# 7 Container und Iteratoren

## 7.1 Standardcontainer

### 7.1.1 Containertypen

#### Sequentielle Containertypen

- `array<T, N>` (seit C++11)
  Lediglich eine Verpackung des Reihentyps `T [N]`, die analog zu anderen Containertypen verwendet werden kann.

- `vector<T>`
  Eine Reihe, die bei Bedarf automatisch vergrößert wird.

  `vector<bool>` ist eine Spezialisierung, die die Elemente kompakt in einem Bitvektor speichert.

- `deque<T>` (double-ended queue)
  Typischerweise eine Reihe von Reihen, die bei Bedarf automatisch vergrößert wird und bei der Elemente an beiden „Enden“ effizient hinzugefügt und entfernt werden können.

- `forward_list<T>` (seit C++11)
  Eine einfach verkettete Liste.

- `list<T>`
  Eine doppelt verkettete Liste.

### Adapter für sequentielle Containertypen

- `stack<T>`
  Ein LIFO-Container (last in, first out).

- `queue<T>`
  Ein FIFO-Container (first in, first out).

- `priority_queue<T>`
  Eine Vorrangwarteschlange.

### Weitere sequenzartige Typen

- `bitset<N>`
  Logisch eine Verpackung des Reihentyps `bool [N]`, dessen Elemente aber (wie bei `vector<bool>`) kompakt in einem Bitvektor gespeichert werden.

- `basic_string<T>`
  Eine Sequenz von Elementen mit zusätzlichen Funktionen wie z. B. `substr` und `operator+`.

  `string` ist lediglich ein Alias für `basic_string<char>`, `wstring` ein Alias für `basic_string<wchar_t>` usw.

## Assoziative Containertypen

- `set<Key>`
  Eine (typischerweise als Baum implementierte) geordnete Menge, die jedes Element nur einmal enthalten kann.

- `multiset<Key>`
  Eine (typischerweise als Baum implementierte) geordnete Menge, die Elemente auch mehrfach enthalten kann.

- `unordered_set<Key>` (seit C++11)
  Eine als Streuwerttabelle (hash table) implementierte Menge, die jedes Element nur einmal enthalten kann.

- `unordered_multiset<Key>` (seit C++11)
  Eine als Streuwerttabelle (hash table) implementierte Menge, die Elemente auch mehrfach enthalten kann.

- `map<Key, T>`
  Eine (typischerweise als Baum implementierte) geordnete Tabelle von Schlüssel-Wert-Paaren, die jeden Schlüssel nur einmal enthalten kann.

- `multimap<Key, T>`
  Eine (typischerweise als Baum implementierte) geordnete Tabelle von Schlüssel-Wert-Paaren, die Schlüssel auch mehrfach enthalten kann.

- `unordered_map<Key, T>` (seit C++11)
  Eine als Streuwerttabelle (hash table) implementierte Tabelle von Schlüssel-Wert-Paaren, die jeden Schlüssel nur einmal enthalten kann.

- `unordered_multimap<Key, T>` (seit C++11)
  Eine als Streuwerttabelle (hash table) implementierte Tabelle von Schlüssel-Wert-Paaren, die Schlüssel auch mehrfach enthalten kann.

**Anmerkungen**

- Anders als in Java und anderen Sprachen, gibt es keine gemeinsame (abstrakte) Basisklasse wie `Collection` o. ä.

- Alle Operationen sind aus Effizienzgründen bewusst nicht-virtuell, d. h. es findet kein dynamisches Binden statt.

- Die Palette der angebotenen Operationen kann je nach Containertyp sehr unterschiedlich sein.
- Die gleiche Operation kann je nach Containertyp sehr unterschiedliche Laufzeit besitzen.
- Deshalb ist die Wahl eines bestimmten Containertyps eine weitreichende Entwurfsentscheidung, die oft nur mit großem Aufwand geändert werden kann.
- Die Adaptertypen `stack`, `queue` und `priority_queue` bieten lediglich angepasste und eingeschränkte Schnittstellen zu einem anderen sequentiellen Containertyp an.
- Eine Menge entspricht logisch und implementierungstechnisch einer Tabelle, die nur Schlüssel und keine zugehörigen Werte speichert.
- Die Namen sind zum Teil historisch begründet:
  - Weil die als Streuwerttabellen implementierten Mengen und Tabellen erst 2011 zur Standardbibliothek hinzugefügt wurden und zu diesem Zeitpunkt bereits viele andere Bibliotheken Typen wie `hash_set` und `hash_map` anboten, wurden vorsichtshalber andere Namen gewählt. (Und die geordneten Container wurden aus Kompatibilitätsgründen natürlich nicht in `ordered_set` etc. umbenannt.)
  - Da es vor 2011 nur doppelt verkettete Listen gab, wurden diese einfach `list` (und nicht etwa `bidirectional_list` o. ä.) genannt (und später nicht umbenannt).

