# 28 Container

Dieses Kapitel behandelt die folgenden Themen:

- Eigenschaften der C++-Container
- vector, list, deque, stack und queue
- Sortierte assoziative Container
- Hash-Container

Ein Container ist ein Objekt, das der Verwaltung anderer Objekte dient, die hier Elemente des Containers genannt werden. Die mit Containern arbeitenden Algorithmen verlassen sich auf eine definierte Schnittstelle von Datentypen und Methoden. Tabelle 28.1 zeigt eine Übersicht der verschiedenen Container der C++-Standardbibliothek.

### Sequenzen

bedeutet, dass die Elemente im Container eine lineare Ordnung haben. `list`, `stack` und `vector` sind aus vorhergehenden Kapiteln bekannt. `array` ist eine Hüllklasse (englisch *wrapper*) für C-Arrays. `deque` ist eine Abkürzung für »double ended queue«, also eine Datenstruktur für eine Warteschlange, aber mit Zugriff von zwei Seiten (Anfang und Ende).

**Tabelle 28.1:** Container-Übersicht

| Container-Art | Header | Klassen-Template |
|---|---|---|
| Sequenzen | `<array>`<br>`<deque>`<br>`<list>`<br>`<queue>`<br>`<stack>`<br>`<vector>` | `array`<br>`deque`<br>`list`<br>`queue`<br>`stack`<br>`vector` |
| Sortierte Assoziative Container | `<map>`<br><br>`<set>` | `map`<br>`multimap`<br>`set`<br>`multiset` |
| Ungeordnete Assoziative Container | `<unordered_map>`<br><br>`<unordered_set>` | `unordered_map`<br>`unordered_multimap`<br>`unordered_set`<br>`unordered_multiset` |
| Spezialfälle | `<bitset>`<br>`<vector>`<br>`<queue>` | `bitset`<br>`vector<bool>`<br>`priority_queue` |



## Assoziative Container

erlauben den schnellen Zugriff auf Daten anhand eines Schlüssels. Es gibt zwei verschiedene Konzepte:

1. Eine Abbildung (englisch *map*) beschreibt eine Beziehung zwischen Elementen zweier Mengen. `map`-Container unterstützen dieses Konzept. So könnte die englische Bezeichnung für ein Buch durch Übergabe des Strings »Buch« als Schlüssel an eine geeignete Methode des Containers ermittelt werden. In diesem Fall ist dem Schlüssel »Buch« das Datum »book« zugeordnet. Es wird also eine Menge von Schlüsseln auf eine Menge von zugeordneten Daten abgebildet. Die Elemente eines solchen Containers sind Paare von Schlüsseln und Daten. Der Typ `map` kennzeichnet eine eindeutige Abbildung, weil ein Schlüssel genau *einem* Datum zugeordnet ist (Ausnahme: `multimap`).
2. Eine Menge (englisch *set*) ist eine Ansammlung unterscheidbarer Objekte, Elemente genannt, die gemeinsame Eigenschaften haben. $\mathcal{N} = \{0; 1; 2; 3; ...\}$ bezeichnet zum Beispiel die Menge der natürlichen Zahlen. `set`-Container unterstützen die Umsetzung dieses Konzepts im Computer. Weil die Elemente unterscheidbar sein müssen, kann es keine zwei gleichen Elemente in einer Menge geben (Ausnahme: `multiset`).

C++ unterscheidet zusätzlich – je nach den zugrunde liegenden Datenstrukturen – zwei verschiedene Arten:

- **Sortierte Assoziative Container**
  In der zugrunde liegenden Datenstruktur liegen die Elemente sortiert vor, weil sie bereits beim Einfügen richtig einsortiert werden. Beim GNU C++-Compiler (und vermutlich auch allen anderen) ist die Datenstruktur ein sogenannter Rot-Schwarz-Baum [CLR], eine Variante des binären Suchbaums.

- **Ungeordnete Assoziative Container**
  Diese Art von Containern bietet sich an, wenn eine Sortierung nicht erforderlich ist. Die zugrunde liegende Datenstruktur basiert auf einem Streuspeicher-Verfahren. Dabei werden die Elemente nach einem (Pseudo-) Zufallsprinzip auf den Speicherbereich »verstreut« (englisch *hashing*). Besonderes Merkmal: Die Zugriffszeit zu einem Element ist bei geeigneter Auslegung von Speicher und Zufallsfunktion konstant, also *unabhängig* von der Anzahl der gespeicherten Elemente.



# 28.1 Gemeinsame Eigenschaften

Ehe ich auf die einzelnen Container-Klassen eingehe, beschreibe ich in diesem Abschnitt gemeinsame Eigenschaften. Tabelle 28.2 zeigt die von den Containern der C++-Standardbibliothek zur Verfügung gestellten und von selbst gebauten Containern zu fordernden Datentypen eines Containers. Dabei ist `X` der Datentyp des Containers, zum Beispiel `vector<int>`, und `T` der Datentyp eines Container-Elements, zum Beispiel `int`. Der Typ `vector<int>::value_type` ist dann identisch mit `int`. Diese Datentypen sind in allen Container-Klassen der Standardbibliothek enthalten. Mit `const_reference` und `const_iterator` können Container-Elemente nicht verändert werden, nur ein lesender Zugriff ist möglich.

Jeder Container stellt einen öffentlichen Satz von Methoden zur Verfügung, die in einem Programm eingesetzt werden können. Die Methoden `begin()` und `end()` für Vektoren wurden bereits erwähnt (Seite 399). Tabelle 28.3 zeigt die von den Containern der C++-Standardbibliothek zur Verfügung gestellten Methoden. `X` sei wieder die Bezeichnung des Container-Typs. Diese Methoden sind in allen Container-Klassen der Standardbibliothek enthalten.

**Tabelle 28.2:** Container-Datentypen

| Datentyp | Bedeutung |
|---|---|
| `X::value_type` | Typ der gespeicherten Elemente |
| `X::reference` | Referenz auf Container-Element |
| `X::const_reference` | dito, aber nur lesend verwendbar |
| `X::iterator` | Iterator |
| `X::const_iterator` | nur lesend verwendbarer Iterator |
| `X::difference_type` | vorzeichenbehafteter integraler Typ |
| `X::size_type` | integraler Typ ohne Vorzeichen für Größenangaben |

`X` = Typ des Containers, zum Beispiel `vector`

Die Funktion `swap()` ist sehr schnell, weil intern nur die Verweise auf den Speicherbereich vertauscht werden. Ausnahme ist das Klassen-Template `array`, weil es die Daten direkt kapselt. Die für `swap()` benötigte Zeitdauer ist proportional zur Anzahl der Elemente des `array`-Objekts.

**Tabelle 28.3:** Container-Methoden

| Rückgabetyp Methode | Bedeutung |
|---|---|
| `X()` | Standardkonstruktor; erzeugt leeren Container |
| `X(const X&)` | Kopierkonstruktor |
| `X(X&&)` | bewegender Konstruktor für temporäre Parameter (R-Werte) |
| `~X()` | Destruktor; ruft die Destruktoren aller Elemente des Containers auf |
| `X(il)` | `il` ist eine Initialisierungsliste, siehe Seite 767. |
| `iterator begin()` | gibt Iterator zurück, der auf das erste Element des Containers, falls vorhanden, verweist; andernfalls wird `end()` zurückgegeben |
| `const_iterator begin()` | dito, aber mit diesem Iterator kann das Element nicht verändert werden |
| `const_iterator cbegin()` | dito (siehe Text) |
| `iterator end()` | Position *nach* dem letzten Element |
| `const_iterator end()` | dito |
| `const_iterator cend()` | dito (siehe Text) |
| `size_type size()` | Größe des Containers = Anzahl der aktuell gespeicherten Elemente = `(end() - begin())` |
| `size_type max_size()` | maximal mögliche Größe des Containers |
| `bool empty()` | `(size() == 0)` bzw. `(begin() == end())` |
| `void swap(X&)` | Vertauschen mit Argument-Container |
| `X& operator=(const X&)` | Zuweisungsoperator |

`X` = Typ des Containers, zum Beispiel `vector`

### begin(), end(), cbegin(), cend()

Betrachten Sie die folgende Schleife zur Ausgabe einer Liste:

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

Mit der Abkürzung `auto` wird dem Compiler die Ermittlung des Iteratortyps überlassen, siehe Seite 90. Falls `container` eine `const`-Referenz auf einen Container ist, geben `begin()` und `end()` einen `const_iterator` zurück, andernfalls einen `iterator`. Es kann aber sein, dass `container` nicht `const` ist und dennoch wie in unserem Beispiel im Schleifenkörper nur lesende Operationen ausgeführt werden sollen. Dann empfehlen sich die neu in den Standardentwurf aufgenommenen Methoden `cbegin()` und `cend()`. Falls dann versehentlich eine verändernde Anweisung wie `*it = wert;` im Schleifenkörper stände, würde schon der Compiler einen Fehler melden. Die Alternative ist also

```
for(auto it = container.cbegin();
        it != container.cend(); ++it) {
   // Der Compiler erlaubt nur lesenden Zugriff.
   cout << *it << endl;
}
```

### Kurzform für for-Schleifen

Es ist mühsam, jedesmal `begin()` und `end()` schreiben zu müssen, um über einen Container zu iterieren. Der neue C++-Standard[1] erlaubt es, mit einer `for`-Schleife über einen ganzen Bereich zu iterieren. Dabei kann für den Bereich alles angegeben werden, wofür man `begin()` und `end()` definieren würde, also Container, Strings u.a. Es geht also noch besser, weil erheblich kürzer und angenehmer zu schreiben und zu lesen. Das obige Beispiel mit der Kurzform formuliert:

```
for(auto wert : container) {
    cout << wert << endl;
}
```

Man kann die Schleife so lesen: Führe den Schleifenkörper für alle Objekte `wert` in `container` aus. Es sind auch ändernde Operationen möglich, wenn eine Referenz angegeben wird:

```
for(double& wert : v) {
   wert = 0.0;           // setzt alle Werte von v auf 0.0
}
```

### Relationale Operatoren

Für alle Container mit Ausnahme der `priority_queue` und der ungeordneten assoziativen Container gibt es die bekannten relationalen Operatoren `==`, `!=`, `<`, `<=`, `>` und `>=`, sodass zwei Container miteinander verglichen werden können. Das Ergebnis basiert auf einem lexikografischen Vergleich. Voraussetzung ist natürlich die Vergleichbarkeit der in den Containern gespeicherten Elemente.

