begin(); it != v.end(); it++)
        std::cout << *it << std::endl;
}
```

Ausgabe für `lower = -2` und `upper = 2`:

```
vorher:    -5 -4 -3 -2 -1  0  1  2  3  4  5
nachher:   -2 -2 -2 -2 -1  0  1  2  2  2  2
```

Das obige Beispiel wäre mit einer einfachen Callback-Funktion nicht so elegant lösbar: Wir müssen dann globale Variablen nutzen, siehe nächste Folien.

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

int lowerBound, upperBound;      // globale Variablen

void limiter(int& val) {         // Funktion, kein Objekt
    if (val < lowerBound)
        val = lowerBound;
    else if (val > upperBound)
        val = upperBound;
}
...
```

```
int main(void) {
    cout << "untere Grenze: ";
    cin >> lowerBound;   // schreibt in globale Variable
    cout << "obere Grenze: ";
    cin >> upperBound;   // schreibt in globale Variable

    vector<int> v;

    for (int i = -5; i <= 5; i++)
        v.push_back(i);

    for_each(v.begin(), v.end(), limiter);

    vector<int>::iterator it;

    for (it = v.begin(); it != v.end(); it++)
        cout << *it << endl;
}
```

Auch wenn in einem Container keine Objekte sondern Zeiger auf Objekte gespeichert werden, sind Functoren sinnvoll einsetzbar. Algorithmen wie `sort` und `min_element`

```
template <class RandomIt>
void sort(RandomIt first, RandomIt last);
template<class RandomIt, class Compare>
void sort(RandomIt first, RandomIt last, Compare comp);

template <class ForwardIt>
ForwardIt min_element(ForwardIt first, ForwardIt last);
template<class ForwardIt, class Compare>
ForwardIt min_element(ForwardIt first, ForwardIt last,
                      Compare comp);
```

benötigen Vergleichsoperatoren `comp`, ebenso der Container `priority_queue`. Da die Vergleichsoperatoren aber auf Objekten definiert sind, würden nur Zeigerwerte, aber nicht die Inhalte verglichen. Siehe dazu das folgende Beispiel.

Functor – Beispiel rationale Zahlen

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

int main(void) {
    vector<Rational *> v;

    v.push_back(new Rational(1, 2));
    v.push_back(new Rational(1, 3));
    v.push_back(new Rational(1, 4));
    v.push_back(new Rational(1, 5));
    v.push_back(new Rational(1, 6));

    Rational *min = *min_element(v.begin(), v.end());
    cout << "Minimum: " << *min << endl;
```

Functor – Beispiel rationale Zahlen

```
    Rational *max = *max_element(v.begin(), v.end());
    cout << "Maximum: " << *max << endl;

    vector<Rational *>::iterator it;

    cout << " vor dem Sortieren:\n";
    for (it = v.begin(); it != v.end(); it++)
        // *it liefert nur einen Zeiger auf Rational
        cout << **it << endl;   // 2x Inhaltsoperator anwenden

    sort(v.begin(), v.end());

    cout << "nach dem Sortieren:\n";
    for (it = v.begin(); it != v.end(); it++)
        // *it liefert nur einen Zeiger auf Rational
        cout << **it << endl;   // 2x Inhaltsoperator anwenden
}
```

Ausgabe:

```
Minimum: (1/2)
Maximum: (1/6)

 vor dem Sortieren:       nach dem Sortieren:
 (1/2)                    (1/2)
 (1/3)                    (1/4)
 (1/4)                    (1/3)
 (1/5)                    (1/5)
 (1/6)                    (1/6)
```

Sieht nicht ganz richtig aus, oder? Hier wurden Zeigerwerte, aber nicht deren Inhalt verglichen!

Korrekt wäre:

```
class RationalComparer {
public:
    bool operator()(Rational *a, Rational *b) {
        return *a < *b;
    }
} rcmp;      // Objekt vom Typ RationalComparer wird angelegt

int main(void) {
    vector<Rational *> v;

    v.push_back(new Rational(1, 2));
    v.push_back(new Rational(1, 3));
    v.push_back(new Rational(1, 4));
    v.push_back(new Rational(1, 5));
    v.push_back(new Rational(1, 6));
```

Functor – Beispiel rationale Zahlen

```
    Rational *min = *min_element(v.begin(), v.end(), rcmp);
    cout << "Minimum: " << *min << endl;

    Rational *max = *max_element(v.begin(), v.end(), rcmp);
    cout << "Maximum: " << *max << endl;

    vector<Rational *>::iterator it;

    cout << " vor dem Sortieren:\n";
    for (it = v.begin(); it != v.end(); it++)
        cout << **it << endl;

    sort(v.begin(), v.end(), rcmp);

    cout << "nach dem Sortieren:\n";
    for (it = v.begin(); it != v.end(); it++)
        cout << **it << endl;
}
```

Es können mehrere Methoden durch `operator()` bereitgestellt werden, obwohl die Methoden keinen Namen haben: Sie müssen sich anhand der Signatur unterscheiden.

```
class Foo {
private:
    int _upLmt;

public:
    Foo(int upLmt) : _upLmt(upLmt) {}

    void operator()(int& x) {
        if (x > _upLmt)
            x = _upLmt;
    }

    bool operator()(int x, int y) {
        return x < y;
    }
};
```

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

int main(void) {
    Foo foo(7);
    vector<int> v = {9, 8, 7, 6, 1, 2, 3, 4, 5};   // Initial.

    int min = *min_element(v.begin(), v.end());
    cout << "min: " << min << endl;

    for_each(v.begin(), v.end(), foo);   // aus std::algorithm

    for (int x: v) // alternative Schreibweise: for all x in v
        cout << x << endl;
}
```

Predicates

Predicates sind Funktionen oder Funktoren, die einen Wert vom Typ `bool` zurückliefern.

- Predicates steuern in Abhängigkeit von einer Bedingung eine Aktion in den Algorithmen der STL.
- Beispiel: `remove_if()` entfernt Elemente, die einer bestimmten Bedingung genügen, aus dem Container.

  Die Bedingung ist als Predicate zu definieren. Liefert das Predicate `true` zurück, so wird das an das Predicate übergebene Element aus dem Container entfernt.

Achtung: Obige Klasse `Foo` kann nicht um ein weiteres Predicate erweitert werden, siehe folgende Folien.

Predicates

```
class Foo {
    int _upLmt, _rmLmt;

public:
    Foo(int upLmt, int rmLmt) : _upLmt(upLmt), _rmLmt(rmLmt) {}

    void operator()(int& x) {
        if (x > _upLmt)
            x = _upLmt;
    }

    bool operator()(int x) {    // zusätzliches Predicate
        return x > _rmvLmt;
    }

    bool operator()(int x, int y) {
        return x < y;
    }
};
```

Fehlermeldung des Compilers:

```
/usr/include/c++/7/bits/stl_algo.h:3884:5: error:
     call of ‘(Foo) (int&)’ is ambiguous
operator.cpp:13: note: candidate: void Foo::operator()(int&)
     void operator()(int& x) {
          ^~~~~~~~
operator.cpp:18: note: candidate: bool Foo::operator()(int)
     bool operator()(int x) {
          ^~~~~~~~
```

Zur Signatur einer Funktion/Methode gehört nicht deren Rückgabetyp!

Daher kann der Compiler nicht zwischen den Methoden `bool operator()(int x)` und `void operator()(int& x)` unterscheiden und nicht die richtige auswählen.