

# **Assoziative Container in C++**

Christian Poulter

Seminar: Die Sprache C++

Mat-Nr.: 53 03 129

E-Mail: [inf@poulter.de](mailto:inf@poulter.de)

# Inhalt

1. Einleitung: Container .....	3
2. Der Assoziative Container von C++ .....	3
3. Definition des Assoziative Containers .....	3
4. Zugriff auf Elemente .....	4
5. Zugriff auf Elemente mit Hilfe von find .....	6
6. Die Datenstruktur pair .....	7
7. Übersicht der Operatoren und Funktionen .....	8
8. Vergleich: Assoziative Container und Hashtable .....	10

# 1. Einleitung: Container

Container sind Datenstrukturen, die zur Aufnahme anderer Datenstrukturen bestimmt sind. Je nach Typ des Containers kann eine bestimmte oder beliebige Menge von Elementen aufgenommen werden.

Da für das jeweilige Element keine explizite Variable deklariert wird, muss der Zugriff entweder über einen Index, einen ausgezeichneten Schlüssel oder anhand besonderer Merkmale (z.B. das letzte Element im Container) erfolgen.

Auch können die Einfüge- bzw. Zugriffsmöglichkeiten eingeschränkt sein. Beispielsweise kann bei einem Container vom Typ *stack* nur auf das jeweils oberste, d.h. zuletzt eingefügte, Element zugegriffen werden.

Container bieten dem Programmierer so die Möglichkeit, neben den von vielen Programmiersprachen unterstützten *array*, Daten in effizienter Weise zu speichern sowie auf schnelle Such- und Zugriffsmethoden zurückzugreifen.

## 2. Der Assoziative Container von C++

In C++ werden dem Programmierer unterschiedliche vorgefertigte Container angeboten. Der Unterschied besteht meist vor allem in der Organisation, sowie den Zugriffs- bzw. Ablagemöglichkeiten der einzelnen Elemente.

Während bei den häufig genutzten *vector*- oder *listen*-Containern die Elemente direkt über die Position in einer Liste, also einem veränderlichem Index, angesprochen werden müssen, bietet ein Assoziativer Container die Möglichkeit jedes Element unter einem beliebigen, aber bestimmten Schlüssel zu speichern. Dies hat den Vorteil, dass ein Element immer unter einem vorher genau festgelegten Schlüssel zu finden und zugreifbar ist. Da sich dieser im Gegensatz zum Index einer Liste nicht verändern kann, werden aber die Sortiermöglichkeiten eingeschränkt. Die assoziativen Container sind aber von Haus aus nach ihren Schlüsseln und einem auf dem Schlüsseltyp definiertem Prädikat sortiert.

## 3. Definition des Assoziative Containers

Der Assoziative Container wird in C++ auch als *map* bezeichnet und in der Headerdatei `<map>` definiert. In vielen Fällen verhält sich die *map* wie ein normaler Container vom Typ *vector* oder *list*. So stehen z.B. Funktionen wie *push\_back* oder *insert* zur Verfügung. Natürlich ist es aber nicht möglich, die Sortiermethoden eines *vectors* o.ä. auf einen Assoziativen Container anzuwenden.

In der Deklaration eines assoziativen Containers (also einer *map*) muss der Typ der Objekte, die in diesem Container gespeichert werden sollen, festgelegt werden. Auch der Typ der zu verwendenden Schlüssel muss während der Deklaration bestimmt werden. Allerdings ist dieser nicht wie z.B. beim *array* auf einen bestimmten Typ eingeschränkt, sondern kann frei ausgewählt werden. Es ist also ohne weiteres möglich, auch Strings oder komplexere Objekte als Schlüssel zu verwenden.

