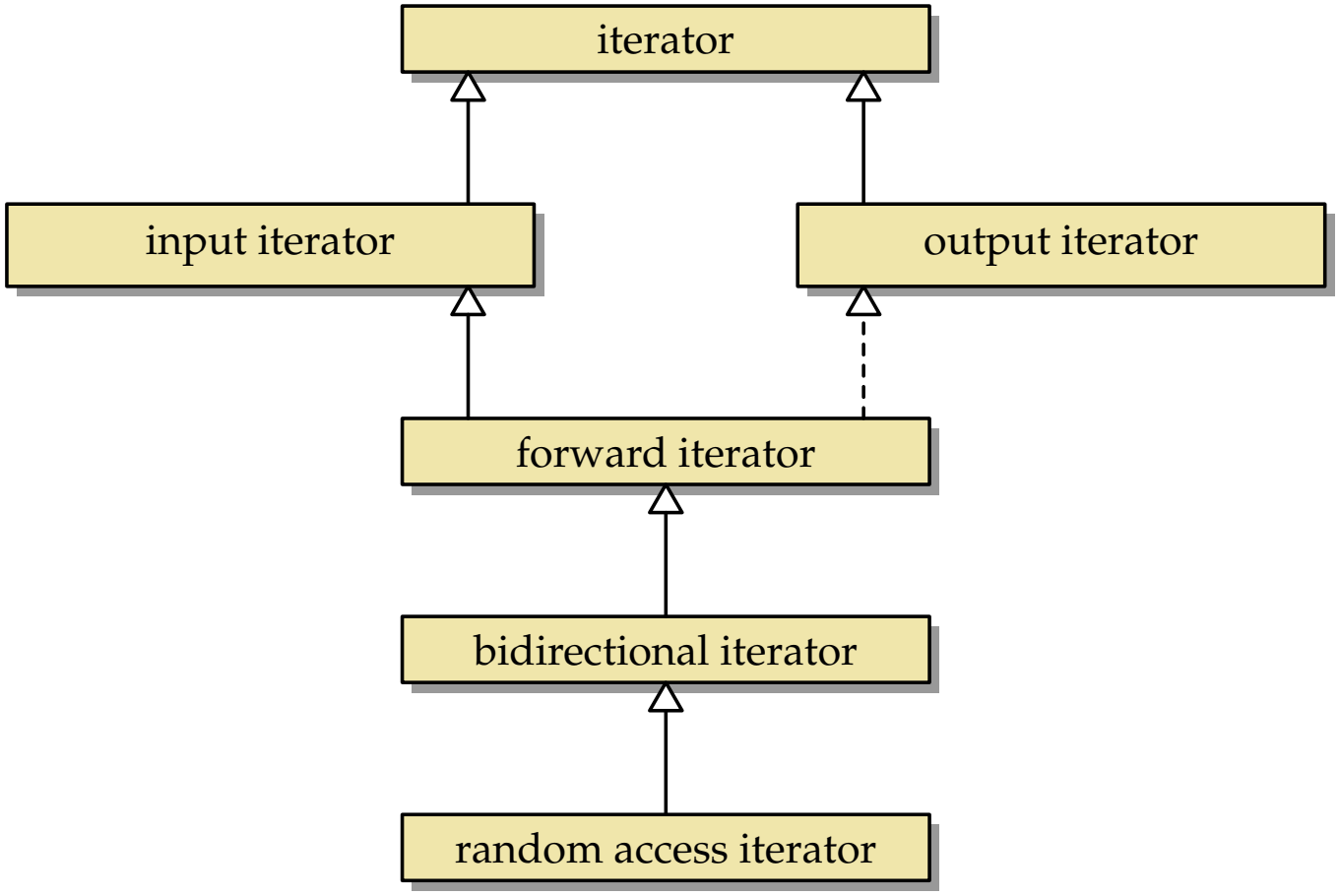
Die Standard-Template-Library (STL) bietet eine Reihe von Template-Klassen für Container, eine allgemeine Schnittstelle für Iteratoren und eine Sammlung von Algorithmen an:

- Iteratoren sind eine Verallgemeinerung von Zeigern und dienen als universelle Schnittstelle für den Zugriff auf eine Sequenz von Objekten.
- Zu den Algorithmen gehören Abfragen auf Sequenzen, die die Objekte nicht verändern (diverse Suchen, Vergleiche), solche die sie verändern (Kopieren, Verschieben, Transformieren) und sonstige Operationen (Sortieren, binäre Suche, Mengen-Operationen auf sortieren Sequenzen). Die Algorithmen arbeiten allesamt mit Iteratoren und sichern wohldefinierte Komplexitäten zu unabhängig von den verwendeten Datenstrukturen.
- Container bieten eine breite Vielfalt an Datenstrukturen, um Objekte zu beherbigen. Dazu gehören u.a. Arrays, lineare Listen, sortierte balancierte binäre Bäume und Hash-Verfahren.