## 7.1.2 Wichtige Operationen auf Containern

- `c.size()` liefert die Größe des Containers `c`, d. h. die Anzahl der Elemente, die er momentan enthält.
  (Nicht zu verwechseln mit der momentanen *Kapazität* eines Vektors, die größer sein kann.)

- `c.empty()` liefert genau dann `true`, wenn der Container `c` momentan leer ist, d. h. keine Elemente enthält.

- `c.front()` bzw. `c.back()` liefert eine Referenz auf das erste bzw. letzte Element des Containers `c`.
  Wenn `c` leer ist, ist das Verhalten undefiniert.

- Für `i` zwischen 0 einschließlich und `c.size()` ausschließlich liefern `c[i]` und `c.at(i)` eine Referenz auf das `i`-te Element des sequentiellen Containers `c`.
  Wenn `i` größer oder gleich `c.size()` ist, ist das Verhalten von `c[i]` undefiniert, während `c.at(i)` eine Ausnahme des Typs `out_of_range` wirft.
  (Da der Parametertyp `size_t` vorzeichenlos ist, kann der Fall `i` kleiner 0 prinzipiell nicht auftreten, weil negative Werte in positive umgewandelt werden.)

- Für einen Schlüssel `k` liefern `c[k]` und `c.at(k)` eine Referenz auf den zu `k` gehörenden Wert der Tabelle (`map` oder `unordered_map`) `c`.
  Wenn die Tabelle keinen Eintrag mit Schlüssel `k` enthält, erstellt `c[k]` einen entsprechenden Eintrag mit Wert `T()` und liefert eine Referenz auf diesen Wert, während `c.at(k)` eine Ausnahme des Typs `out_of_range` wirft.

- `c.push_front(x)` bzw. `c.push_back(x)` fügt das Element `x` am Anfang bzw. am Ende des sequentiellen Containers `c` hinzu.

- `c.pop_front()` bzw. `c.pop_back()` entfernt das erste bzw. letzte Element des sequentiellen Containers `c`.
  Wenn `c` leer ist, ist das Verhalten undefiniert.

- `c.insert(i, x)` (bzw. `c.insert_after(i, x)` bei einer `forward_list` `c`) fügt das Element `x` vor (bzw. nach) der durch den Iterator `i` (vgl. § 7.2.1) bezeichneten Position in den sequentiellen Container `c` ein und liefert einen Iterator auf das eingefügte Element.

- `c.insert(x)` fügt das Element `x` in den assoziativen Container `c` ein, sofern es noch nicht enthalten ist oder der Container Elemente mehrfach enthalten kann.
  Wenn `c` eine Tabelle ist, muss `x` ein `std::pair<const Key, T>` sein.

- `c.erase(i)` (bzw. `c.erase_after(i)` bei einer `forward_list` `c`) entfernt das durch den Iterator `i` bezeichnete Element (bzw. das Element danach) aus dem Container `c`.

- `c.erase(x)` bzw. `c.erase(k)` entfernt das Element `x` bzw. das Element mit Schlüssel `k` aus der Menge bzw. Tabelle `c`, sofern ein solches Element enthalten ist. Falls mehrere solche Elemente enthalten sind, werden alle entfernt.

## Verfügbarkeit und Laufzeit der Operationen

| | array | vector | deque | list | forward _list | [multi] set/map | unordered_ [multi]set/map |
|---|---|---|---|---|---|---|---|
| front | kon. | kon. | kon. | kon. | kon. | | |
| back | kon. | kon. | kon. | kon. | | | |
| [] at | kon. | kon. | kon. | | | log.[4] | kon.[4] |
| push_front pop_front | | | kon.[1] | kon. | kon. | | |
| push_back pop_back | | kon.[1] | kon.[1] | kon. | | | |
| insert erase | | lin.[2] | lin.[2] | kon. | kon.[3] | log. | kon. |

- „kon.“ bedeutet konstante Laufzeit.
- „lin.“ bedeutet lineare Laufzeit bezogen auf die momentane Größe des Containers.
- „log.“ bedeutet logarithmische Laufzeit bezogen auf die die momentane Größe des Containers.
- Ein leerer Eintrag bedeutet, dass es die entsprechende Operation nicht gibt.