Folgendes Beispielprogramm demonstriert, wie eine *map* erzeugt wird:

```
001 #include <map>
002 #include <string>
003
004 using std::map;
005 using std::string;
006
007 int main(){
008     map<string, int> zaehler;
009 }
```

Zunächst werden die benötigten Headerdatei inkludiert und der verwendete Namensraum festgelegt (Zeilen 1-5). Innerhalb der *main* – Funktion an Zeile 8 wird dann ein assoziativer Container mit dem Namen *zaehler* erzeugt. In diesen können später Elemente vom Typ *int* abgelegt werden. Der Zugriff auf die einzelnen Elemente muss in diesem Beispiel über Schlüssel vom Typ *string* verfolgen.

## **4. Zugriff auf Elemente**

Der Zugriff auf einen Assoziativen Container kann auf zwei unterschiedliche Arten erfolgen.

Zunächst können - ähnlich wie bei einem Array - die rechteckigen Klammern benutzt werden. Da Elemente grundsätzlich immer über einen Schlüssel angesprochen werden müssen, wird dieser beim Zugriff innerhalb der Klammern mit übergeben:

```
int i = zaehler[schluessel];
```

Hier wird die Integerzahl, die dem Schlüssel „*schluessel*“ zugeordnet ist, aus dem Container gesucht und in die Variable *i* abgelegt.

Sollte im Container kein Element dem gewünschten Schlüssel zugeordnet sein, wird einfach ein neues Element erzeugt und in den Container unter dem verwendeten Schlüssel abgelegt. Dabei wird das neue Element Wert-

initialisiert. Dies bedeutet, dass bei Objekten der Konstruktor des Objektes zur Initialisierung benutzt oder bei einfachen Datentypen, wie z.B. dem Integer, eine 0 angenommen wird.

Folgendes Beispiel veranschaulicht den Zugriff auf Elemente und demonstriert das Einfügen mit Hilfe des `[]`-Operators:

```
001 #include <map>
002 #include <string>
003 #include <iostream>
004
005 using std::map;
006 using std::string;
007 using std::cin;
008
009 int main(){
010     string s;
011
012     map<string, int> zaehler;
013
014     while( cin >> s )
015         ++zaehler[s];
016
017 }
```

Innerhalb der *while* – Schleife wird kontinuierlich ein *string s* von der Eingabe gelesen. Dann wird an Zeile 15 versucht die dem *string s* zugeordnete Zahl aus der *map* zu suchen. Wenn noch keine Zahl vorhanden ist, wird eine neue mit 0 initialisierte Zahl in den Container unter dem Schlüssel *s* abgelegt und gleichzeitig auch als Ergebnis des `[]`-Operators zurückgegeben. Danach folgt die Inkrementierung der gefundenen Zahl mit Hilfe des `++`-Operators.

Bei *maps*, die als *const* definiert wurden, ist kein Zugriff mit Hilfe des `[]`-Operators auf Elemente möglich, d.h. dieser Operator ist in diesem Fall gar nicht definiert. Der Hintergrund ist, dass eine als *const* deklarierte *map* nicht mehr verändert werden darf und es daher nicht möglich ist ein Element neu einzufügen, falls es nicht gefunden wurde.

## **5. Zugriff auf Elemente mit Hilfe von find**

Um aber trotzdem auf die Elemente einer *map* zugreifen zu können, gibt es eine weitere Zugriffsmöglichkeit. Die Funktion

*find( schluesselwert)*

liefert das dem *schluesselwert* zugeordnete Element aus dem assoziativen Container. Wie auch beim Zugriff über den *[]*-Operator muss der Funktion innerhalb der Klammern der Schlüssel des Elementes übergeben werden. Der Typ richtet sich hierbei natürlich nach den in der Deklaration verwendeten Eigenschaften der entsprechenden *map*.

Das Ergebnis der *find* - Funktion ist ein Iterator. Dieser zeigt auf das Element, welches dem übergebenen Schlüssel zugeordnet ist. Genau genommen zeigt der Iterator noch nicht direkt auf das Element, sondern auf eine Datenstruktur vom Typ *pair*. Auf diese wird im nächsten Abschnitt detailliert eingegangen.