### Schreibweisen

Dieser Abschnitt beschreibt einige Konventionen zur Schreibweise, die für alle folgenden Container-Beschreibungen gelten.

`p` und `q` sind dereferenzierbare Iteratoren desselben Containers (Typ `iterator`).

Oft müssen Bereiche angegeben werden. Dafür wird die in der Mathematik übliche Notation für Intervalle verwendet. Eckige Klammern bezeichnen dabei Intervalle einschließlich der Grenzwerte, runde Klammern Intervalle ausschließlich der Grenzwerte. Im folgenden Text ist `[i, j)` also ein Intervall einschließlich `i` und ausschließlich `j`.

`i` und `j` sind vom Typ eines Input-Iterators. Mit `InputIterator` ist ein Iterator gemeint, der zum Lesen verwendet wird und der zu einem anderen Container, auch unterschiedlichen Typs, gehört.

Ein Parameter `n` ist von dem integralen Typ `size_type` des Containers. Ein Argument `t` ist ein Element des Container-Typs `value_type` (identisch mit dem Template-Parameter `T`).

## 28.1.1 Initialisierungslisten

Initialisierungslisten sind von den C-Arrays bekannt (Seite 192), und auch von der Initialisierung von Objekten (Seite 155). Das Konzept wurde erheblich ausgedehnt, um es für STL-Container nutzbar zu machen, zum Beispiel sind die Anweisungen

[1] Wird vom GNU C++-Compiler erst ab Version 4.6 unterstützt.

```
vector<string> vs = {"gute", "Idee"};
vector<int>  vi = {1, 2, 3};
vi.insert(vi.end(), {4, 5, 6});
```

damit möglich. Zur Realisierung wurde ein eigener Typ `initializer_list` entworfen, mit dem Konstruktoren entworfen werden können, zum Beispiel (Zitat aus [ISOC++], Abschnitt 8.5.4):

```
struct S {
   S(std::initializer_list<double>); // #1
   S(std::initializer_list<int>); // #2
   // ...
};
S s1 = { 1.0, 2.0, 3.0 };          // invoke #1
S s2 = { 1, 2, 3 };                // invoke #2
```

Es kann eine beliebige Anzahl von Elementen übergeben werden – für einen Container also ideal.

### 28.1.2 Konstruktion an Ort und Stelle

Eine weitere Neuerung ist das Einfügen mit der Methode `emplace()`, die eine Konstruktion an Ort und Stelle bewirkt, um eine Kopie zu vermeiden. `emplace()` entspricht insofern grob dem Placement-new aus Abschnitt 33.2. Der Vergleich mit der Vektor-Methode `push_back()` zeigt den Effekt.

```
// cppbuch/k28/emplace/vec.cpp
#include<vector>
class A {
public:
   A(int i, int j) : i_(i), j_(j) { }
private:
   int i_;
   int j_;
};
using namespace std;

int main() {
   vector<A> v;
   A a1(1,2);
   v.push_back(a1);          // echte Kopie (default-Kopierkonstruktor)
   // v.push_back(3, 4); nicht erlaubt!
   v.push_back({3, 4});      // Initialisierunsliste: erlaubt (entspricht A{3, 4})
   v.emplace(v.end(), 3, 4); // nur Konstruktoraufruf A(3, 4),
                             // kein nachfolgender Aufruf des Kopierkonstruktors
}
```

### 28.1.3 Reversible Container

Eine einfach verkettete Liste kann nur in einer Richtung durchlaufen werden, eine doppelt verkettete Liste in zwei Richtungen, nämlich auch vom Ende zum Anfang. Alle Container mit dieser Eigenschaft heißen Reversible Container. Die Iteratoren für solche

Container sind bidirektional, das heißt, sie können mit dem ++-Operator einen Schritt vorwärts und mit dem ---Operator einen Schritt rückwärts gehen.

Für solche Container werden zusätzliche Iteratoren bereitgestellt, die vom Ende her mit dem ++-Operator bis an den Anfang laufen können. Diese Iteratoren haben den Typ `reverse_iterator` und können mit den entsprechenden Methoden ermittelt werden:

```
const_reverse_iterator rbegin() const,
const_reverse_iterator crbegin() const   und
      reverse_iterator rbegin()
```

geben einen Reverse-Iterator zurück, der auf das letzte Element zeigt.

```
const_reverse_iterator rend() const,
const_reverse_iterator crend() const   und
      reverse_iterator rend()
```

geben einen Reverse-Iterator zurück, der auf eine fiktive Position vor dem ersten Element zeigt. Anwendungsbeispiel für einen Container `container` des Typs `vector<int>`:

```
// Mit Hilfe eines Iterators Daten ausgeben, dabei am Ende beginnen:
vector<int>::reverse_iterator iter = container.rbegin();
while(iter != container.rend()) {
  cout << (*iter) << endl;
  ++iter;
}
```

# 28.2 Sequenzen

Die grundlegenden Sequenz-Typen sind `list`, `vector` und `deque`. Ein `stack`-Objekt kann mit einem dieser Typen gebildet werden. Eine Liste (`list`) ist empfehlenswert, wenn oft Elemente in der Mitte eingefügt oder gelöscht werden sollen. Eine »double ended queue« (`deque`) sollte man nehmen, wenn Einfügen und Löschen am Anfang oder Ende gefragt sind. Wenn keine besonderen Anforderungen vorliegen, ist ein `vector` die richtige Wahl. Zusätzlich zu den Methoden der Tabelle 28.3 gelten für Sequenzen die in der Tabelle 28.4 aufgeführten Methoden.

**Tipp**

`iterator`-Rückgabewerte der Methoden nutzen! Grund: Die übergebenen Iteratoren und Iteratoren auf nachfolgende Elemente werden ungültig.

Die Zeitkomplexität (siehe Glossar Seite 957) der Methoden hängt von der Art der Sequenz ab. So ist die Zeitkomplexität des Einfügens eines Elements in die Mitte eines Vektors $O(n)$, weil im Mittel $n/2$ Elemente verschoben werden. Der Zugriff auf ein Element in der Mitte mit dem `[]`-Operator ist dagegen schnell ($O(1)$). Bei einer Liste ist es gerade umgekehrt: Einfügen geht schnell ($O(1)$), aber der Zugriff auf ein Element in der Mitte dauert ($O(n)$), weil die Liste durchlaufen werden muss, um das Element zu finden.

**Tabelle 28.4:** Sequenz-Methoden

| Rückgabetyp Methode | Bedeutung |
|---|---|
| `X(n, t)` | Erzeugt eine Sequenz mit `n` Kopien von `t`. |
| `X(i, j)` | Erzeugt eine Sequenz aus den Elementen des Intervalls `[i,j)`. |
| `iterator emplace(p, args)` | Fügt ein Objekt, das mit den Parametern `args` konstruiert wird, vor `p` ein (siehe Seite 768). Der zurückgegebene Iterator zeigt auf die eingefügte Kopie. |
| `iterator insert(p, t)` | Fügt eine Kopie von `t` vor `p` ein. Der zurückgegebene Iterator zeigt auf die eingefügte Kopie. |
| `void insert(p, n, t)` | Fügt `n` Kopien von `t` vor `p` ein. |
| `void insert(p, i, j)` | Fügt Kopien der Elemente im Bereich `[i,j)` vor `p` ein. Die Iteratoren `i` und `j` dürfen nicht in den aufnehmenden Container verweisen. |
| `void insert(p, il)` | `il` ist eine Initialisierungsliste. |
| `iterator erase(q)` | Löscht das Element, auf das `q` zeigt. Der zurückgegebene Iterator zeigt anschließend auf das `q` folgende Element, wenn es existiert, ansonsten wird ein End-Iterator (`end()`) zurückgegeben. |
| `iterator erase(p, q)` | Löscht alle Elemente des Bereichs `[p,q)`. Der zurückgegebene Iterator zeigt anschließend auf das `q` folgende Element, wenn es existiert, ansonsten wird ein End-Iterator zurückgegeben. |
| `void clear()` | Löscht alle Elemente. |
| `void assign(i, j)` | Alle Elemente löschen, danach Kopie von `[i,j)` einfügen. `i` und `j` dürfen nicht in den Container verweisen. |
| `void assign(il)` | `il` ist eine Initialisierungsliste. |
| `void assign(n, t)` | Alle Elemente löschen, danach `n` Kopien von `t` einfügen. Dabei darf `t` keine Referenz auf ein Element des Containers sein. |

`X` = `array`, `deque`, `list`, `vector`



## 28.2.1 vector

Ein `vector`, eingebunden durch den Header `<vector>`, stellt öffentliche Typen und Methoden zur Verfügung, die nachfolgend kurz beschrieben werden. Vorab sei auf die schon genannten vorhandenen Datentypen und Methoden hingewiesen, um sie nicht zu wiederholen:

- Datentypen der Tabelle 28.2 (Seite 765)
- Konstruktoren und Methoden der Tabelle 28.3 (Seite 766)
- relationale Operatoren (Seite 767)
- Datentypen/Methoden des Abschnitts 28.1.3 (Reversible Container, Seite 768)
- Konstruktoren und Methoden der Tabelle 28.4 (Sequenzen, Seite 770)

Ein Vektor hat noch weitere öffentliche Datentypen, die in der Tabelle 28.5 aufgelistet sind. Die Deklaration der Klasse ist `template<typename T> class vector;`

**Tabelle 28.5:** Zusätzliche Datentypen für `vector`

| Datentyp | Bedeutung |
|---|---|
| `pointer` | Zeiger auf Vektor-Element |
| `const_pointer` | dito, aber nur lesend verwendbar |

Die folgende Aufstellung zeigt die Methoden eines Vektors, die zusätzlich zu denen, auf die am Abschnittsanfang verwiesen wird, benutzbar sind. Die Zeitkomplexität ist $O(1)$, wenn nicht anders angegeben.