- Iteratoren können als Verallgemeinerung von Zeigern gesehen werden, die einen universellen Zugriff auf Datenstrukturen erlauben.
- Durch die syntaktisch gleichartige Verwendung von Iteratoren und Zeigern können auch immer reguläre Zeiger als Iteratoren verwendet werden.
- Für die einzelnen Operationen eines Iterators gibt es einheitliche semantische Spezifikationen, die auch die jeweilige Komplexität angeben. Diese entsprechen denen der klassischen Zeiger-Operatoren.
- Dies sollte eingehalten werden, damit die auf Iteratoren arbeitenden Algorithmen semantisch korrekt sind und die erwartete Komplexität haben.
- Die auf Iteratoren basierenden Algorithmen sind immer generisch, d.h. es wird typischerweise mit entsprechenden impliziten Template-Parametern gearbeitet.



- Der C++-Standard spezifiziert eine (auf statischem Polymorphismus) beruhende semantische Hierarchie der Iteratoren-Klassen.

Alle Iteratoren erlauben das Dereferenzieren und das Weitersetzen:

Operator	Rückgabe-Typ	Beschreibung
$*it$	*Element&*	Zugriff auf ein Element; nur zulässig, wenn *it* dereferenzierbar ist
$++it$	*Iterator*	Iterator vorwärts weitersetzen

Iteratoren unterstützen Kopierkonstruktoren, Zuweisungen und *std*::*swap*. Wieweit ein Iterator weitergesetzt werden darf bzw. ob jeweils eine Dereferenzierung zulässig ist, lässt sich diesen Operationen nicht entnehmen.

Diese Iteratoren erlauben es, eine Sequenz zu konsumieren, d.h. sukzessive auf die einzelnen Elemente zuzugreifen:

Operator	Rückgabe-Typ	Beschreibung
it1 != *i2*	**bool**	Vergleich zweier Iteratoren
∗*it*	*Element*	Lesezugriff auf ein Element
it−>*member*	Typ von *member*	Lesezugriff auf ein Datenfeld
++*it*	*Iterator*&	Iterator vorwärts weitersetzen
it++	*Iterator*&	Iterator vorwärts weitersetzen
∗*it*++	*Element*	dereferenziert den Iterator und liefert diesen Wert; der Iterator wird danach weitergesetzt

avg.cpp

```
#include <iostream>
#include <iterator>

int main() {
   std::istream_iterator<double> it(std::cin);
   std::istream_iterator<double> end;
   unsigned int count = 0; double sum = 0;
   while (it != end) {
      sum += *it++; ++count;
   }
   std::cout << (sum / count) << std::endl;
}
```

- *std::istream_iterator* ist ein Input-Iterator, der den >>-Operator verwendet, um die einzelnen Werte des entsprechenden Typs von der Eingabe einzulesen.
- Charakteristisch ist hier der konsumierende Charakter, d.h. es ist nur ein einziger Durchlauf möglich.

Diese Iteratoren erlauben es, den Objekten einer Sequenz sukzessive neue Werte zuzuweisen oder eine Sequenz neu zu erzeugen:

Operator	Rückgabe-Typ	Beschreibung
$*it = element$	–	Zuweisung; *it* ist danach nicht notwendigerweise dereferenzierbar
$++it$	*Iterator&*	Iterator vorwärts weitersetzen
$it++$	*Iterator&*	Iterator vorwärts weitersetzen
$*it++ = element$	–	Zuweisung mit anschließendem Weitersetzen des Iterators

Zu beachten ist hier, dass Mehrfachzuweisungen auf $*it$ nicht notwendigerweise zulässig sind. Die Dereferenzierung ist nur auf der linken Seite einer Zuweisung zulässig.

inserter.cpp

```
#include <iostream>
#include <list>
#include <iterator>
#include <algorithm>
int main() {
   std::list<double> values;
   std::istream_iterator<double> input(std::cin);
   std::istream_iterator<double> input_end;
   std::back_insert_iterator<std::list<double>> output(values);
   std::copy(input, input_end, output);
   for (auto value: values) { std::cout << value << std::endl; }
}
```

- Einfüge-Iteratoren sind Output-Iteratoren, die alle ihnen übergebenen Werte in einen Container einfügen.
- Zur Verfügung stehen *std::back_insert_iterator*, *std::front_insert_iterator* und *std::insert_iterator*.
- *std*::*copy* ist ein im Standard vorgegebener Algorithmus, der zwei einen Bereich definierende Input-Iteratoren und einen Output-Iterator als Parameter erhält.

Forward-Iteratoren unterstützen alle Operationen eines Input-Iterators. Im Vergleich zu diesen ist es zulässig, die Sequenz mehrfach zu durchlaufen. Entsprechend gilt:

- Aus $it1 == it2$ folgt $++it1 == ++it2$.
- Wenn $it1 == it2$ gilt, dann sind entweder beide Iteratoren dereferenzierbar oder keiner der beiden.
- Wenn die Iteratoren $it1$ und $it2$ dereferenzierbar sind, dann gilt $it1 == it2$ genau dann, wenn $*it1$ und $*it2$ das gleiche Objekt adressieren.