Sollte kein Element gefunden werden, wird aber nicht *null* oder ein Zufallswert sondern ebenfalls ein Iterator zurückgegeben. Dieser zeigt jetzt aber nicht mehr auf Elemente des Containers, sondern hat den gleichen Wert, der auch von der Funktion *end()* zurückgegeben werden würde. Er zeigt also auf das Ende des Containers. Um zu prüfen, ob überhaupt ein gültiges Objekt gefunden wurde, sollte vor der Benutzung des Rückgabewertes dieser erst mit dem Wert von *end()* verglichen werden.

```
001 #include <map>
002 #include <string>
003 #include <iostream>
004
005 using std::map;
006 using std::string;
007 using std::cin;
008 using std::cout;
009 using std::endl;
010
011 int main(){
012     string s;
013
014     map<string, int> zaehler;
015
016     while( cin >> s )
017         ++zaehler[s];
018
019     s = "abcd";
020
021     map<string, int>::const_iterator it;
022     it = zaehler.find(s);
023
```

```

024     if ( it == zaehler.end() )
025         cout << "Kein Element gefunden." << endl;
026     else
027         cout << "Element gefunden." << endl;
028 }

```

Die Erweiterung des obigen Programms versucht nach dem Einlesen das Element, welches dem *string* „abcd“ zugeordnet ist, zu finden. Hierbei benutzt es die oben besprochene Funktion *find*. Zunächst wird die Variable *s* mit dem Suchstring initialisiert (Zeile 19) und dann ein Iterator vom Typ `map<string, int>::const_iterator` deklariert (Zeile 21), in den der Rückgabewert von *find* abgelegt wird (Zeile 22). Zum Schluss prüft die *if*-Abfrage (Zeile 24), ob auch tatsächlich ein gültiges Element gefunden wurde und gibt einen entsprechenden Text aus.

## **6. Die Datenstruktur pair**

Da in einem Assoziativen Container immer Schlüssel- und Wertpaare abgelegt werden, erhält der Programmierer bei der Dereferenzierung eines mit *find(schluesel)* bestimmten Iterators noch keinen direkten Zugriff auf das gefundene Element, sondern auf eine Datenstruktur vom Typ *pair*. Diese verfügt im wesentlichen über zwei Membervariablen: *first* und *second*.

Die beiden Variablen bieten nun den Zugriff auf den Schlüssel (*first*) bzw. den Wert (*second*) an. Natürlich konnten die Datentypen der beiden Membervariablen noch nicht im voraus festgelegt werden, sondern hängen vielmehr von der Deklaration der verwendeten *map* ab.

Ein Variable vom Typ *pair* wird ähnlich wie die *map* deklariert. In den spitzen Klammern werden der jeweilige Typ des Schlüssels und des Wertes angegeben. Diese müssen natürlich mit den Typen der verwendeten *map* übereinstimmen. Der Schlüssel kann aber nur als *const* definiert werden, um nachträgliche Änderungen des Schlüssels eines Elementes zu verhindern und den selbst sortieren Charakter der *map* nicht zu untergraben.

```

001 #include <map>
002 #include <string>
003 #include <iostream>
004
005 using std::map;
006 using std::string;
007 using std::cin;
008 using std::cout;
009 using std::endl;
010 using std::pair;

```

```

011
012  int main(){
013     string s;
014
015     map<string, int> zaehler;
016
017     while( cin >> s )
018         ++zaehler[s];
019
020     s = "abcd";
021
022     map<string, int>::const_iterator it;
023     it = zaehler.find( s );
024
025     pair<const string, int> element;
026     element = *it;
027
028     cout << "Erstes : " << element.first << endl;
029     cout << "Zweites: " << element.second << endl;
030
031 }

```

Das Beispielprogramm sucht zunächst wieder einen Iterator für einen bestimmten Schlüssel und legt diesen in *it* ab (Zeile 20-23). Dann folgt die Deklaration der Variable *element* vom Typ *pair* als *int* über einen *string* in Zeile 25. In der folgenden Zeile wird der Iterator dereferenziert und dann das *pair*-Objekt in die Variable *element* abgelegt. Zeile 28 und 29 enthalten dann die Ausgabe des Schlüssels und des Wertes mit Hilfe der Membervariablen *first* und *second*.