- `const_reference front() const` und
  `reference front()`
  liefern eine Referenz auf das erste Element.
- `const_reference back() const` und
  `reference back()`
  liefern eine Referenz auf das letzte Element.
- `const_pointer data() const` und
  `pointer data()`
  liefern einen Zeiger auf das erste Element. Es gilt also `data() == &front()`, falls der Vektor nicht-leer ist. Der Zeiger erlaubt den Zugriff auf die Daten des Vektors wie bei einem C-Array.
- `const_reference operator[](size_type n) const` und
  `reference operator[](size_type n)`
  geben eine Referenz auf das n-te Element zurück.
- `const_reference at(size_type n) const` und
  `reference at(size_type n)`
  geben eine Referenz auf das n-te Element zurück. Der Unterschied zum vorhergehenden `operator[]()` besteht in der Prüfung, ob `n` in einem gültigen Bereich liegt. Falls nicht, d.h. `n` ist $\geq$ `size()`, wird eine `out_of_range`-Exception geworfen.
- `void emplace_back(args)` fügt ein Objekt, dessen Konstruktor mit den Parametern `args` aufgerufen wird, am Ende ein.
- `void push_back(const T& t)` fügt `t` am Ende ein.
  (`void push_back(T&& t)` für R-Werte)
- `void pop_back()`
  löscht das letzte Element.
- `void resize(size_type n, T t = T())`
  Vektorgröße ändern. Dabei werden `n - size()` Elemente `t` am Ende hinzugefügt bzw. `size() - n` Elemente am Ende gelöscht, je nachdem, ob `n` kleiner oder größer als die aktuelle Größe ist. Die Zeitkomplexität ist $O(|\ size() - n\ |)$.
- `void reserve(size_type n)`
  Speicherplatz reservieren, sodass der verfügbare Platz (Kapazität) größer als der aktuell benötigte ist. Zweck: Vermeiden von Speicherbeschaffungsoperationen während der Benutzung des Vektors. Die Zeitkomplexität ist $O(n)$.
- `size_type capacity() const`
  gibt den Wert der Kapazität zurück (siehe `reserve()`). `size()` ist immer kleiner oder gleich `capacity()`.

### 28.2.2 vector<bool>

Ein Bitset (Seite 801) alloziert festen Speicher, ein Vektor kann seinen Speicher dynamisch ändern. Falls dies auch für die Speicherung von Bits gewünscht ist und gleichzeitig nicht ein Byte pro Bit verschwendet werden soll, bietet sich eine Spezialisierung der

Klasse `vector` an: die Klasse `vector<bool>`. Sie hat dieselben öffentlichen Datentypen wie die Vektor-Klasse mit Ausnahme des Datentyps `reference`, der für Manipulationen an einzelnen Bits gedacht ist:

```
// Typ vector<bool>::reference
class reference {
   friend class vector;
   reference();
 public:
  ~reference();
   reference& operator=(const bool x);  // für b[i] = x;
   reference& operator=(const reference&);  // für b[i] = b[j];
   operator bool() const;               // für x = b[i];
   void flip();                         // für b[i].flip();
};
```

Ein `bool`-Vektor hat die Methoden eines Vektors `vector<T>`, wenn anstelle des Platzhalters `T` für den Typ `bool` eingesetzt wird. Die Spezialisierung `vector<bool>` bietet zusätzlich die Methode `void flip()` an, die alle Elemente negiert. Das folgende kleine Programm gibt *false true false false* aus:

**Listing 28.1:** Bool-Vektor

```
// cppbuch/k28/vector/vectorbool.cpp
#include<vector>
#include<iostream>
#include<showSequence.h> // siehe Seite 648
using namespace std;

int main() {
   vector<bool> vecBool(4, true); // alles true
   vecBool.flip();              // alles false
   vecBool[1].flip();           // Bit 1 wird true
   cout.setf(ios_base::boolalpha);
   showSequence(vecBool);
}
```

**Hinweis**

`vector<bool>` speichert die Werte als ein Bit pro `bool` ab, also sehr kompakt, und verhält sich daher nicht ganz wie andere Vektoren. So kann nicht die Adresse eines Elements genommen werden. Wenn die Anzahl der Bits nicht veränderbar sein muss, empfiehlt sich `bitset` als Alternative (Seite 801).

### 28.2.3 list

Die Liste dieses Abschnitts ist eine doppelt-verkettete Liste, die das Hinzufügen und Entnehmen von Elementen an jeder Stelle mit konstanter Zeit erlaubt, also unabhängig von der Zahl bereits gespeicherter Elemente. Die Klasse `list`, eingebunden durch den Header `<list>`, stellt öffentliche Typen und Methoden zur Verfügung, die nachfolgend kurz beschrieben werden. Vorab sei auf die schon genannten vorhandenen Datentypen und Methoden hingewiesen, um sie hier nicht im Einzelnen zu wiederholen:

- Datentypen der Tabelle 28.2 (Seite 765)
- Konstruktoren und Methoden der Tabelle 28.3 (Seite 766)
- relationale Operatoren (Seite 767)
- Datentypen/Methoden des Abschnitts 28.1.3 (Reversible Container, Seite 768)
- Konstruktoren und Methoden der Tabelle 28.4 (Sequenzen, Seite 770)

Eine Liste hat wie ein Vektor noch die Datentypen `pointer` und `const_pointer` für Zeiger auf Listenelemente (siehe Tabelle 28.5). Die Deklaration der Klasse ist

```
template<typename T> class list;
```

Die folgende Aufstellung zeigt die Methoden einer Liste, die zusätzlich zu den oben erwähnten benutzbar sind. Die Zeitkomplexität ist $O(1)$, wenn nicht anders angegeben. Das $n$ in $O(n)$ ist die Anzahl der Elemente, d.h. identisch mit dem Wert von `size()`.

- `const_reference front() const` und
  `reference front()`
  liefern eine Referenz auf das erste Element einer Liste.
- `const_reference back() const` und
  `reference back()`
  liefern eine Referenz auf das letzte Element einer Liste.
- `void emplace_front(args)` fügt ein Objekt, dessen Konstruktor mit den Parametern `args` aufgerufen wird, am Anfang ein.
- `void emplace_back(args)` fügt ein Objekt, dessen Konstruktor mit den Parametern `args` aufgerufen wird, am Ende ein.
- `void push_front(const T& t)` fügt `t` am Anfang ein. (`void push_front(T&& t)` für R-Werte)
- `void pop_front()` löscht das erste Element.
- `void push_back(const T& t)` fügt `t` am Ende ein.
  (`void push_back(T&& t)` für R-Werte)
- `void pop_back()`
  löscht das letzte Element.
- `void remove(const T& t)`
  entfernt alle Elemente, die gleich dem übergebenen Element `t` sind. Die Zeitkomplexität ist $O(n)$.
- `template<class Praedikat> void remove_if(Praedikat P)`
  entfernt alle Elemente, auf die das Prädikat zutrifft. Die Zeitkomplexität ist $O(n)$.
- `void resize(size_type neu, T t = T())`
  Listengröße ändern. Dabei werden `neu - size()` Elemente `t` am Ende hinzugefügt bzw. `size() - neu` Elemente am Ende gelöscht, je nachdem, ob `neu` kleiner oder größer als die aktuelle Größe ist. Die Zeitkomplexität ist $O(|\ n - neu\ |)$.
- `void reverse()`
  kehrt die Reihenfolge der Elemente in der Liste um (Zeitkomplexität $O(n)$).
- `void sort()`
  sortiert die Elemente in der Liste. Die Zeitkomplexität ist $O(n\log n)$. Sortierkriterium ist der für die Elemente definierte Operator `<`.