Forward-Iteratoren können auch die Operationen eines Output-Iterators unterstützen. Dann sind sie schreibbar und erlauben Mehrfachzuweisungen, ansonsten erlauben sie nur Lesezugriffe (*constant iterator*).

Beispiel für einen Forward-Iterator

forward.cpp

```
#include <iostream>
#include <forward_list>
#include <iterator>
#include <algorithm>
int main() {
   std::forward_list<double> values;
   std::istream_iterator<double> input(std::cin);
   std::istream_iterator<double> input_end;
   std::copy(input, input_end, front_inserter(values));
   // move all values < 0 to the front:
   auto middle = std::partition(values.begin(), values.end(),
      [](double val) { return val < 0; });
   std::ostream_iterator<double> out(std::cout, " ");
   std::cout << "negative values: " << std::endl;
   std::copy(values.begin(), middle, out); std::cout << std::endl;
   std::cout << "positive values: " << std::endl;
   std::copy(middle, values.end(), out); std::cout << std::endl;
}
```

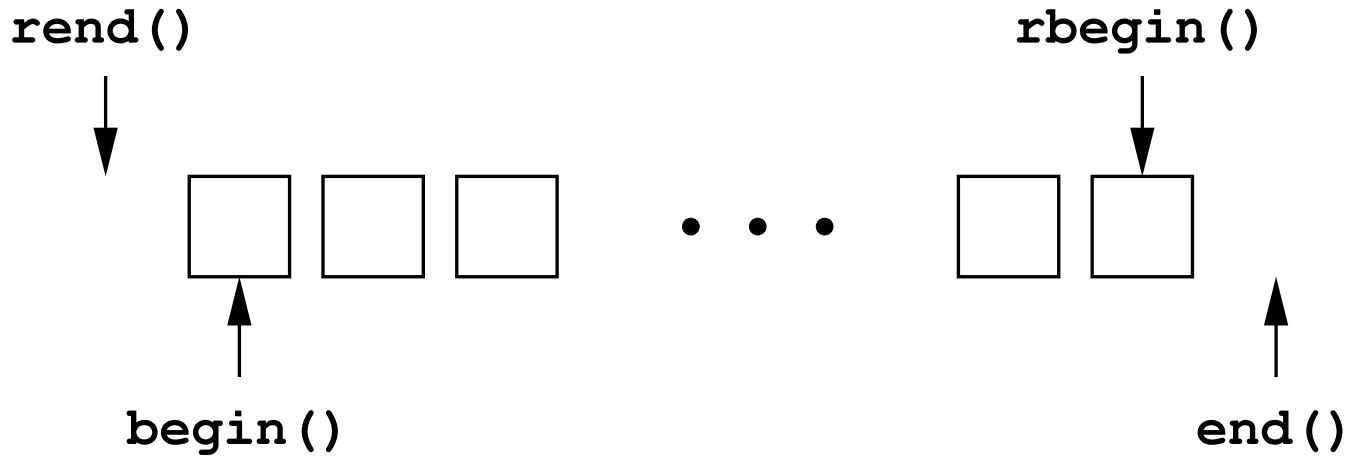
- *std*::*forward_list* ist eine einfach verkettete lineare Liste.

Bidirektionale Iteratoren sind Forward-Iteratoren, bei denen der Iterator in beide Richtungen versetzt werden kann:

Operator	Rückgabe-Typ	Beschreibung
$--it$	*Iterator*	Iterator rückwärts weitersetzen
$it--$	*Iterator*	Iterator rückwärts weitersetzen
$*it-- = element$	*Element&*	Zuweisung mit anschließendem Zurücksetzen des Iterators

Random-Access-Iteratoren sind bidirektionale Iteratoren, die die Operationen der Zeigerarithmetik unterstützen:

Operator	Rückgabe-Typ	Beschreibung
it+*n*	*Iterator*	liefert einen Iterator zurück, der *n* Schritte relativ zu *it* vorangegangen ist
it−*n*	*Iterator*	liefert einen Iterator zurück, der *n* Schritte relativ zu *it* zurückgegangen ist
it[*n*]	*Element&*	äquivalent zu $*(it+n)$
$it1 < it2$	**bool**	äquivalent zu $it2 - it1 > 0$
$it2 < it1$	**bool**	äquivalent zu $it1 - it2 > 0$
$it1 <= it2$	**bool**	äquivalent zu $!(it1 > it2)$
$it1 >= it2$	**bool**	äquivalent zu $!(it1 < it2)$
$it1 - it2$	*Distance*	Abstand zwischen *it1* und *it2*; dies liefert einen negativen Wert, falls $it1 < it2$



iterator	bidirektionaler Iterator, der sich an der Ordnung des Containers orientiert (soweit eine Ordnung existiert)
const_iterator	analog zu *iterator*, jedoch sind schreibende Zugriffe auf die referenzierten Elemente nicht möglich
reverse_iterator	bidirektionaler Iterator, dessen Richtung der Ordnung des Containers entgegen gesetzt ist
const_reverse_iterator	analog zu *reverse_iterator*, jedoch sind schreibende Zugriffe auf die referenzierten Elemente nicht möglich

