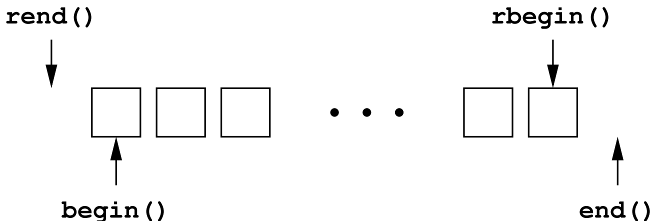
Die Standard-Template-Library (STL) bietet eine Reihe von Template-Klassen für Container, eine allgemeine Schnittstelle für Iteratoren und eine Sammlung von Algorithmen an.

Container-Klassen der STL:

Implementierungstechnik	Name der Template-Klasse	
Lineare Listen	*deque*	*queue*
	list	*stack*
Dynamische Arrays	*string*	*vector*
Balancierte binäre sortierte Bäume	*set*	*multiset*
	map	*multimap*

- All die genannten Container-Klassen besitzen eine Ordnung. Eine kleine Ausnahme sind hier die Template-Klassen *multiset* und *multimap*, die keine definierte Ordnung für mehrfach vorkommende Schlüssel haben.
- Bislang bietet der ISO-Standard für C++ keine Hash-Tabellen an (weder in der Fassung von 1999 noch der von 2003).
- Die aktuelle Arbeitsfassung sieht jedoch Hash-Tabellen vor. Die entsprechenden Template-Klassen tragen die Namen *unordered_set*, *unordered_multiset*, *unordered_map* und *unordered_multimap*.
- Andere, den Standard ergänzende Bibliotheken haben teilweise Template-Klassen mit den Namen *hash_set*, *hash_map* usw. unterstützt.



iterator	bidirektionaler Iterator, der sich an der Ordnung des Containers orientiert
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

- *iterator* und *reverse_iterator* erlauben Schreibzugriffe auf die referenzierten Elemente. Der Schlüssel ist jedoch davon ausgenommen im Falle assoziativer Container wie *set*, *map* usw.
- Keiner der Iteratoren ist *robust*, d.h. Entfernungen oder Einfügungen aus oder in den Container führen zu ungültigen Iteratoren (mehr dazu im ISO-Standard selbst).
- Es gibt noch weitere Iteratoren, die hier nicht vorgestellt werden: Unidirektionale Iteratoren, Einfüge-Iteratoren und Stream-Iteratoren.

Obwohl Iteratoren keine Zeiger im eigentlichen Sinne sind, werden die entsprechenden Operatoren weitgehend unterstützt, so dass sich Zeiger und Iteratoren in der Anwendung ähneln.
Sei *Iterator* der Typ eines Iterators, *it* ein Iterator dieses Typs und *Element* der Element-Typ des Containers, mit dem *Iterator* und *it* verbunden sind. Ferner sei *member* ein Datenfeld von *Element*.

Operator	Rückgabe-Typ	Beschreibung
$*it$	*Element&*	Zugriff auf ein Element
$it->member$	Typ von *member*	Zugriff auf ein Datenfeld
$++it$	*Iterator*	Iterator vorwärts weitersetzen
$--it$	*Iterator*	Iterator rückwärts weitersetzen

- 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 den Container nicht verlassen, nicht einmal um einen einzelnen Schritt.
- Iteratoren unterstützen Default-Konstruktoren, Kopierkonstruktoren, Zuweisungen, $==$ und $!=$.

Iteratoren, die von *vector*, *deque* und *string* geliefert werden, erlauben einen indizierten Zugriff:

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 $it1 < it2$

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*)	wirft das Element aus dem Container heraus, 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	*vector*, *deque*

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).

Vorteile:

- Überall konstanter Aufwand beim Einfügen und Löschen.
- Unterstützung des Zusammenlegens von Listen, des Aufteilens und des Umdrehens.

Nachteile:

- Kein indizierter Zugriff.

Vorteile:

- Schneller indizierter Zugriff (theoretisch kann dies gleichziehen mit den eingebauten Arrays).
- Konstanter Aufwand für Einfüge- und Löschoperationen am Ende.

Nachteile:

- Weder *push_front* noch *pop_front* werden unterstützt.

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

Bislang gibt es vier assoziative Container-Klassen in der STL:

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

- Alle bisherigen Container-Klassen der STL verwalten die Schlüssel in einer sortierten Reihenfolge.
- 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.