- `template<class Compare> void sort(Compare cmp)`
  wie `sort()`, aber mit dem Sortierkriterium des Compare-Objekts `cmp`. Das Compare-Objekt ist ein Funktionsobjekt (siehe Seite 344).
- `void unique()`
  löscht gleiche aufeinanderfolgende Elemente bis auf das erste (Zeitkomplexität $O(n)$). Anwendung auf eine sortierte Liste bedeutet, dass danach kein Element mehrfach auftritt.
- `template<class binaeresPraedikat> void unique(binaeresPraedikat P)`
  dito, nur dass statt des Gleichheitskriteriums ein anderes binäres Prädikat genommen wird.
- `void merge(list& L)`
  Verschmelzen zweier sortierter Listen (Zeitkomplexität $O(n1 + n2)$). Die aufgerufene Liste `L` ist anschließend leer, die aufrufende enthält nachher alle Elemente.
- `template<class Compare> void merge(list& L, Compare cmp)`
  wie vorher, aber für den Vergleich von Elementen wird ein Compare-Objekt genommen.
- `void splice(iterator pos, list& x)`
  fügt den Inhalt von Liste `x` vor `pos` ein. `x` ist anschließend leer.
- `void splice(iterator p, list&x, iterator i)`
  Fügt Element `*i` aus `x` vor `p` ein und entfernt `*i` aus `x`.
- `void splice(iterator pos, list& x, iterator first, iterator last)`
  fügt Elemente im Bereich `[first, last)` aus `x` vor `pos` ein und entfernt sie aus `x`. Bei dem Aufruf für dasselbe Objekt (das heißt `&x == this`) wird konstante Zeit benötigt, ansonsten ist der Aufwand von der Ordnung $O(last - first)$. `pos` darf nicht im Bereich `[first, last)` liegen.

Das folgende kleine Programm zeigt, wie zwei Listen mit `merge()` verschmolzen werden.

**Listing 28.2:** Verschmelzen zweier Listen

```
// cppbuch/k28/list/merge.cpp
#include<list>
#include<showSequence.h> // siehe Seite 648
using namespace std;

int main( ) {
    list<int> L1, L2;
    // Listen mit sortierten Zahlen füllen:
    for(int i = 0; i < 5; ++i) {
        L1.push_back(2*i);      // gerade Zahlen
        L2.push_back(2*i+1);    // ungerade Zahlen
    }
    showSequence(L1); // 0 2 4 6 8
    showSequence(L2); // 1 3 5 7 9
    L1.merge(L2);     // verschmelzen
    showSequence(L1); // 0 1 2 3 4 5 6 7 8 9
    showSequence(L2); // (leere Liste)
}
```

### 28.2.4 deque

Der Name Deque ist die Abkürzung für *double ended queue*, also eine Warteschlange, die das Hinzufügen und Entnehmen von Elementen sowohl am Anfang als auch am Ende erlaubt. Die Klasse `deque`, eingebunden durch den Header `<deque>`, stellt öffentliche Typen und Methoden zur Verfügung, die nachfolgend kurz beschrieben werden. Vorab sei auf die schon genannten vorhandenen Datentypen und Methoden hingewiesen, um sie hier nicht im Einzelnen zu wiederholen:

- Datentypen der Tabelle 28.2 (Seite 765)
- Konstruktoren und Methoden der Tabelle 28.3 (Seite 766)
- relationale Operatoren (Seite 767)
- Datentypen/Methoden des Abschnitts 28.1.3 (Reversible Container, Seite 768)
- Konstruktoren und Methoden der Tabelle 28.4 (Sequenzen, Seite 770)

Eine Deque hat wie ein Vektor noch die Datentypen `pointer` und `const_pointer` für Zeiger auf Deque-Elemente (siehe Tabelle 28.5). Die Deklaration der Klasse ist

```
template<typename T> class deque;
```

Die folgende Aufstellung zeigt die Methoden einer Deque, die zusätzlich zu den eben erwähnten benutzbar sind.

- `const_reference front() const` und
  `reference front()`
  liefern eine Referenz auf das erste Element eines Deque-Objekts.
- `const_reference back() const` und
  `reference back()`
  liefern eine Referenz auf das letzte Element eines Deque-Objekts.
- `const_reference operator[](size_type n) const` und
  `reference operator[](size_type n)`
  geben eine Referenz auf das n-te Element zurück.
- `const_reference at(size_type n) const` und
  `reference at(size_type n)`
  geben eine Referenz auf das n-te Element zurück. Der Unterschied zum vorhergehenden `operator[]()` besteht in der Prüfung, ob `n` in einem gültigen Bereich liegt. Falls nicht, d.h. `n` ist $\geq$ `size()`, wird eine `out_of_range`-Exception geworfen.
- `void emplace_front(args)` fügt ein Objekt, dessen Konstruktor mit den Parametern `args` aufgerufen wird, am Anfang ein.
- `void emplace_back(args)` fügt ein Objekt, dessen Konstruktor mit den Parametern `args` aufgerufen wird, am Ende ein.
- `void push_front(const T& t)` fügt `t` am Anfang ein.
  (`void push_front(T&& t)` für R-Werte)
- `void pop_front()` löscht das erste Element.
- `void push_back(const T& t)` fügt `t` am Ende ein.
  (`void push_back(T&& t)` für R-Werte)
- `void pop_back()` löscht das letzte Element.

- `void resize(size_type n, T t = T())`
  Deque-Größe ändern. Dabei werden `n - size()` Elemente `t` am Ende hinzugefügt bzw. `size() - n` Elemente am Ende gelöscht, je nachdem, ob `n` kleiner oder größer als die aktuelle Größe ist.

### 28.2.5 stack

Ein Stack (Header `<stack>`) ist ein Container, der Ablage und Entnahme nur von einer Seite erlaubt. Zuerst abgelegte Objekte werden zuletzt entnommen. Ein Stack benutzt intern einen anderen Container, der die Operationen `back()`, `push_back()` und `pop_back()` unterstützt. Die Stack-Methoden delegieren ihre Aufgabe an die entsprechenden Methoden des Containers, etwa wie auf Seite 299 beschrieben. Es kommen die Container `vector`, `list` und `deque` in Frage. Falls nichts anderes angegeben wird, wird eine Deque (siehe Seite 775) verwendet. Ein Stack stellt zusätzlich den Datentyp `container_type` bereit.

```
// mit deque realisierter Stack (Voreinstellung)
stack<double> Stack1;
// mit vector realisierter Stack
stack<double, vector<double> > Stack2;
```

Die Deklaration der Klasse ist

```
template<typename T, class Container = deque<T> > class stack;
```

Einem Stack stehen die relationalen Operatoren (Seite 767) sowie die folgenden Methoden zur Verfügung. Die Zeitkomplexität ist jeweils $O(1)$, wenn nicht anders angegeben.

- `stack(const Container& cont = Container())`
  Konstruktor. Ein Stack kann mit einem bereits vorhandenen Container initialisiert werden. `Container` ist der Typ des Containers. In diesem Fall ist die Zeitkomplexität $O(cont.size())$.
- `bool empty() const`
  gibt zurück, ob der Stack leer ist.
- `size_type size() const`
  gibt die Anzahl der im Stack befindlichen Elemente zurück.
- `const value_type& top() const` und
  `value_type& top()`
  geben das oberste Element zurück.
- `void push(const value_type& x)`
  legt das Element `x` auf dem Stack ab.
- `void pop()`
  entfernt das oberste Element vom Stack.

Die Typen `size_type` und `value_type` sind dem Container, der intern zur Realisierung des Stacks verwendet wird, entlehnt. Das folgende Programm zeigt eine `stack`-Anwendung.

**Listing 28.3:** Stack

```
// cppbuch/k28/stack.cpp
#include<stack>
#include<iostream>
using namespace std;
```

```
int main() {
    int numbers[] = {1, 5, 6, 0, 9, 1, 8, 7, 2};
    const int COUNT = sizeof(numbers)/sizeof(int);
    stack<int> einStack;
    cout << "Zahlen auf dem Stack ablegen:" << endl;
    for(int i = 0; i < COUNT; ++i) {
        cout.width(6);
        cout << numbers[i];   // protokollieren
        einStack.push(numbers[i]);
    }
    cout << endl;
    cout << "Zahlen vom Stack holen (umgekehrte Reihenfolge!),"
       " anzeigen und löschen:" << endl;
    while(!einStack.empty()) {
        cout.width(6);
        cout << einStack.top(); // obersten Wert anzeigen
        einStack.pop();         // Wert löschen
    }
    cout << endl;
}
```

### 28.2.6 queue

Eine Queue (Header `<queue>`) oder Warteschlange erlaubt die Ablage von Objekten auf einer Seite und ihre Entnahme von der anderen Seite. Die Objekte an den Enden der Queue können ohne Entnahme gelesen werden. Sowohl `list` als auch `deque` sind geeignete Datentypen zur Implementierung. Falls nichts anderes angegeben wird, wird eine Deque (siehe Seite 775) verwendet. Eine Queue stellt zusätzlich den Datentyp `container_type` bereit.

```
// mit deque realisierte Queue (Voreinstellung)
queue<double> Queue1;
// mit list realisierte Queue
queue<double, list<double> > Queue2;
```

Die Deklaration der Klasse ist

```
template<typename T, class Container = deque<T> > class queue;
```

Einer Queue stehen die relationalen Operatoren (Seite 767) sowie die folgenden Methoden zur Verfügung. Die Zeitkomplexität ist jeweils $O(1)$, wenn nicht anders angegeben.

- `queue(const Container& cont = Container())`
  Konstruktor. Eine Queue kann mit einem bereits vorhandenen Container initialisiert werden. `Container` ist der Typ des Containers. In diesem Fall ist die Zeitkomplexität $O(cont.size())$.
- `size_type size() const`
  gibt die Anzahl der in der Queue befindlichen Elemente zurück.
- `const value_type& front() const` und
  `value_type& front()`
  geben das erste Element zurück.