- Es ist zu beachten, dass ein Iterator *in* einen Container zeigen muss, damit auf ein Element zugegriffen werden kann, d.h. die Rückgabe-Werte von *end()* und *rend()* dürfen nicht dereferenziert werden.
- Analog ist es auch nicht gestattet, Zeiger mehr als einen Schritt jenseits der Container-Grenzen zu verschieben.
- $--it$ darf bei einem *iterator* oder *const_iterator* den Container nicht verlassen, nicht einmal um einen einzelnen Schritt.

Implementierungstechnik	Name der Template-Klasse	
Lineare Listen	*list*	*forward_list*
Dynamische Arrays	*vector*	*string*
	deque	
Adapter	*stack*	*queue*
Balancierte binäre	*set*	*multiset*
sortierte Bäume	*map*	*multimap*
Hash-Verfahren	*unordered_set*	*unordered_multiset*
	unordered_map	*unordered_multimap*

- All die genannten Container-Klassen mit Ausnahme der *unordered*-Varianten besitzen eine Ordnung. Eine weitere Ausnahme sind hier noch die Template-Klassen *multiset* und *multimap*, die keine definierte Ordnung für mehrfach vorkommende Schlüssel haben.
- Die Unterstützung von Hash-Tabellen (*unordered_map* etc.) ist erst mit C++11 gekommen. Zuvor sah die Standard-Bibliothek keine Hash-Tabellen vor.
- Gelegentlich gab es früher den Standard ergänzende Bibliotheken, die dann andere Namen wie etwa *hash_set*, *hash_map* usw. hatten.

Eine gute Container-Klassenbibliothek strebt nach einer übergreifenden Einheitlichkeit, was sich auch auf die Methodennamen bezieht:

Methode	Beschreibung
begin()	liefert einen Iterator, der auf das erste Element verweist
end()	liefert einen Iterator, der hinter das letzte Element zeigt
rbegin()	liefert einen rückwärts laufenden Iterator, der auf das letzte Element verweist
rend()	liefert einen rückwärts laufenden Iterator, der vor das erste Element zeigt
empty()	ist wahr, falls der Container leer ist
size()	liefert die Zahl der Elemente
clear()	leert den Container
erase(*it*)	entfernt das Element aus dem Container, auf das *it* zeigt

Methode	Beschreibung	unterstützt von
front()	liefert das erste Element eines Containers	*vector*, *list*, *deque*
back()	liefert das letzte Element eines Containers	*vector*, *list*, *deque*
push_front()	fügt ein Element zu Beginn ein	*list*, *deque*
push_back()	hängt ein Element an das Ende an	*vector*, *list*, *deque*
pop_front()	entfernt das erste Element	*list*, *deque*
pop_back()	entfernt das letzte Element	*vector*, *list*, *deque*
[*n*]	liefert das *n*-te Element	*vector*, *deque*
at(*n*)	liefert das *n*-te Element mit Index-Überprüfung zur Laufzeit	*vector*, *deque*

Listen gibt es in zwei Varianten: *std*::*list* ist doppelt verkettet und wird überwiegend verwendet. Wenn der Speicherverbrauch minimiert werden soll, kann die einfach verkettete *std*::*forward_list* verwendet werden, die aber nicht mehr alle Vorteile der regulären Liste bietet:

Vorteile:

- Überall konstanter Aufwand beim Einfügen und Löschen. (Dies schließt nicht das Finden eines Elements in der Mitte ein.)
- Unterstützung des Zusammenlegens von Listen, des Aufteilens und des Umdrehens.

Nachteile:

- Kein indizierter Zugriff. Entsprechend ist der Suchaufwand linear.

Vorteile:

- Schneller indizierter Zugriff (theoretisch kann dies gleichziehen mit den eingebauten Arrays).
- Konstanter Aufwand für Einfüge- und Löschoperationen am Ende. (Beim Einfügen kann es aber Ausnahmen geben, siehe unten.)
- Geringerer Speicherverbrauch, weil es keinen Overhead für einzelne Elemente gibt.
- Cache-freundlich, da die Elemente des Vektors zusammenhängend im Speicher liegen.

Nachteile:

- Da der belegte Speicher zusammenhängend ist, kann eine Vergrößerung eines Vektors zu einer Umkopieraktion führen mit linearem Aufwand.
- Weder *push_front* noch *pop_front* werden unterstützt.