## **7. Übersicht der Operatoren und Funktionen**

### [k]-Operator

Liefert das Element, welches dem in den Klammern übergebenen Schlüssel *k* zugeordnet ist. Fügt ggf. ein neues wert-initialisiertes Element in den Container ein.

### count(k)

Zählt die Anzahl der Elemente mit dem Schlüssel *k*.

### find(k)

Liefert einen Iterator auf das Element mit dem Schlüssel *k* oder *end()* falls kein Element vorhanden ist.

`insert(k)`

Fügt ein neues Element *k* ein.

`insert (pos,k)`

Fügt ein neues Element *k* an der Position *pos* ein.

`insert (first, last)`

Fügt alle Elemente zwischen den Iteratoren *first* und *last* in den Container ein.

`erase(k)`

Löscht das Element mit dem Schlüssel *k*.

`erase (pos)`

Löscht das Element an der Position *pos*.

`erase (first, last)`

Löscht alle Elemente zwischen den Iteratoren *first* und *last*.

`lower_bound(k)`

Liefert einen Iterator auf die erste Position an der das Element mit dem Schlüssel *k* eingefügt werden könnte.

`upper_bound(k)`

Liefert einen Iterator auf die letzte Position an der das Element mit dem Schlüssel *k* eingefügt werden könnte.

`equal_range(k)`

Liefert eine *pair* von zwei Iteratoren, die die erste und letzte Position angeben, an denen das Element mit dem Schlüssel *k* eingefügt werden könnte.

`begin()`

Liefert einen Iterator auf das erste Element des Containers.

`end()`

Liefert einen Iterator hinter das letzte Element des Containers.

`empty()`

Ist wahr, wenn der Container leer ist.

`size()`

Aktuelle Größe des Containers.

`clear()`

Löscht alle Elemente des Containers.

## **8. Vergleich: Assoziative Container und Hashtable**

C++ bietet im Gegensatz zu anderen Programmiersprachen noch keinen vordefinierten Typ *Hashtable* an. Zwar ist in vielen Implementation dieser trotzdem bereits vorhanden, er gehört aber nicht zum allgemeinen C++ Standard.

*Hashtables* bieten den Vorteil den Schlüssel über eine *Hashfunktion* bestimmen zu können. Dies kann bei guten *Hashfunktionen* enorme Leistungssteigerungen beim Suchen, Einfügen und Löschen mit sich bringen. Allerdings muss diese Funktion vom Programmierer selbst entwickelt werden, was je nach Aufgabe einige Zeit in Anspruch nehmen kann. Zudem ist die Geschwindigkeit des *Hashtables* abhängig von der Qualität der *Hashfunktion*.

In Assoziativen Containern werden die Elemente hingegen grundsätzlich immer nach einem vorher festgelegten Vergleichsprädikat organisiert. In der Regel wird hier einfach auf die Ordnungsrelationen  $>$  bzw.  $<$  zurückgegriffen, welche die Ordnung zwischen zwei Elemente bestimmen. Zwei Element sind dann gleich, wenn weder  $>$  noch  $<$  zutrifft. Es steht dem Programmierer aber auch frei ein eigenes Prädikat zu implementieren und dieses zum Vergleich zweier Elemente zu verwenden.

Häufig werden die assoziativen Containern mit Hilfe von selbst sortierenden Bäumen implementiert. Hierdurch werden geringe Zugriffszeiten bei der Suche nach Elementen erreicht. Sie beträgt für alle Zugriffe einen Wert von  $\log(n)$  bei  $n$  Elementen. Allerdings hat die Baumorganisation auch Nachteile. Vor allem beim Einfügen und Löschen von Elementen, insbesondere für Container, die sehr viele Elemente enthalten, kann es zu Leistungseinbrüchen kommen.