## Fußnoten zur Tabelle

1. Die Laufzeit der `push`-Operationen bei `vector` und `deque` ist *im Durchschnitt* konstant.
   Einzelne Operationen, die eine Vergrößerung des internen Speichers erfordern, können jedoch lineare Laufzeit besitzen.

2. Die Laufzeiten von `insert` und `erase` bei `vector` und `deque` beziehen sich auf den *allgemeinen* Fall.
   Wenn eine solche Operation gleichbedeutend mit einer (bei diesem Containertyp verfügbaren) `push`- bzw. `pop`-Operation ist, ist ihre Laufzeit (im Durchschnitt) konstant.
   Bei `vector` ist die Laufzeit von `insert` und `erase` allgemein proportional zur Anzahl der *nachfolgenden* Elemente.

3. Bei `forward_list` heißen die Operationen nicht `insert` und `erase`, sondern `insert_after` und `erase_after`.

4. Von den insgesamt acht assoziativen Containertypen besitzen nur `map` und `unordered_map` die Operationen `[]` und `at`.

## 7.1.3 Verwendung geordneter Mengen und Tabellen

- Die Containertypen `set`, `multiset`, `map` und `multimap` verwenden standardmäßig den Kleiner-Operator des Element- bzw. Schlüsseltyps `Key` zur Sortierung der Einträge.

- Die Gleichheit zweier Elemente bzw. Schlüssel `x` und `y` wird indirekt mittels `!(x < y) && !(y < x)`, d. h. durch zwei Aufrufe des Kleiner-Operators überprüft.
(Obwohl die Gleichheit zweier Elemente seit C++20 mit nur einem Aufruf des Dreiwegvergleichs <=> effizienter überprüft werden könnte, wird nach wie vor der Kleiner-Operator verwendet.)

- Für einen benutzerdefinierten Typ muss ggf. ein passender Kleiner-Operator definiert werden, zum Beispiel (vgl. § 5.1.2):

```
bool operator< (Rational x, Rational y) {
  return double(x.num)/x.den < double(y.num)/y.den;
}
```

- Um ein anderes Vergleichskriterium zu verwenden, kann als zusätzlicher Schablonenparameter eine Klasse angegeben werden, die eine konstante Elementfunktion `operator()` mit zwei Parametern des Element- bzw. Schlüsseltyps `Key` und Resultattyp `bool` besitzt, die dann anstelle des Kleiner-Operators verwendet wird.

## Beispiel

```
using str = const char*;

set<str> ss1;

struct StrLess {
  bool operator() (str s1, str s2) const {
    return strcmp(s1, s2) < 0;
  }
};

set<str, StrLess> ss2;
```

- Die Menge `ss1` verwendet zum Vergleich von Elementen den Kleiner-Operator für Zeiger, was vermutlich nicht sinnvoll ist, weil inhaltlich gleiche Zeichenketten mit verschiedenen Adressen dann als „ungleich“ behandelt werden.

- Die Menge `ss2` hingegen verwendet die in der Klasse `StrLess` definierte Vergleichsfunktion, die einen inhaltlichen Vergleich der Zeichenketten durchführt.

## 7.1.4 Verwendung ungeordneter Mengen und Tabellen

- Die Containertypen `unordered_set`, `unordered_multiset`, `unordered_map` und `unordered_multimap` verwenden standardmäßig den Gleichheitsoperator des Element- bzw. Schlüsseltyps `Key` zum Vergleich von Einträgen sowie die Elementfunktion `operator()` der Bibliotheksklasse `std::hash<Key>` (sofern diese existiert) zur Berechnung von Streuwerten.

- Für einen benutzerdefinierten Typ müssen diese Funktionen ggf. passend definiert werden, zum Beispiel (vgl. § 5.1.2):

```
bool operator== (Rational x, Rational y) {
  return x.num * y.den == y.num * x.den;
}

template <>
struct std::hash<Rational> {
  size_t operator() (Rational x) const {
    return x.num << 16 | x.den;
  }
}
```

(Sowohl `std::` als auch `const` ist notwendig!)

- Um andere Funktionen zu verwenden, können als zusätzliche Schablonenparameter zwei Klassen `Hash` und `Pred` angegeben werden, die jeweils eine passende Elementfunktion `operator()` besitzen.

- In jedem Fall müssen die Funktionen zum Vergleich von Elementen und zur Berechnung von Streuwerten semantisch zusammenpassen, d. h. Objekte, die gemäß der Vergleichsfunktion „gleich“ sind, müssen den gleichen Streuwert besitzen.