Vorteile:

- Erlaubt indizierten Zugriff in konstanter Zeit
- Einfüge- und Lösch-Operationen an den Enden mit konstanten Aufwand.

Nachteile:

- Einfüge- und Lösch-Operationen in der Mitte haben einen linearen Aufwand.
- Kein Aufteilen, kein Zusammenlegen (im Vergleich zu Listen).
- Erheblich erhöhter Speicheraufwand im Vergleich zu einem Vektor, da es sich letztlich um einen Vektor von Vektoren handelt.
- Der indizierte Aufwand ist zwar konstant, hat aber eine Indirektion mehr als beim Vektor.
- Eine Deque ist somit ineffizienter als ein Vektor oder eine Liste auf deren jeweiligen Paradedisziplinen. Sie ist nur sinnvoll, wenn der indizierte Zugriff und beidseitiges Einfügen und Löschen wichtig sind.

std::*queue* und *std*::*stack* basieren auf einem anderen Container-Typ (zweiter Template-Parameter, *std*::*deque* per Voreinstellung) und bieten dann nur die entsprechende Funktionalität an:

Operation	Rückgabe-Typ	Beschreibung
empty()	**bool**	liefert *true*, falls der Container leer ist
size()	*size_type*	liefert die Zahl der enthaltenen Elemente
top()	*value_type*&	liefert das letzte Element; eine **const**-Variante wird ebenfalls unterstützt
push(*element*)	**void**	fügt ein Element hinzu
pop()	**void**	entfernt ein Element

Es gibt vier sortierte assoziative Container-Klassen in der STL:

	Schlüssel/Werte-Paare	Nur Schlüssel
Eindeutige Schlüssel	*map*	*set*
Mehrfache Schlüssel	*multimap*	*multiset*

- Der Aufwand der Suche nach einem Element ist logarithmisch.
- Kandidaten für die Implementierung sind AVL-Bäume oder Red-Black-Trees.
- Voreinstellungsgemäß wird $<$ für Vergleiche verwendet, aber es können auch andere Vergleichs-Operatoren spezifiziert werden. Der $==$-Operator wird nicht verwendet. Stattdessen wird die Äquivalenzrelation von $<$ abgeleitet, d.h. a und b werden dann als äquivalent betrachtet, falls $!(a < b) \&\& !(b < a)$.
- Alle assoziativen Container haben die Eigenschaft gemeinsam, dass vorwärts laufende Iteratoren die Schlüssel in monotoner Reihenfolge entsprechend des Vergleichs-Operators durchlaufen. Im Falle von Container-Klassen, die mehrfach vorkommende Schlüssel unterstützen, ist diese Reihenfolge nicht streng monoton.

- Assoziative Container mit eindeutigen Schlüsseln akzeptieren Einfügungen nur, wenn der Schlüssel bislang noch nicht verwendet wurde.
- Im Falle von *map* und *multimap* ist jeweils ein Paar, bestehend aus einem Schlüssel und einem Wert zu liefern. Diese Paare haben den Typ *std*::*pair*<**const** *Key*, *Value*>, der dem Typ *value_type* der instanziierten Template-Klasse entspricht.
- Der gleiche Datentyp für Paare wird beim Dereferenzieren von Iteratoren bei *map* und *multimap* geliefert.
- Das erste Feld des Paares (also der Schlüssel) wird über den Feldnamen *first* angesprochen; das zweite Feld (also der Wert) ist über den Feldnamen *second* erreichbar.

- Die Template-Klasse *map* unterstützt den []-Operator, der den Datentyp für Paare vermeidet, d.h. Zuweisungen wie etwa *mymap*[*key*] = *value* sind möglich.
- Jedoch ist dabei Vorsicht geboten: Es gibt keine **const**-Variante des []-Operators und ein Zugriff auf *mymap*[*key*] führt zum Aufruf des Default-Konstruktors für das Element, wenn es bislang noch nicht existierte. Entsprechend ist der []-Operator nicht zulässig in **const**-Methoden und stattdessen erfolgt der Zugriff über einen *const_iterator*.

Methode	Beschreibung
insert(*t*)	Einfügen eines Elements: *std*::*pair*<*iterator*,**bool**> wird von *map* und *set* geliefert, wobei der Iterator auf das Element mit dem Schlüssel verweist und der **bool**-Wert angibt, ob die Einfüge-Operation erfolgreich war oder nicht. Bei *multiset* und *multimap* wird nur ein Iterator auf das neu hinzugefügte Element geliefert.
insert(*it*,*t*)	Analog zu *insert*(*t*). Falls das neu einzufügende Element sich direkt hinter *t* einfügen lässt, erfolgt die Operation mit konstantem Aufwand.
erase(*k*)	Entfernt alle Elemente mit dem angegebenen Schlüssel.
erase(*it*)	Entfernt das Element, worauf *it* zeigt.
erase(*it1*, *it2*)	Entfernt alle Elemente aus dem Bereich $[it1, it2)$.