Zugriffs-Operatoren für assoziative Container

- 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 *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. Ja, die Namen sind unglücklich gewählt.

- 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: *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.

- 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.h

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

      // destructor
      ~TemplateTester();

      // operators
      TemplateTester& operator=(const TemplateTester& orig);
      TemplateTester& operator=(const 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.C

```
template<class BaseType>
TemplateTester<BaseType>::TemplateTester() :
      id(instanceCounter++) {
   std::cerr << "TemplateTester: CREATE #" << id <<
      " (default constructor)" << 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.

TestList.C

```
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 (list<Test>::iterator it(myList.begin());
      it != myList.end(); ++it) {
   cout << "Found " << *it << " in the list." << endl;
}
```

```
dublin$ testList >/dev/null
TemplateTester: CREATE #0 (constructor with parameter 0)
TemplateTester: CREATE #1 (copy constructor of 0)
TemplateTester: DELETE #0
TemplateTester: CREATE #2 (constructor with parameter 1)
TemplateTester: CREATE #3 (copy constructor of 2)
TemplateTester: DELETE #2
TemplateTester: CONVERT #1 to 0
TemplateTester: CONVERT #3 to 1
TemplateTester: DELETE #1
TemplateTester: DELETE #3
dublin$
```

TestVector.C

```
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;
}
```

```
dublin$ testVector >/dev/null
TemplateTester: CREATE #0 (default constructor)
TemplateTester: CREATE #1 (copy constructor of 0)
TemplateTester: CREATE #2 (copy constructor of 0)
TemplateTester: DELETE #0
TemplateTester: ASSIGN value 0 to #1
TemplateTester: ASSIGN value 1 to #2
TemplateTester: CONVERT #1 to 0
TemplateTester: CONVERT #2 to 1
TemplateTester: DELETE #1
TemplateTester: DELETE #2
dublin$
```

TestMap.C

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

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

```
TemplateTester: CREATE #0 (default constructor)
TemplateTester: CREATE #1 (copy constructor of 0)
TemplateTester: CREATE #2 (copy constructor of 1)
TemplateTester: DELETE #1
TemplateTester: DELETE #0
TemplateTester: ASSIGN value 0 to #2
TemplateTester: CREATE #3 (default constructor)
TemplateTester: CREATE #4 (copy constructor of 3)
TemplateTester: CREATE #5 (copy constructor of 4)
TemplateTester: DELETE #4
TemplateTester: DELETE #3
TemplateTester: ASSIGN value 1 to #5
```

TestMap.C

```
// print all values of the map
for (int i = 0; i < 2; ++i) {
   cout << myMap[i] << endl;
}
```

```
TemplateTester: CREATE #6 (default constructor)
TemplateTester: CREATE #7 (copy constructor of 6)
TemplateTester: DELETE #7
TemplateTester: DELETE #6
TemplateTester: CONVERT #2 to 0
TemplateTester: CREATE #8 (default constructor)
TemplateTester: CREATE #9 (copy constructor of 8)
TemplateTester: DELETE #9
TemplateTester: DELETE #8
TemplateTester: CONVERT #5 to 1
TemplateTester: DELETE #5
TemplateTester: DELETE #2
```

TestMapIndex.C

```
typedef TemplateTester<int> Test;
typedef map<Test, int> MyMap; MyMap myMap;
for (int i = 0; i < 2; ++i) myMap[i] = i;
for (MyMap::iterator it(myMap.begin());
      it != myMap.end(); ++it) {
   cout << it->second << endl;
}
```

```
TemplateTester: CREATE #0 (constructor with parameter 0)
TemplateTester: CREATE #1 (copy constructor of 0)
TemplateTester: CREATE #2 (copy constructor of 1)
TemplateTester: DELETE #1
TemplateTester: DELETE #0
TemplateTester: CREATE #3 (constructor with parameter 1)
TemplateTester: CREATE #4 (copy constructor of 3)
TemplateTester: COMPARE #4 with #2
TemplateTester: COMPARE #2 with #4
TemplateTester: CREATE #5 (copy constructor of 4)
TemplateTester: COMPARE #4 with #2
TemplateTester: DELETE #4
TemplateTester: DELETE #3
TemplateTester: DELETE #5
TemplateTester: DELETE #2
```