**Beispiel (vgl. § 7.1.3)**

```
using str = const char*;

unordered_set<str> ss1;

struct StrEqual {
  bool operator() (str s1, str s2) const {
    return strcmp(s1, s2) == 0;
  }
};
```

```
struct StrHash {
  // Streuwert der Zeichenkette s.
  // Berechnung wie bei java.lang.String.hashCode.
  // Arithmetischer Überlauf ist für vorzeichenlose Typen
  // wie size_t wohldefiniert.
  size_t operator() (str s) const {
    size_t h = 0;
    while (char c = *s++) h = h * 31 + c;
    return h;
  }
};

unordered_set<str, StrHash, StrEqual> ss2;
```

- Die Menge `ss1` verwendet zum Vergleich von Elementen den Gleichheitsoperator für Zeiger und zur Berechnung von Streuwerten die entsprechende Spezialisierung von `std::hash`, was – ähnlich wie in § 7.1.3 – vermutlich beides nicht sinnvoll (aber zusammen semantisch korrekt) ist.

- Die Menge `ss2` hingegen verwendet die in den Klassen `StrEqual` und `StrHash` definierten Funktionen, die jeweils den Inhalt der vorliegenden Zeichenketten berücksichtigen.

# 7.2 Iteratoren

## 7.2.1 Grundprinzip

- *Iteratoren* sind eine Verallgemeinerung von Zeigern: Obwohl es sich um Objekte beliebiger Typen handeln kann, können sie mehr oder weniger wie Zeiger verwendet werden, weil Operatoren wie `*` und `++` bei Bedarf entsprechend überladen sind.

- Ein Iterator `i` verweist normalerweise auf ein Element irgendeiner *Folge* von Objekten, beispielsweise eines Containers.
  In diesem Fall liefert `*i` dieses Element (je nach Typ des Iterators als Referenz, `const`-Referenz oder als Wert).

- Ein Iterator kann aber z. B. auch auf das Ende einer Folge verweisen, d. h. auf die Position *nach dem letzten Element* der Folge (vgl. § 2.3.7).
  In diesem Fall ist das Verhalten von `*i` undefiniert.

- Analog zu einem Zeiger, kann ein *Vorwärtsiterator* (forward iterator) `i`, der auf ein Element einer Folge verweist, mittels `++i` oder `i++` inkrementiert werden, damit er auf das nächste Element der Folge verweist. (Das heißt, ein Vorwärtsiteratortyp muss sowohl Präfix- als auch Postfix-Inkrement unterstützen. `++i` liefert den Iterator, der auf das nächste Element der Folge verweist, `i++` den ursprünglichen Iterator.)

- Außerdem können zwei Iteratoren `i` und `j` mittels `i == j` und `i != j` verglichen werden. (Das heißt, jeder Iteratortyp muss sowohl Gleichheit als auch Ungleichheit unterstützen, wofür seit C++20 aber die Definition des Gleichheitsoperators ausreicht, vgl. § 6.2.2). Zwei Iteratoren sind genau dann gleich, wenn sie auf das gleiche Element bzw. auf die gleiche Position verweisen.

- Damit ermöglichen Iteratoren u. a. die Iteration über alle Elemente einer Folge, ohne dass man deren interne Datenstruktur kennen muss:
  Wenn eine Folge `s` Elementfunktionen `begin` und `end` besitzt, die Iteratoren auf das erste Element bzw. auf das o. g. Ende der Folge liefern, dann kann man z. B. wie folgt alle Elemente der Folge ausgeben:

  ```
  for (auto i = s.begin(); i != s.end(); i++) cout << *i;
  ```

  (Wenn die Folge leer ist, muss `s.begin()` den gleichen Iterator liefern wie `s.end()`.)

- Ein *bidirektionaler Iterator* (bidirectional iterator) `i`, der nicht auf das erste Element einer Folge verweist, kann mittels `--i` oder `i--` auch dekrementiert werden, damit er auf das vorige Element der Folge verweist. (Das heißt, ein bidirektionaler Iteratortyp muss – zusätzlich zu Präfix- und Postfix-Inkrement sowie Gleichheit und Ungleichheit – sowohl Präfix- als auch Postfix-Dekrement unterstützen. `--i` liefert den Iterator, der auf das vorige Element der Folge verweist, `i--` den ursprünglichen Iterator.)