- `const value_type& back() const` und
  `value_type& back()`
  geben das letzte Element zurück.
- `void push(const value_type& x)`
  fügt das Element `x` am Ende der Queue ein.
- `void pop()`
  entfernt das erste Element der Queue.
- `bool empty() const`
  gibt zurück, ob die Queue leer ist.

Die Typen `size_type` und `value_type` sind dem Container, der intern zur Realisierung der Queue verwendet wird, entlehnt. Das folgende Programm zeigt eine `queue`-Anwendung.

**Listing 28.4:** Warteschlange

```
// cppbuch/k28/queue.cpp
#include<queue>
#include<iostream>
using namespace std;

int main() {
    int numbers[] = {1, 5, 6, 0, 9, 1, 8, 7, 2};
    const int COUNT = sizeof(numbers)/sizeof(int);
    queue<int> eineQueue;
    cout << "Zahlen in die Warteschlange schreiben:" << endl;
    for(int i = 0; i < COUNT; ++i) {
        cout.width(6);
        cout << numbers[i];       // protokollieren
        eineQueue.push(numbers[i]);
    }
    cout << endl;
    cout << "Zahlen aus der Warteschlange holen, "
       "anzeigen und löschen:" << endl;
    while(!eineQueue.empty()) {
        cout.width(6);
        cout << eineQueue.front(); // ersten Wert anzeigen
        eineQueue.pop();           // Wert löschen
    }
    cout << endl;
}
```

### 28.2.7 priority_queue

Eine Priority-Queue ist eine prioritätsgesteuerte Warteschlange. Jedem Element ist eine Priorität zugeordnet, die den Platz innerhalb der Priority-Queue schon beim Einfügen bestimmt. Die relative Priorität wird durch Vergleich jeweils zweier Elemente bestimmt, indem entweder der <-Operator oder wahlweise ein Funktionsobjekt zum Vergleich herangezogen wird. Sowohl `vector` als auch `deque` sind geeignete Datentypen zur Implementierung. Falls nichts anderes angegeben ist, wird `deque` (Seite 775) verwendet.

Die genannten Container, als Sequenz eingesetzt, wären nicht gut geeignet, denn das einfache Einfügen eines Elements in die Mitte eines Vektors (das Element habe mittlere

Priorität) hat die Zeitkomplexität $O(n)$, weil ja etwa $n/2$ Elemente nach hinten geschoben werden müssten. Weil aber ein wahlfreier Zugriff mit der Methode `at(int i)` oder dem Index-Operator `[]` möglich ist, wird der implementierende Container intern nicht als Sequenz, sondern als *Heap* aufgebaut. Der Zeitaufwand, ein Element hinzuzufügen, sinkt damit auf $O(\log n)$, eine drastische Verbesserung bei großen Werten von $n$. Einzelheiten zur Heap-Struktur finden Sie auf Seite 688. Die Deklaration der Klasse ist

```
template<class T, class Container = deque<T>,
        class Compare = less<typename Container::value_type> >
class priority_queue;
```

Eine Priority-Queue stellt dieselben Datentypen wie die Queue und die folgenden Methoden zur Verfügung. Die Zeitkomplexität ist $O(1)$, wenn nicht anders angegeben.

- `priority_queue(const Compare& cmp = Compare(),`
  `const Container& = Container())`
  Konstruktor. Eine Priority_Queue kann mit einem bereits vorhandenen Container initialisiert werden. `Container` ist der Typ des Containers. In diesem Fall ist die Zeitkomplexität $O(cont.size())$. `cmp` ist das Funktionsobjekt, mit dem verglichen wird. `less<Container::value_type>` wird angenommen, falls kein Typ angegeben ist.
- `template<class InputIterator>`
  `priority_queue(InputIterator first, InputIterator last,`
  `const Compare& cmp = Compare(),`
  `const Container& cont = Container())`
  Dieser Konstruktor unterscheidet sich von dem vorhergehenden, indem die Elemente im Bereich `[first, last)` eines anderen Containers bei der Konstruktion zusätzlich eingefügt werden. Die Zeitkomplexität ist $O(last - first + cont.size())$.
- `bool empty() const`
  gibt zurück, ob die Priority-Queue leer ist.
- `size_type size() const`
  gibt die Anzahl der in der Priority_Queue befindlichen Elemente zurück.
- `const value_type& top() const`
  gibt das erste Element zurück, d.h. das mit der größten Priorität.
- `void push(const value_type& x)`
  fügt das Element `x` ein. Die Zeitkomplexität ist $O(\log n)$.
- `void pop()`
  entfernt das erste Element. Die Zeitkomplexität ist $O(\log n)$.

Für die `priority_queue`-Klasse gibt es keine relationalen Operatoren. Das folgende Programm zeigt eine `priority_queue`-Anwendung.

**Listing 28.5:** Prioritätsgesteuerte Warteschlange

```
// cppbuch/k28/priorityqueue.cpp
#include<queue>
#include<vector>
#include<iostream>
using namespace std;

int main() {
```

```
    int numbers[] = {1, 5, 6, 0, 9, 1, 8, 7, 2};
    const int COUNT = sizeof(numbers)/sizeof(int);
    // Standard-Implementierung mit deque und less
    priority_queue<int> einePrioQueue;
    cout << "Zahlen in die Prioritäts-Warteschlange schreiben:"
         << endl;
    for(int i = 0; i < COUNT; ++i) {
         cout.width(6);
         cout << numbers[i];        // protokollieren
         einePrioQueue.push(numbers[i]);
    }
    cout << endl;
    cout << "Zahlen aus der Prioritäts-Warteschlange holen,"
          " anzeigen und löschen:" << endl;
    while(!einePrioQueue.empty()) {
         cout.width(6);
         cout << einePrioQueue.top(); // 'wichtigsten' Wert anzeigen
         einePrioQueue.pop();       // Wert löschen
    }
    cout << endl;
}
```

Im Beispiel werden die größeren Zahlen zuerst ausgegeben. Wenn kleinen Zahlen eine hohe Priorität zugeordnet werden soll, ist `greater` statt `less` einzusetzen. Weil dieser vordefinierte Template-Parameter an dritter Stelle steht, ist auch der zweite Parameter, der Container für die interne Implementierung, anzugeben. Beispiel mit `vector`:

```
priority_queue<int, vector<int>, greater<int> > einePrioQueue;
```

### 28.2.8 array

Die Algorithmen der C++-Standardbibliothek, die auf Containern arbeiten, benutzen oft die Methoden `begin()`, `end()` und `size()`. Ein C-Array ist keine Klasse und hat diese Methoden daher nicht. Die Anzahl der Elemente eines C-Arrays kann mit `sizeof` nur unter bestimmten Voraussetzungen ermittelt werden (siehe Seite 191). Das Klassen-Template `array` (Header `<array>`) ist eine Art Hüllklasse (englisch *wrapper*), die ein C-Array kapselt und die entsprechenden Methoden bereitstellt. Im Unterschied zu einem Vektor hat ein `array` eine fixe, zur Compilationszeit festgelegte Größe. Ein `array` ist zwar eine Sequenz, die Methoden unterscheiden sich aber von denen der Tabelle 28.4 (Seite 770). Die Deklaration eines `array` ist

```
template<typename T, size_t N> class array;
```

Dabei gibt `T` den Typ der abzuspeichernden Elemente an. `N` ist die Anzahl der Elemente. Es gibt nur den (vorgegebenen) Standardkonstruktor. Ein `array` hat strukturelle Ähnlichkeit mit dem Beispiel eines Stacks festgelegter Größe aus Abschnitt 6.3.2 (Seite 249). Ein `array` stellt die folgenden öffentlichen Typen und Methoden zur Verfügung:

- Datentypen der Tabelle 28.2 (Seite 765)
- Konstruktoren und Methoden der Tabelle 28.3 (Seite 766)
- relationale Operatoren (Seite 767)

- Datentypen/Methoden des Abschnitts 28.1.3 (Reversible Container, Seite 768)

Darüber hinaus gibt es die folgenden Methoden. Die Zeitkomplexität ist $O(1)$, wenn nicht anders angegeben.

- `void assign(const T& t)`
  weist allen Elementen des Arrays eine Kopie von `t` zu. Die Zeitkomplexität ist $O(N)$.
- `const_reference front() const` und
  `reference front()`
  liefern eine Referenz auf das erste Element.
- `const_reference back() const` und
  `reference back()`
  liefern eine Referenz auf das letzte Element.
- `const_pointer data() const` und
  `pointer data()`
  liefern einen Zeiger auf das erste Element. Es gilt also `data() == &front()`, falls das Array nicht-leer ist.
- `const_reference operator[](size_type n) const` und
  `reference operator[](size_type n)`
  geben eine Referenz auf das n-te Element zurück.
- `const_reference at(size_type n) const` und
  `reference at(size_type n)`
  geben eine Referenz auf das n-te Element zurück. Der Unterschied zum vorhergehenden `operator[]()` besteht in der Prüfung, ob `n` in einem gültigen Bereich liegt. Falls nicht, d.h. `n` ist $\geq$ `size()`, wird eine `out_of_range`-Exception geworfen.