Methode	Beschreibung
find(k)	Liefert einen Iterator, der auf ein Element mit dem gewünschten Schlüssel verweist. Falls es keinen solchen Schlüssel gibt, wird *end()* zurückgeliefert.
count(k)	Liefert die Zahl der Elemente mit einem zu k äquivalenten Schlüssel. Dies ist insbesondere bei *multimap* und *multiset* sinnvoll.
lower_bound(k)	Liefert einen Iterator, der auf das erste Element verweist, dessen Schlüssel nicht kleiner als k ist.
upper_bound(k)	Liefert einen Iterator, der auf das erste Element verweist, dessen Schlüssel größer als k ist.

Es gibt vier unsortierte assoziative Container-Klassen in der STL:

	Schlüssel/Werte-Paare	Nur Schlüssel
Eindeutige Schlüssel	*unordered_map*	*unordered_set*
Mehrfache Schlüssel	*unordered_multimap*	*unordered_multiset*

- Die unsortierten assoziativen Container werden mit Hilfe von Hash-Organisationen implementiert.
- Der Standard sichert zu, dass der Aufwand für das Suchen und das Einfügen im Durchschnittsfall konstant sind, im schlimmsten Fall aber linear sein können (wenn etwa alle Objekte den gleichen Hash-Wert haben).
- Für den Schlüsseltyp muss es eine Hash-Funktion geben und einen Operator, der auf Gleichheit testet.
- Die Größe der Bucket-Tabelle wird dynamisch angepasst. Eine Umorganisation hat linearen Aufwand.

- Für die elementaren Datentypen einschließlich der Zeigertypen darauf und einigen von der Standard-Bibliothek definierten Typen wie *std*::*string* ist eine Hash-Funktion bereits definiert.
- Bei selbst definierten Schlüsseltypen muss dies nachgeholt werden. Der Typ der Hash-Funktion muss dann als dritter Template-Parameter angegeben werden und die Hash-Funktion als weiterer Parameter beim Konstruktor.
- Hierzu können aber die bereits vordefinierten Hash-Funktionen verwendet und typischerweise mit dem „$\hat{}$“-Operator verknüpft werden.

Persons.cpp

```
struct Name {
   std::string first;
   std::string last;
   Name(const std::string first, const std::string last) :
      first(first), last(last) {
   }
   bool operator==(const Name& other) const {
      return first == other.first && last == other.last;
   }
};
```

- Damit ein Datentyp als Schlüssel für eine Hash-Organisation genutzt werden kann, müssen der „==“-Operator und eine Hash-Funktion gegeben sein.

Persons.cpp

```
auto hash = [](const Name& name) {
   return std::hash<std::string>()(name.first) ^
      std::hash<std::string>()(name.last);
};
```

- *hash*<*std*::*string*>() erzeugt ein temporäres Hash-Funktionsobjekt, das einen Funktions-Operator mit einem Parameter (vom Typ *std*::*string*) anbietet, der den Hash-Wert (Typ *size_t*) liefert.
- Hash-Werte werden am besten mit dem XOR-Operator „^“ verknüpft.
- Eine Hash-Funktion muss immer den gleichen Wert für den gleichen Schlüssel liefern.
- Für zwei verschiedene Schlüssel *k1* und *k2* sollte die Wahrscheinlichkeit, dass die entsprechenden Hash-Werte gleich sind, sich 1.0 / *numeric_limits*<*size_t*>::*max*() nähern.

Persons.cpp

```
int main() {
   auto hash = [](const Name& name) { /* ... */ };
   std::unordered_map<Name, std::string, decltype(hash)> address(32, hash);
   address[Name("Marie", "Maier")] = "Ulm";
   address[Name("Hans", "Schmidt")] = "Neu-Ulm";
   address[Name("Heike", "Vogel")] = "Geislingen";
   std::string first; std::string last;
   while (std::cin >> first >> last) {
      auto it = address.find(Name(first, last));
      if (it != address.end()) {
         std::cout << it->second << std::endl;
      } else {
         std::cout << "Not found." << std::endl;
      }
   }
}
```

- Der erste Parameter beim Konstruktor für Hash-Organisationen legt die initiale Größe der Bucket-Tabelle fest, der zweite spezifiziert die gewünschte Hash-Funktion.

- Template-Container-Klassen benutzen implizit viele Methoden und Operatoren für ihre Argument-Typen.
- Diese ergeben sich nicht aus der Klassendeklaration, sondern erst aus der Implementierung der Template-Klassenmethoden.
- Da die implizit verwendeten Methoden und Operatoren für die bei dem Template als Argument übergebenen Klassen Voraussetzung sind, damit diese verwendet werden können, wird von Template-Abhängigkeiten gesprochen.
- Da diese recht unübersichtlich sind, erlauben Test-Templateklassen wie die nun vorzustellende *TemplateTester*-Klasse eine Analyse, welche Operatoren oder Methoden wann aufgerufen werden.