- Damit könnte eine Iteration in umgekehrter Richtung wie folgt implementiert werden (siehe aber auch § 7.2.2):

```
auto i = s.end();
while (i != s.begin()) cout << *--i;
```

(Beachte die Asymmetrie von `begin` und `end`: `begin` liefert einen Iterator auf das erste Element der Folge, `end` jedoch auf die Position *nach* dem letzten Element.)

- *Iteratoren mit wahlfreiem Zugriff* (random access iterators) erlauben zusätzlich beliebige „Adressarithmetik“ analog zu Zeigern (vgl. § 2.3.7) sowie beliebige Vergleiche, d. h. für derartige Iteratoren `i` und `j` und eine ganze Zahl `n` müssen zusätzlich alle folgenden Ausdrücke möglich sein:

```
i + n          i += n          n + i
i - n          i -= n          i - j
i < j          i <= j
i > j          i >= j
```

Außerdem ist `i[n]` wie bei Zeigern äquivalent zu `*(i+n)`.

- Daraus folgt: Jeder Zeiger ist ein random access iterator (und damit auch ein bidirektionaler Iterator und ein Vorwärtsiterator).

## 7.2.2 Iteratoren der Standardcontainer

- Jeder in § 7.1.1 genannte Containertyp der Standardbibliothek (außer `bitset` und den Adaptertypen `stack`, `queue` und `priority_queue`) besitzt Elementfunktionen `begin` und `end` sowie `cbegin` und `cend` mit folgender Bedeutung:

- `begin` und `cbegin` liefern jeweils einen Iterator auf das erste Element des Containers, sofern dieser nicht leer ist; andernfalls wird der gleiche Iterator wie bei `end` bzw. `cend` geliefert.
- `end` und `cend` liefern jeweils einen Iterator auf das Ende des Containers, d. h. auf die Position nach dem letzten Element.
- Die von `cbegin` und `cend` gelieferten Iteratoren `i` sind *konstant*, d. h. `*i` liefert jeweils eine Referenz auf ein *konstantes* Element, sodass Zuweisungen an `*i` nicht möglich sind.
- Seit C++11 sind auch die von `begin` und `end` gelieferten Iteratoren `i` von Mengen konstant, weil eine Zuweisung an `*i` die „innere Ordnung“ des Containers zerstören könnte.
  (Dass dies vor C++11 nicht so festgelegt war, war offensichtlich ein Fehler. In gängigen Implementierungen war es aber trotzdem schon so. Die Iteratoren von Tabellen verweisen auf Elemente des Typs `std::pair<const Key, T>`, an deren erste Komponente grundsätzlich nicht zugewiesen werden kann.)
- Außerdem liefern `begin` und `end` für einen konstanten Container eines beliebigen Typs immer konstante Iteratoren.
- Ansonsten liefern `begin` und `end` nicht-konstante Iteratoren `i`, d. h. Zuweisungen an `*i` sind möglich.

- Alle Containertypen außer `forward_list` und den ungeordneten Mengen und Tabellen besitzen zusätzlich Elementfunktionen `rbegin` und `rend` sowie `crbegin` und `crend`, die „spiegelbildlich“ zu den zuvor genannten Funktionen funktionieren, das heißt:
  - `rbegin` und `crbegin` liefern jeweils einen Iterator auf das *letzte* Element des Containers (*nicht* auf die Position danach) bzw. den gleichen Iterator wie `rend` bzw. `crend`.
  - `rend` und `crend` liefern jeweils einen Iterator auf den *Anfang* des Containers, d. h. auf die Position *vor* dem ersten Element.
  - Wenn ein solcher *reverse iterator* inkrementiert bzw. dekrementiert wird, verweist er anschließend auf das *vorige* bzw. *nächste* Element des Containers.
  - Damit erlauben diese Funktionen die Implementierung von *Rückwärtsiterationen* nach demselben Schema wie Vorwärtsiterationen, zum Beispiel:

```
for (auto i = c.rbegin(); i != c.rend(); i++) cout << *i;
```

  - „Normale“ und „umgekehrte“ Iteratoren besitzen potentiell unterschiedliche Typen, d. h. sie können nicht miteinander kombiniert werden.

- `forward_list` bietet stattdessen zusätzliche Elementfunktionen `before_begin` sowie `cbefore_begin`, die jeweils einen (normalen) Iterator auf den Anfang der Liste liefern, d. h. auf die Position vor dem ersten Element.
  Diese sind nützlich, um mittels `insert_after` bzw. `erase_after` Elemente am Anfang einer Liste einfügen bzw. entfernen zu können.

- Die Iteratoren der Typen `array` und `vector` erlauben wahlfreien Zugriff.