Falls das Array leer ist (d.h. `N == 0`), ist das Ergebnis der Funktionen `data()`, `front()` und `back()` undefiniert. `size()` liefert natürlich stets den unveränderlichen Wert `N`. Ein Array dieser Art kann wie ein Tupel (siehe Abschnitt27.4 auf Seite 752) mit `N` Elementen aufgefasst werden. Um kompatibel zu `tuple` zu sein, besitzt `array` die entsprechenden `get()`-Methoden, für deren Aufruf die Voraussetzung ist `0` $\leq$ `I` < `N` gelten muss.

```
template <int I, class T, size_t N>
T& get(array<T, N>& a);

template <int I, class T, size_t N>
const T& get(const array<T, N>& a);
```

Das Beispiel unten zeigt einige Verwendungsmöglichkeiten eines `array`-Objekts. Wie zu sehen ist, kann es mit einer durch geschweifte Klammern begrenzten Liste initialisiert werden (vgl. Abschnitt 5.2.3 auf Seite 192).

**Listing 28.6:** Klasse Array

```
// cppbuch/k28/array.cpp
#include<array>
#include<iostream>
using namespace std;

int main() {
  const size_t ANZAHL = 3;
  array<int, ANZAHL> tabelle = {{ 9, -10, 5 }}; // Initialisierung:
```

```
  for(auto i = tabelle.begin();
      i != tabelle.end(); ++i) {              // Benutzung mit Iterator
     cout << *i << endl;
  }
  for(size_t i = 0; i < tabelle.size(); ++i) {
     cout << tabelle[i] << endl;              // Benutzung wie ein Vektor
  }
  // Benutzung wie ein Tupel
  cout << get<0>(tabelle) << endl;
  cout << get<1>(tabelle) << endl;
  cout << get<2>(tabelle) << endl;
}
```

# 28.3 Sortierte assoziative Container

Wegen der intern verwendeten Baumstruktur für die Daten ist die Zugriffszeit zu einem Element proportional zu log *n*, wenn *n* die Anzahl der gespeicherten Elemente ist. `map` ist die programmtechnische Umsetzung einer Relation oder diskreten Abbildung im mathematischen Sinn. Das mathematische Konzept einer Menge wird mit `set` realisiert. `multimap` und `multiset` erlauben mehrfache identische Schlüssel.

## 28.3.1 map

Die Klasse `map<Key, T>`, eingebunden durch den Header `<map>`, speichert *Paare* von Schlüsseln und zugehörigen Daten. Dabei ist der Schlüssel eindeutig, es kann also keine zwei Datensätze zu demselben Schlüssel geben. `map` ist ein assoziativer Container: Die Daten werden durch direkte Angabe des Schlüssels gefunden.

Damit ist ein `map`-Objekt eine Abbildung der enthaltenen Schlüssel auf die zugehörigen Daten. Obwohl die Schlüssel für einen assoziativen Zugriff nicht sortiert sein müssen, liegen sie in der Klasse `map` sortiert vor. Das ergibt den Vorteil der sortierten Ausgabe zum Beispiel beim Durchwandern vom ersten bis zum letzten Element. Dem steht der Nachteil einer längeren Zugriffszeit im Vergleich zu Datenstrukturen gegenüber, die auf der gestreuten Speicherung basieren (sogenannte Hash-Maps, siehe Abschnitt 28.4.1 weiter unten). Der Typ eines `map`-Elements ist `pair<const Key, T>`. Die Konstanz des Schlüssels ist notwendig, damit ein Schlüssel, der ja dank der Sortierung einer Position im Container entspricht, nicht geändert werden kann – dies würde die Sortierung zerstören. Die Deklaration der Klasse ist

```
template<class Key,                  // Typ der Schlüssel
         class T,                    // Typ der Daten
         class Compare = less<Key> > // Standardvergleich für Sortierung
class map;
```

Mit der `map`-Klasse lässt sich leicht ein Wörterbuch realisieren:

```
map<string, string> woerterbuch;
```

```
// Eintrag aufnehmen
woerterbuch.insert(pair<string,string>("Buch","book"));
// hier lässt sich auch make_pair von Seite 751 einsetzen:
woerterbuch.insert(make_pair("Ziffer","digit"));
// ... weitere Einträge aufnehmen

// assoziativer Zugriff auf einen Eintrag:
cout << woerterbuch["Buch"] << endl;     // book

// sortierte Ausgabe des ganzen Wörterbuchs
auto i = woerterbuch.begin();
while(i != woerterbuch.end()) {
  cout << (*i).first << " " << (*i).second << endl;
  ++i;
}
```

Falls die Sortierung nicht mit dem <-Operator (das heißt `less<Key>`) hergestellt werden soll, kann eine Klasse für Funktionsobjekte angegeben werden:

```
class Vergleich { ....}; // Klasse für Funktionsobjekt

// Funktionsobjekt Vergleich():
map<string, string, Vergleich> woerterbuch;

Vergleich einVergleich;
// Funktionsobjekt einVergleich:
map<string, string, Vergleich> woerterbuch(einVergleich);
```

Ein `map`-Container stellt unter anderem die folgenden öffentlichen Datentypen und Methoden bereit:

- Datentypen der Tabelle 28.2 (Seite 765)
- Konstruktoren und Methoden der Tabelle 28.3 (Seite 766)
- relationale Operatoren (Seite 767)
- Datentypen/Methoden des Abschnitts 28.1.3 (Reversible Container, Seite 768)

Es gibt weitere öffentliche Datentypen, die in der Tabelle 28.6 aufgelistet sind.

**Tabelle 28.6:** Zusätzliche Datentypen für `map<Key, T, Compare>`

| Datentyp | Bedeutung |
|---|---|
| `pointer` | Zeiger auf Map-Element |
| `const_pointer` | nur lesend verwendbarer Zeiger |
| `key_type` | entspricht `Key` |
| `value_type` | entspricht `pair<const Key, T>` |
| `mapped_type` | entspricht `T` |
| `key_compare` | `Compare` |
| `value_compare` | Klasse für Funktionsobjekte, siehe Funktion `value_comp()` auf Seite 785 |

Außer den oben angegebenen Methoden gibt es noch die folgenden (die Zeitkomplexität ist $O(1)$, wenn nicht anders angegeben):

- `map(const Compare& cmp = Compare())`
  Konstruktor, der ein Compare-Objekt akzeptieren kann. Falls keines angegeben ist, wird `less<Key>()` genommen.
- `template<class InputIterator>`
  `map(InputIterator first, InputIterator last,`
  `  const Compare& cmp = Compare())`
  Eine Map kann mit Daten initialisiert werden, wenn dem Konstruktor ein Bereich von Schlüssel/Wert-Paaren in Form von Iteratoren übergeben wird. Die Zeitkomplexität ist $O(n\log n)$ mit $n = last - first$.
- `T& operator[](const key_type& k)`
  gibt eine Referenz auf die zum Schlüssel `k` gehörenden Daten zurück. Falls der Schlüssel nicht existiert, wird zu diesem Schlüssel ein Objekt mit dem Konstruktor `T()` für den Wertteil angelegt. Die Zeitkomplexität ist $O(\log n)$.
- `const T& at(const key_type& k) const` und
  `T& at(const key_type& k)`
  geben eine Referenz auf die zum Schlüssel `k` gehörenden Daten zurück. Falls der Schlüssel nicht existiert, wird eine `out_of_range`-Exception geworfen. Die Zeitkomplexität ist $O(\log n)$.
- `pair<iterator, bool> emplace(args)`
  fügt ein Objekt, das mit den Parametern `args` konstruiert wird, ein (zu `emplace` siehe Seite 768). Zum Rückgabewert siehe `insert()`.
- `pair<iterator, bool> insert(const value_type& x)`
  fügt das Schlüssel/Daten-Paar `x` ein, sofern noch kein Element mit dem entsprechenden Schlüssel vorhanden ist. Der Iterator des zurückgegebenen Paares zeigt auf das eingefügte Element bzw. auf das Element mit demselben Schlüssel wie `x`. Der Wahrheitswert zeigt, ob überhaupt ein Einfügen stattgefunden hat – es könnte ja sein, dass der Schlüssel bereits existiert. Die Zeitkomplexität ist $O(\log n)$.
- `void insert(InputIterator i, InputIteratorj)`
  fügt die Elemente aus dem Iteratorbereich `[i, j)` ein. Die Zeitkomplexität ist $O(n\log n)$ mit $n = j - i$.
- `iterator insert(iterator p, const value_type& x)`
  wie `insert(const value_type& x)`, wobei der Iterator `p` ein Hinweis sein soll, wo die Suche zum Einfügen beginnen soll. Der zurückgegebene Iterator zeigt auf das eingefügte Element bzw. auf das mit demselben Schlüssel wie `x`. Die Zeitkomplexität ist praktisch $O(1)$, wenn mit `p` die richtige Stelle getroffen wird, ansonsten $O(\log n)$.
- `void insert(il)` fügt die Elemente aus der Initialisierungsliste `il` ein.
- `size_type erase(const key_type& k)`
  alle Elemente mit einem Schlüssel gleich `k` löschen. Es wird die Anzahl $N$ der gelöschten Elemente (hier: 0 oder 1, bei einer `multimap` auch mehr) zurückgegeben. Die Zeitkomplexität ist $O(\log n)$, für eine `multimap`: $O(\log n + N)$.
- `iterator erase(iterator p)`
  das Element löschen, auf das der Iterator `p` zeigt. Der zurückgegebene Iterator zeigt anschließend auf das `p` folgende Element, wenn es existiert, ansonsten wird ein End-Iterator (`end()`) zurückgegeben. Die Zeitkomplexität ist $O(\log n)$.