TemplateTester.hpp

```
template<class BaseType>
class TemplateTester {
   public:
      TemplateTester();
      TemplateTester(const TemplateTester& orig);
      TemplateTester(const BaseType& val);
      TemplateTester(TemplateTester&& orig);
      TemplateTester(BaseType&& val);
      ~TemplateTester();

      TemplateTester& operator=(const TemplateTester& orig);
      TemplateTester& operator=(const BaseType& val);
      TemplateTester& operator=(TemplateTester&& orig);
      TemplateTester& operator=(BaseType&& val);
      bool operator<(const TemplateTester& other) const;
      bool operator<(const BaseType& val) const;
      operator BaseType() const;

   private:
      static int instanceCounter; // gives unique ids
      int id; // id of this instance
      BaseType value;
}; // class TemplateTester
```

TemplateTester.tpp

```
template<typename BaseType>
TemplateTester<BaseType>::TemplateTester() :
      id(instanceCounter++) {
   std::cerr << "TemplateTester: CREATE #" << id <<
      " (default constructor)" << std::endl;
} // default constructor
```

- Alle Methoden und Operatoren von *TemplateTester* geben Logmeldungen auf *cerr* aus.
- Die *TemplateTester*-Klasse ist selbst eine Wrapper-Template-Klasse um *BaseType* und bietet einen Konstruktor an, der einen Wert des Basistyps akzeptiert und einen dazu passenden Konvertierungs-Operator.
- Die Klassen-Variable *instanceCounter* erlaubt die Identifikation individueller Instanzen in den Logmeldungen.

Test der *list*-Klasse

TestList.cpp

```
typedef TemplateTester<int> Test;
list<Test> myList;
// put some values into the list
for (int i = 0; i < 2; ++i) {
   myList.push_back(i);
}
// iterate through the list
for (int val: myList) {
   cout << "Found " << val << " in the list." << endl;
}
```

```
TemplateTester: CREATE #0 (constructor with parameter 0)
TemplateTester: CREATE #1 (move constructor of 0)
TemplateTester: DELETE #0
TemplateTester: CREATE #2 (constructor with parameter 1)
TemplateTester: CREATE #3 (move constructor of 2)
TemplateTester: DELETE #2
TemplateTester: CONVERT #1 to 0
TemplateTester: CONVERT #3 to 1
TemplateTester: DELETE #1
TemplateTester: DELETE #3
clonmel$
```

TestVector.cpp

```
typedef TemplateTester<int> Test;
vector<Test> myVector(2);
// put some values into the vector
for (int i = 0; i < 2; ++i) {
   myVector[i] = i;
}
// print all values of the vector
for (int i = 0; i < 2; ++i) {
   cout << myVector[i] << endl;
}
```

```
clonmel$ TestVector >/dev/null
TemplateTester: CREATE #0 (default constructor)
TemplateTester: CREATE #1 (default constructor)
TemplateTester: ASSIGN value 0 to #0
TemplateTester: ASSIGN value 1 to #1
TemplateTester: CONVERT #0 to 0
TemplateTester: CONVERT #1 to 1
TemplateTester: DELETE #0
TemplateTester: DELETE #1
clonmel$
```

TestMap.cpp

```
typedef TemplateTester<int> Test;
map<int, Test> myMap;

// put some values into the map
for (int i = 0; i < 2; ++i) {
   myMap[i] = i;
}
```

```
clonmel$ TestMap >/dev/null
TemplateTester: CREATE #0 (default constructor)
TemplateTester: ASSIGN value 0 to #0
TemplateTester: CREATE #1 (default constructor)
TemplateTester: ASSIGN value 1 to #1
TemplateTester: CONVERT #0 to 0
TemplateTester: CONVERT #1 to 1
TemplateTester: DELETE #1
TemplateTester: DELETE #0
clonmel$
```

Test der Indizierung bei *map*

TestMapIndex.cpp

```
typedef TemplateTester<int> Test;
typedef map<Test, int> MyMap; MyMap myMap;
for (int i = 0; i < 2; ++i) myMap[i] = i;
for (const auto& pair: myMap) {
   cout << pair.second << endl;
}
```

```
clonmel$ TestMapIndex >/dev/null
TemplateTester: CREATE #0 (constructor with parameter 0)
TemplateTester: CREATE #1 (move constructor of 0)
TemplateTester: DELETE #0
TemplateTester: CREATE #2 (constructor with parameter 1)
TemplateTester: COMPARE #1 with #2
TemplateTester: CREATE #3 (move constructor of 2)
TemplateTester: COMPARE #1 with #3
TemplateTester: COMPARE #3 with #1
TemplateTester: DELETE #2
TemplateTester: DELETE #3
TemplateTester: DELETE #1
clonmel$
```