- Die Iteratoren der Typen `deque` und `list` sowie der geordneten Mengen und Tabellen (`set`, `multiset`, `map`, `multimap`) sind bidirektional.

- Die Iteratoren des Typs `forward_list` sowie der ungeordneten Mengen und Tabellen (`unordered_set`, `unordered_multiset`, `unordered_map`, `unordered_multimap`) sind lediglich Vorwärtsiteratoren.

- Das Einfügen und Entfernen von Elementen kann je nach Typ des Containers dazu führen, dass auch Iteratoren (sowie Zeiger und Referenzen) auf *andere* Elemente ungültig werden.

### 7.2.3 Operationen mit Iteratoren

Iteratoren treten an unzähligen Stellen der Standardbibliothek auf, zum Beispiel:

- Jeder Standardcontainertyp, der eine Elementfunktion `insert` (bzw. `insert_after`) zum Einfügen eines einzelnen Elements besitzt, besitzt eine weitere gleichnamige Elementfunktion, die anstelle eines Elements zwei beliebige Iteratoren `first` und `last` als Parameter erhält und alle Elemente von `first` einschließlich bis `last` ausschließlich in den Container einfügt.
- Außerdem besitzt jeder solche Containertyp einen Konstruktor, der ebenfalls zwei Iteratoren `first` und `last` als Parameter erhält und ebenfalls alle Elemente von `first` einschließlich bis `last` ausschließlich in den neuen Container einfügt. Damit kann ein Container bequem z. B. mit allen Elementen eines beliebigen anderen Containers initialisiert werden.
- Die Elementfunktion `find` von Mengen und Tabellen liefert als Resultat einen Iterator auf das gesuchte Element, sofern es gefunden wurde, bzw. den gleichen Iterator wie die Elementfunktion `end`, wenn es nicht gefunden wurde.
- Generische Algorithmen wie z. B. `find`, `count`, `copy`, `remove` und `sort` (vgl. Definitionsdatei `<algorithm>`) sowie reguläre Ausdrücke (vgl. Definitionsdatei `<regex>`) arbeiten immer auf Folgen (oder Teilfolgen) von Elementen, die durch zwei Iteratoren `first` und `last` begrenzt sind. (Für reguläre Ausdrücke gibt es zusätzlich „Bequemlichkeitsfunktionen“, die direkt auf Zeichenketten arbeiten.)

## 7.2.4 Beispiele

```
// Eine Liste von Zeichen.
list<char> ls;
......

// Alle in der Liste enthaltenen Ziffern ausgeben.
// (Die Deklaration von isdigit macht den überladenen
// Funktionsnamen im aktuellen Block eindeutig, sodass er
// an die generische Funktion find_if übergeben werden kann.
int isdigit (int);
for (auto i = ls.begin(), e = ls.end();
                          (i = find_if(i, e, isdigit)) != e; i++) {
  cout << *i << endl;
}

// Überprüfen, ob die Liste mindestens zwei
// aufeinanderfolgene Ziffern enthält.
regex r ("[0-9]{2,}");
cout << regex_search(ls.begin(), ls.end(), r) << endl;
```

## 7.2.5 Beispiel eines selbstdefinierten Iteratortyps

### Anforderungsdefinition

- Für zwei Werte `first` und `last` eines beliebigen Typs `T`, dessen Werte mittels Addition oder Inkrementoperatoren inkrementiert werden können, soll `range(first, last)` die Folge der Werte von `first` bis `last` (jeweils einschließlich) repräsentieren, ohne all diese Werte explizit zu speichern.

- Stattdessen sollen die Elemente einer solchen Folge indirekt über Iteratoren abfragbar sein.

- Zum Beispiel:

```
auto letters = range('a', 'z');
for (auto i = letters.begin(), e = letters.end(); i != e; i++) {
  cout << *i << endl;
}
```

- Oder kürzer (vgl. § 7.4):

```
for (char c : range('a', 'z')) {
  cout << c << endl;
}
```