- `iterator erase(iterator p, iterator q)`
  alle Elemente im Iteratorbereich `[p, q)` löschen. Der zurückgegebene Iterator zeigt anschließend auf das `q` folgende Element, wenn es existiert, ansonsten wird ein End-Iterator (`end()`) zurückgegeben. Die Zeitkomplexität ist $O((q - p)\log n)$.
- `void clear()`
  löscht alle Elemente. Ein Aufruf entspricht `erase(begin(), end())`. Die Zeitkomplexität ist $O(n\log n)$.
- `key_compare key_comp() const`
  gibt eine Kopie des Vergleichsobjekts zurück, das zur Konstruktion der Map benutzt wurde.
- `value_compare value_comp() const`
  gibt ein Funktionsobjekt zurück, das zum Vergleich von Objekten des Typs `value_type` (also Paaren) benutzt werden kann. Dieses Funktionsobjekt vergleicht zwei Paare auf der Basis ihrer Schlüssel und des Vergleichsobjekts, das zur Konstruktion der `map` benutzt wurde.
- `const_iterator find(const key_type& k) const` und
  `iterator find(const key_type& k)`
  geben einen Iterator auf ein Element mit dem Schlüssel `k` zurück, falls vorhanden. Andernfalls wird `end()` zurückgegeben. Die Zeitkomplexität ist $O(\log n)$.
- `size_type count(const key_type& k) const`
  gibt die Anzahl $N$ der Elemente (hier: 0 oder 1, bei einer `multimap` auch mehr) mit dem Schlüssel `k` zurück. Die Zeitkomplexität ist $O(\log n)$, für eine `multimap`: $O(\log n + N)$.
- `const_iterator lower_bound(const key_type& k) const` und
  `iterator lower_bound(const key_type& k)`
  zeigen auf das erste Element, dessen Schlüssel nicht kleiner als `k` ist. Die Zeitkomplexität ist $O(\log n)$.
- `const_iterator upper_bound(const key_type& k) const` und
  `iterator upper_bound(const key_type& k)`
  geben einen Iterator auf das erste Element zurück, dessen Schlüssel größer als `k` ist. Die Zeitkomplexität ist $O(\log n)$.
- `pair<const_iterator, const_iterator>`
  `equal_range(const key_type& k) const` und
  `pair<iterator, iterator> equal_range(const key_type& k)`
  geben ein Paar von Iteratoren zurück, zwischen denen die Schlüssel gleich `k` sind. Die Differenz der Iteratoren kann bei einer `map` nur 0 oder 1 sein. Die Zeitkomplexität ist $O(\log n)$. Bei einer `multimap` (siehe unten) können die Differenz und die Komplexität größer sein.

Das Programm unten zeigt eine Anwendung, in der zu jedem Namen eine Telefonnummer abgespeichert wird. Das Einfügen geschieht unsortiert, die Ausgabe ist wegen der internen Implementierung als Suchbaum automatisch sortiert.

Die Suche mit `operator[]` ist trotz der einfacheren Schreibweise unvorteilhaft, weil bei Eingabe eines nicht-existierenden Namens automatisch ein Eintrag dieses Namens mit der Nummer 0 angelegt wird.

```
cout << aMap[derName] << endl; // nicht empfehlenswert!
```

Aus diesem Grund gibt es die Funktion `at()`. In Analogie zur gleichnamigen Funktion bei anderen Containern wirft `at()` eine Exception, wenn der Eintrag nicht vorhanden ist:

```
try {
   cout << aMap.at(derName) << endl;     // O(logN)
} catch(const exception& e) {
   cout << "Nicht gefunden! Exception:" << e.what() << endl;
}
```

**Listing 28.7:** Sortierte Abbildung (map)

```
// cppbuch/k28/sortedmap.cpp
#include<map>
#include<string>
#include<iostream>
using namespace std;

// Zwei typedefs zur Abkürzung
typedef map<string, long> MapType; // Vergleichsobjekt ist less<string>()
typedef MapType::value_type ValuePair;

int main() {
   MapType aMap;
   aMap.insert(ValuePair("Thomas", 5192835));
   aMap.insert(ValuePair("Werner", 24439404));
   aMap.insert(ValuePair("Manfred", 535353));
   aMap.insert(ValuePair("Heiko", 635352723));
   aMap.insert(ValuePair("Andreas", 42536347));
   aMap.insert(ValuePair("Karin", 9273539));
   // Das zweite Einfügen von Heiko mit einer anderen Nummer wird
   // nicht ausgeführt, weil der Schlüssel schon existiert.
   aMap.insert(ValuePair("Heiko", 1000000));
   cout << "Ausgabe:\n"; // sortiert
   auto iter = aMap.begin();
   while(iter != aMap.end()) {
      cout << (*iter).first << ':'   // Name
           << (*iter).second << endl; // Nummer
      ++iter;
   }
   cout << "Ausgabe der Nummer nach Eingabe des Namens\n"
        << "Name: ";
   string derName;
   cin >> derName;
   cout << "Suche mit Iterator: ";
   iter = aMap.find(derName);        // O(log N)
   if(iter != aMap.end()) {
      cout << (*iter).second << endl;
   }
   else {
      cout << "Nicht gefunden!" << endl;
   }
   try {
      cout << "Suche mit at(): " << aMap.at(derName) << endl;
   } catch(const exception& e) {
```

```
        cout << "Nicht gefunden! Exception: " << e.what() << endl;
    }
}
```

### 28.3.2 multimap

Die Klasse `multimap` unterscheidet sich von der Klasse `map` dadurch, dass *mehrfache* Einträge von Elementen mit identischen Schlüsseln möglich sind. Die Datentypen und Methoden entsprechen denen der Klasse `map` mit den folgenden wenigen Unterschieden:

- Es gibt keinen Index-Operator `T& operator[](const key_type& x)`.
- Die `insert()`-Methoden fügen das Schlüssel/Daten-Paar ein, auch wenn ein Element mit dem entsprechenden Schlüssel schon vorhanden ist.
- Die Methode `emplace()` hat einen anderen Rückgabetyp, die Signatur ist `iterator emplace(args)`. Der Iterator zeigt auf das eingefügte Objekt,

### 28.3.3 set

Die Klasse `set<Key, Compare>` für Mengen, eingebunden durch den Header `<set>`, entspricht der Klasse `map` von Seite 782, nur dass Schlüssel und Daten zusammenfallen. Das heißt, es werden nur Schlüssel gespeichert. Dabei ist der Schlüssel eindeutig, es kann also keine zwei Datensätze zu demselben Schlüssel geben. `set` ist ein assoziativer Container: Die direkte Angabe des Schlüssels sagt, ob er vorhanden ist oder nicht. Obwohl für den assoziativen Zugriff die Schlüssel nicht sortiert sein müssen, liegen sie in der Klasse `set` sortiert vor. Das ergibt den Vorteil der sortierten Ausgabe zum Beispiel beim Durchwandern vom ersten bis zum letzten Element. Dem steht der Nachteil einer längeren Zugriffszeit im Vergleich zu Datenstrukturen gegenüber, die auf der gestreuten Speicherung basieren (sogenannte Hash-Sets, siehe Abschnitt 28.4.3 weiter unten). Die Deklaration der Klasse ist

```
template<class Key,                    // Schlüssel
        class Compare = less<Key> > // Standardvergleich für Sortierung
class set;
```

Falls die Sortierung nicht mit dem <-Operator (das heißt `less<Key>`) hergestellt werden soll, kann eine Klasse für Funktionsobjekte angegeben werden:

```
class Vergleich { ....}; // Klasse für Funktionsobjekt
// Funktionsobjekt Vergleich():
set<string, Vergleich> Woertermenge;
Vergleich einVergleich; // Funktionsobjekt
set<string, Vergleich> Woertermenge(einVergleich);
```

Ein `set`-Container stellt unter anderem die folgenden öffentlichen Datentypen und Methoden bereit:

- Datentypen der Tabelle 28.2 (Seite 765)
- Konstruktoren und Methoden der Tabelle 28.3 (Seite 766)
- relationale Operatoren (Seite 767)
- Datentypen/Methoden des Abschnitts 28.1.3 (Reversible Container, Seite 768)

Es gibt weitere öffentliche Datentypen, die in der Tabelle 28.7 aufgelistet sind.