- Gelegentlich wird über die Verwendung von Kopierkonstruktoren oder Zuweisungen der Inhalt eines Objekts aufwendig kopiert (*deep copy*), worauf kurz danach das Original eliminiert wird.
- Das Problem tritt bei temporären Objekten auf. Da auch bei STL-Containern nicht wenig mit temporäre Objekte zum Einsatz kommen, ist dies u.U. recht teuer.
- Hier wäre es besser, wenn wir einfach die (möglicherweise sehr umfangreiche) interne Datenstruktur einfach „umhängen“ könnten.
- Beginnend mit C++11 gibt es einen weiteren Referenztyp, der auch temporäre Objekte unterstützt. Dieser Referenztyp verwendet zwei Und-Zeichen statt einem: „&&“.
- Diese Technik kommt auch bei *std*::*swap* zum Einsatz.

Trie.hpp

```
Trie(Trie&& other) :
      root(other.root), number_of_objects(other.number_of_objects) {
   // unlink tree from rvalue-referenced object other
   other.root = 0;
   other.number_of_objects = 0;
}
```

- Anders als beim Kopierkonstruktor übernimmt der Verlagerungskonstruktor den Objekt-Inhalt von dem übergebenen Objekt und initialisiert das referenzierte Objekt auf den leeren Zustand.
- Letzteres ist zwingend notwendig, da in jedem Falle anschließend noch der Dekonstruktor für das referenzierte Objekt aufgerufen wird.
- Da das referenzierte Objekt hier verändert wird, entfällt die Angabe von **const**.

Trie.hpp

```
Trie& operator=(Trie&& other) {
   delete root;
   root = other.root;
   number_of_objects = other.number_of_objects;
   other.root = 0;
   other.number_of_objects = 0;
   return *this;
}
```

- Analog kann bei der Zuweisung auch der Fall unterstützt werden, dass wir den Inhalt eines temporären Objekts übernehmen.
- Da das temporäre Objekt nicht das eigene sein kann, entfällt hier der entsprechende Test.

Trie.hpp

```
friend void swap(Trie& first, Trie& other) {
   std::swap(first.root, other.root);
   std::swap(first.number_of_objects, other.number_of_objects);
}
```

- Noch eleganter wird das alles, wenn eine *swap*-Methode eingeführt wird. Ihre Aufgabe ist es, den Inhalt der beiden Argumente auszutauschen.
- Es gibt bereits eine *std*::*swap*-Funktion, die per **#include** <utility> zur Verfügung steht.
- Für die elementaren Datentypen ist sie bereits definiert, für selbst-definierte Klassen kann sie (wie hier) als normale Funktion definiert werden, die dank der **friend**-Deklaration vollen Zugang zu den privaten Daten hat.

Trie.hpp

```
Trie(Trie&& other) : Trie() {
   swap(*this, other);
}
// ...
Trie& operator=(Trie other) {
   swap(*this, other);
   return *this;
}
```

- Wenn die passende *swap*-Funktion zur Verfügung steht, lassen sich der Verlagerungs-Konstruktor und die Zuweisung dramatisch vereinfachen.
- Die eigentlichen Aufgaben werden dann nur noch in *swap* bzw. dem Kopierkonstruktor geleistet.
- Es gibt dann nur noch einen Zuweisungsoperator, der mit einer Kopie(!) arbeitet und nicht mit Referenzen.
- Das eröffnet mehr Möglichkeiten für den C++-Optimierer. Wenn eine tiefe Kopie wirklich notwendig ist, erfolgt sie bei der Parameterübergabe, danach wird diese nicht ein weiteres Mal kopiert, sondern nur noch verlagert. Wenn keine tiefe Kopie notwendig ist, wird auch keine durchgeführt.

- Die Algorithmen der STL sind über **#include** <algorithm> zugänglich.
- Die Algorithmen arbeiten alle auf Sequenzen, die mit Iteratoren spezifiziert werden.
- Sie unterteilen sich in
  - nicht-modifizierende Algorithmen auf Sequenzen
  - Algorithmen, die Sequenzen verändern und
  - weitere Algorithmen, wie Sortieren, binäre Suche, Mengen-Operationen, Heap-Operationen und die Erzeugung von Permutationen.

Der Standard legt folgende Komplexitäten fest. Hierbei ist n normalerweise die Länge der Sequenz.

$O(1)$	*swap*(), *iter_swap*()
$O(\log n)$	*lower_bound*(), *upper_bound*(), *equal_range*(), *binary_search*(), *push_heap*(), *pop_heap*()
$O(n \log n)$	*inplace_merge*(), *stable_partition*(), *sort*(), *stable_sort*(), *partial_sort*(), *partial_sort_copy*(), *sort_heap*()
$O(n^2)$	*find_end*(), *find_first_of*(), *search*(), *search_n*()
$O(n)$	alle anderen Funktionen

(Siehe Abschnitt 25 im Standard und 32.3.1 bei Stroustrup.)