## Mögliche Lösung

```
// Hilfstyp zur Speicherung eines Bereichs.
template <typename T>
struct Range {
  // Bereichsgrenzen.
  T first, last;
  Range (T first, T last) : first{first}, last{last} {}

  // Zugehöriger Iteratortyp.
  // Die Basisklasse iterator<forward_iterator_tag, T>
  // kennzeichnet den Typ als Vorwärtsiterator mit Elementtyp T
  // und ermöglicht die Verwendung von iterator_traits<Iter>,
  // was für viele Verwendungen notwendig ist.
  struct Iter : iterator<forward_iterator_tag, T> {
    // Element des Bereichs, auf das der Iterator verweist.
    T current;
    Iter (T current) : current{current} {}

    // Operator zur Abfrage dieses Elements.
    T operator* () const {
      return current;
    }
```

```
// Operatoren zum Inkrementieren/Weitersetzen des Iterators.
// Der Präfixoperator ist (wie erwartet) parameterlos und
// liefert den weitergesetzten Iterator *this per Referenz
// zurück.
// Der Postfixoperator muss zur Unterscheidung einen Dummy-
// Parameter mit Typ int besitzen und liefert den
// ursprünglichen Iterator i als Wert zurück.
// (Er verwendet sinnvollerweise den Präfixoperator.)
Iter& operator++ () {
  ++current;
  return *this;
}
Iter operator++ (int) {
  Iter i = *this;
  ++*this;
  return i;
}
```

```
    // Operatoren zum Vergleich mit einem anderen Iterator that.
    // (Auch hier verwendet ein Operator sinnvollerweise den
    // anderen. Ab C++20 muss der Ungleichoperator nicht mehr
    // explizit definiert werden.)
    bool operator== (const Iter& that) const {
      return current == that.current;
    }
    bool operator!= (const Iter& that) const {
      return !(*this == that);
    }
  };

  // Iteratoren auf das erste Element bzw. auf das Ende der Folge
  // (d. h. logisch auf die Position nach dem letzten Element).
  Iter begin () { return Iter{first}; }
  Iter end () { return Iter{last + 1}; }
};

// Die für den Benutzer sichtbare Funktionsschablone.
template <typename T>
Range<T> range (T first, T last) {
  return Range<T>{first, last};
}
```

# 7.3 Initialisiererlisten (initializer lists)

## 7.3.1 Prinzip

- An zahlreichen Stellen der C++-Grammatik können *Initialisiererlisten* verwendet werden, die aus beliebig vielen Ausdrücken in geschweiften Klammern bestehen.
- Der Übersetzer ersetzt eine solche Initialisiererliste durch ein Objekt `x` des Typs `initializer_list<T>`, wobei sich der Typ `T` meist aus dem Kontext ergibt.
- Die Typen der Ausdrücke müssen dann implizit in `T` umwandelbar sein, wobei Umwandlungen verboten sind, die einen Wert möglicherweise verfälschen könnten (sog. „narrowing conversions“, z. B. von `int` nach `char`, von `double` nach `int`, aber auch von `int` nach `double`).
- Ausnahme: Entsprechende Umwandlungen konstanter Werte sind erlaubt, wenn sich die umgewandelten Werte im Zieltyp exakt darstellen lassen (was zur Übersetzungszeit geprüft werden kann).
- Wenn sich der Typ `T` nicht aus dem Kontext ergibt, wird der Typ der Ausdrücke verwendet, der in diesem Fall für alle Ausdrücke gleich sein muss.

- Zur Laufzeit werden die Ausdrücke einer Initialisiererliste von links nach rechts ausgewertet und ihre Werte in eine anonyme, temporäre Reihe kopiert, die indirekt über das o. g. Objekt `x` verfügbar ist:
  - `x.size()` liefert die Anzahl der Elemente der Reihe bzw. der Initialisiererliste.
  - `x.begin()` liefert einen Zeiger (und damit auch einen random access iterator) mit Typ `const T*` auf das erste Element der Reihe.
  - `x.end()` liefert einen Zeiger (und damit auch einen random access iterator) mit Typ `const T*` auf das Ende der Reihe, d. h. `x.begin() + x.size()`.

## 7.3.2 Verwendung in der Standardbibliothek

Initialisiererlisten werden ebenfalls an zahlreichen Stellen der Standardbibliothek verwendet, zum Beispiel:

- Jeder Standardcontainertyp, der eine Elementfunktion `insert` (bzw. `insert_after`) zum Einfügen eines einzelnen Elements besitzt, besitzt eine weitere gleichnamige Elementfunktion, die anstelle eines Elements eine Initialisiererliste des Elementtyps als Parameter erhält und alle Elemente dieser Liste in den Container einfügt (vgl. auch § 7.2.3).
- Außerdem besitzt jeder solche Containertyp einen Konstruktor, der – auch implizit – mit einer Initialisiererliste des Elementtyps aufgerufen werden kann und alle Elemente dieser Liste in den neuen Container einfügt.
- Zum Beispiel:

```
vector<int> v = { 1, 2, 3, 4 };
v.insert(v.end(), { 5, 6, 7 });
list<vector<int>> ls = { { 1, 2 }, { 3, 4 } };

void f (const set<string>& names) { ..... }
f({ "Anton", "Berta", "Christian" });
```

- Die Funktionen `min`, `max` und `minmax` können entweder mit zwei Werten eines beliebigen Typs `T` oder mit einer entsprechenden Initialisiererliste aufgerufen werden.

## 7.3.3 Beispiele

- Summe und Durchschnittswert beliebig vieler Zahlen (vgl. § 5.5.1):

```
double sum (initializer_list<double> il) {
  double s = 0;
  for (double x : il) s += x;
  return s;
}

double avg (initializer_list<double> il) {
  if (il.size() == 0) throw invalid_argument("empty list");
  return sum(il) / il.size();
}
```

- Verwendungsmöglichkeiten:

```
sum({ 1.5, 2.7, -3.8 });

int x, y, z;
......
sum({ x, y, z });        // Fehler: narrowing conversions.
sum({ 1, 2, 3 });        // OK: 1, 2, 3 lassen sich exakt
                         // als double-Werte darstellen.
```

# 7.4 Schleifen über Bereiche (range-based for loops)

## 7.4.1 Schleifen über Reihen

- Für eine Reihe `a` mit `n` Elementen ist eine Schleife der Gestalt

```
for (T x : a) ......
```

  gleichbedeutend mit

```
for (auto i = a, e = i + n; i != e; ++i) {
  T x = *i;
  ......
}
```

  d. h. `x` durchläuft nacheinander alle Elemente von `a`.

- Um eventuelle Namenskonflikte zu vermeiden, bleiben die Namen der Hilfsvariablen `i` und `e` jedoch verborgen.

- Durch die Verwendung des Schlüsselworts `auto` kann der Code unabhängig vom konkreten Elementtyp der Reihe formuliert werden. (Der Typ von `i` und `e` ist dann der entsprechende Zeigertyp.)

## 7.4.2 Schleifen über andere Objekte

- Wenn der Typ von `a` eine Klasse mit parameterlosen Elementfunktionen `begin` und `end` ist (typischerweise ein Containertyp oder eine Initialisiererliste), ist eine Schleife der Gestalt

  ```
  for (T x : a) ......
  ```

  gleichbedeutend mit

  ```
  for (auto i = a.begin(), e = a.end(); i != e; ++i) {
    T x = *i;
    ......
  }
  ```

  d. h. wenn die Elementfunktionen die „übliche“ Bedeutung haben (vgl. § 7.2.2 und § 7.3.1), durchläuft `x` der Reihe nach alle Elemente von `a`.

- Obwohl der Ausdruck `a` in der „Übersetzung“ zweimal auftritt, wird er zur Laufzeit jedoch nur einmal ausgewertet.

- Wenn der Typ von `a` keine Klasse ist oder die Klasse keine parameterlosen Elementfunktionen `begin` und `end` besitzt, werden `a.begin()` und `a.end()` ersetzt durch `begin(a)` und `end(a)`.

- Wenn es dann keine passenden globalen Funktionen `begin` und `end` (in sog. „associated namespaces“ des Typs von `a`) gibt, erhält man einen Übersetzungsfehler.

### 7.4.3 Anmerkungen

- Nach den üblichen Regeln (vgl. § 3.3.3), wird am Anfang bzw. Ende jedes Schleifendurchlaufs der Konstruktor bzw. Destruktor von `x` ausgeführt.
- Die Variable `x` muss auf diese Weise lokal in der Schleife deklariert werden, d. h. es kann keine zuvor deklarierte Variable verwendet werden.
- Der Typ `T` kann auch ein ggf. `const`-qualifizierter Referenztyp sein.
- Anstelle von `T` kann auch `auto` verwendet werden.

### 7.4.4 Beispiele

- Ausgabe einer Menge:

```
set<int> s;
......

// Formulierung mit explizitem Iterator in C++98.
for (set<int>::iterator i = s.begin(); i != s.end(); ++i) {
  int x = *i;
  cout << x << endl;
}
```

```
// Vereinfachung durch auto in C++11.
for (auto i = s.begin(); i != s.end(); ++i) {
  int x = *i;
  cout << x << endl;
}

// Kompakte Formulierung mit verborgenem Iterator.
for (int x : s) {
  cout << x << endl;
}
```

- Verdopplung aller Elemente eines Vektors:

```
vector<int> v;
......

for (int& x : v) x *= 2;
```

- Ausgabe eines Containers mit unbekanntem Typ:

```
template <typename C>
void print (const C& c) {
  for (auto& x : c) cout << x << endl;
}
```