**Tabelle 28.7:** Zusätzliche Datentypen für `set<Key, Compare>`

| Datentyp | Bedeutung |
|---|---|
| `pointer` | Zeiger auf Set-Element |
| `const_pointer` | dito, aber nur lesend verwendbar |
| `key_type` | entspricht `Key` |
| `value_type` | entspricht `Key` (im Gegensatz zur `map`) |
| `key_compare` | `Compare` |
| `value_compare` | `Compare` (im Gegensatz zur `map`) |

Zusätzlich zu den oben angegebenen Methoden gibt es die folgenden (die Zeitkomplexität ist $O(1)$, wenn nicht anders angegeben):

- `set(const Compare& cmp = Compare())`
  Konstruktor, der ein Compare-Objekt akzeptieren kann. Falls kein Typ angegeben ist, wird `less<Key>` genommen.
- `template<class InputIterator>`
  `set(InputIterator first, InputIterator last, const Compare& cmp = Compare())`
  Ein Set kann mit Daten initialisiert werden, wenn dem Konstruktor ein Bereich in Form von Iteratoren übergeben wird. Die Zeitkomplexität ist $O(n\log n)$ mit $n = last - first$.
- `pair<iterator, bool> emplace(args)`
  fügt ein Objekt ein, das mit den Parametern `args` konstruiert wird (siehe Seite 768). Zum Rückgabewert siehe `insert()`.
- `pair<iterator, bool> insert(const value_type& k)`
  fügt den Schlüssel `k` ein, sofern er noch nicht vorhanden ist. Der Iterator des zurückgegebenen Paares zeigt auf den eingefügten Schlüssel bzw. auf den schon vorhandenen Schlüssel mit demselben Wert wie `k`. Der Wahrheitswert zeigt, ob überhaupt ein Einfügen stattgefunden hat – es könnte ja sein, dass der Schlüssel bereits existiert. Die Zeitkomplexität ist $O(\log n)$.
- `iterator insert(iterator p, const value_type& k)`
  wie `insert(const key_type& k)`, wobei der Iterator `p` ein Hinweis ist, wo die Suche zum Einfügen beginnen soll. Falls `p` die richtige Stelle trifft, ist die Zeitkomplexität $O(1)$, ansonsten $O(\log n)$. Der zurückgegebene Iterator zeigt auf das eingefügte Element bzw. auf das schon vorhandene.
- `void insert(InputIterator i, InputIterator j)`
  fügt die Elemente aus dem Iteratorbereich `[i, j)` ein. Mit $N = j - i$ ist die Zeitkomplexität $O(N\log N)$.
- `size_type erase(const key_type& k)`
  alle Schlüssel gleich `k` löschen. Es wird die Anzahl $N$ der gelöschten Elemente (hier: 0 oder 1) zurückgegeben. Die Zeitkomplexität ist $O(\log n)$ bzw. $O(\log n + N)$ bei einer `multimap`.
- `iterator erase(iterator p)`
  löscht das Element, auf das der Iterator `p` zeigt. Der zurückgegebene Iterator zeigt anschließend auf das `p` folgende Element, wenn es existiert, ansonsten wird ein End-Iterator (`end()`) zurückgegeben. Die Zeitkomplexität ist $O(\log n)$.

- `void erase(iterator p, iterator q)`
  alle Elemente im Iteratorbereich `[p, q)` löschen. Der zurückgegebene Iterator zeigt anschließend auf das `q` folgende Element, wenn es existiert, ansonsten wird ein End-Iterator (`end()`) zurückgegeben. Die Zeitkomplexität ist $O((q - p)\log n)$.
- `void clear()`
  löscht alle Elemente. Der Aufruf entspricht `erase(begin(), end())`. Die Zeitkomplexität ist $O(n\log n)$.
- `key_compare key_comp() const`
  gibt eine Kopie des Vergleichsobjekts zurück, das zur Konstruktion des Sets benutzt wurde.
- `value_compare value_comp() const`
  dasselbe wie `key_comp()`. Diese Funktion ist nur der Vollständigkeit halber aufgeführt, weil sie auch in der Klasse `map` vorkommt, dort mit anderer Bedeutung.
- `const_iterator find(const key_type& k) const` und
  `iterator find(const key_type& k)`
  geben einen Iterator auf den Schlüssel `k` zurück, falls vorhanden. Andernfalls wird `end()` zurückgegeben. Die Zeitkomplexität ist $O(\log n)$.
- `size_type count(const key_type& k) const`
  gibt die Anzahl $N$ der Elemente (hier: 0 oder 1, bei einem `multiset` auch mehr) mit dem Schlüssel `k` zurück. Die Zeitkomplexität ist $O(\log n)$, für einen `multiset`: $O(\log n + N)$.
- `const_iterator lower_bound(const key_type& k) const` und
  `iterator lower_bound(const key_type& k)`
  zeigen auf den ersten Schlüssel, der nicht kleiner als `k` ist. Die Zeitkomplexität ist $O(\log n)$.
- `const_iterator upper_bound(const key_type& k) const` und
  `iterator upper_bound(const key_type& k)`
  geben einen Iterator auf das erste Element zurück, dessen Wert größer als `k` ist. Die Zeitkomplexität ist $O(\log n)$.
- `pair<const_iterator, const_iterator>`
  `equal_range(const key_type& k) const` und
  `pair<iterator, iterator> equal_range(const key_type& k)`
  geben ein Paar von Iteratoren zurück, zwischen denen die Schlüssel gleich `k` sind. Die Differenz der Iteratoren kann hier nur 0 oder 1 sein. Die Zeitkomplexität ist $O(\log n)$. Bei einem `multiset` (siehe unten) können die Differenz und die Komplexität größer sein.

Das folgende Programm zeigt die Verwendung eines Sets. Die Ausgabe zeigt, dass die Elemente trotz zweifachen `insert()`-Aufrufs nur einmal vorkommen. Anschließend wird ein Element gelöscht, einmal über einen Iterator und einmal über seinen Wert.

**Listing 28.8:** Set

```
// cppbuch/k28/setm.cpp
#include<set>
#include"<showSequence.h>
using namespace std;

int main() {
   set<int> einSet; // vordefinierter Vergleich: less<int>()
   for(int i = 0; i < 10; ++i) einSet.insert(i);
   for(int i = 0; i < 10; ++i) einSet.insert(i); // keine Wirkung
   showSequence(einSet);              // 0 1 2 3 4 5 6 7 8 9

   cout << "Löschen mit Iterator\n Welches Element? (0..9)" ;
   int i;
   cin >> i;
   auto iter = einSet.find(i);
   if(iter == einSet.end()) {
      cout << i << " nicht gefunden!\n";
   }
   else {
       // count() kann nur 0 oder 1 ergeben.
       cout << "Das Element " << i          // 1
            << " existiert " << einSet.count(i) << " mal."
            << endl;
       einSet.erase(iter);
       cout << i << " gelöscht!\n";
       cout << "Das Element " << i          // 0
            << " existiert " << einSet.count(i) << " mal."
            << endl;
   }
   showSequence(einSet);

   cout << "Löschen durch Suche nach dem Wert\n"
          "Welches Element? (0..9)" ;
   cin >> i;
   int anzahl = einSet.erase(i);
   if(anzahl == 0) {
      cout << i << " nicht gefunden!\n";
   }
   showSequence(einSet);
}
```

Ein `set` kann nicht nur mit `insert()` initialisiert werden, sondern auch durch Angabe eines Iterator-Paars. Alle Elemente des Bereichs werden dabei eingefügt. Das funktioniert auch für ein C-Array. Dubletten werden eliminiert:

```
int tabelle[] = { 1, 2, 2, 4, 9, 13, 1, 0, 5}; // 2 und 1 doppelt
size_t count = sizeof(tabelle)/sizeof(tabelle[0]);
set<int> nochEinSet(tabelle, tabelle + count);
showSequence(nochEinSet); // 0 1 2 4 5 9 13
```

### 28.3.4 multiset

Die Klasse `multiset` unterscheidet sich von der Klasse `set` dadurch, dass *mehrfache* Einträge identischer Schlüssel möglich sind. Die Datentypen und Methoden entsprechen denen der Klasse `set` mit folgenden Ausnahmen:

- `iterator insert(const key_type& k)`
  fügt den Schlüssel `k` ein, auch wenn er schon vorhanden ist. Der zurückgegebene Iterator zeigt auf das eingefügte Element.
- Die Methode `emplace()` hat einen anderen Rückgabetyp, die Signatur ist `iterator emplace(args)`. Der Iterator zeigt auf das eingefügte Objekt.

## 28.4 Hash-Container

Die Sortierung der assoziativen Container wird manchmal nicht benötigt. Die Reihenfolge der Elemente einer Menge oder Abbildung muss durchaus nicht definiert sein. Der Verzicht auf die Sortierung erlaubt es, aus dem Schlüssel die Adresse eines gesuchten Elements direkt zu berechnen. Zum Beispiel baut ein Compiler eine Symboltabelle auf, auf deren Elemente sehr schnell zugegriffen werden soll. Die Zeitkomplexität des Zugriffs ist $O(1)$, unabhängig von der Anzahl $N$ der Elemente in der Tabelle. Voraussetzung ist, dass die Adresse in konstanter Zeit durch eine einfache Formel berechnet werden kann, ausreichend Speicher zur Verfügung steht und dass die Adressberechnung eine gleichmäßige Verteilung der Elemente im Speicher liefert.

Diese Art der Ablage wird Streuspeicherung genannt. Sie ist immer dann geeignet, wenn die tatsächliche Anzahl zu speichernder Schlüssel klein ist, verglichen mit der Anzahl der möglichen Schlüssel. Ein Compiler kann eine Symboltabelle mit 10000 Einträgen vorsehen; die Anzahl der möglichen Variablennamen mit zum Beispiel nur 10 Zeichen ist sehr viel größer. Wenn wir der Einfachheit halber annehmen, dass nur die 26 Kleinbuchstaben verwendet werden sollen, ergeben sich bereits $26^{10}$ = ca. $1{,}4 \cdot 10^{14}$ Möglichkeiten. Dasselbe Problem stellt sich bei der Speicherung riesiger Matrizen, deren Elemente nur zu einem kleinen Prozentsatz ungleich Null sind.

Die Funktion $h(k)$ zur Transformation des Schlüssels $k$ in die Adresse heißt *Hash-Funktion* (vom englischen *to hash* = hacken, haschieren, durcheinanderbringen usw.), weil alle $N$ Möglichkeiten der Schlüssel auf $M$ Speicherplätze abgebildet werden müssen, indem Informationen zerhackt und verwürfelt werden. Dabei sei $M$ sehr viel kleiner als $N$, woraus sofort ein Problem resultiert: Zwei verschiedene Schlüssel ergeben möglicherweise dieselbe Adresse. Solchen Kollisionen muss Rechnung getragen werden. Die Funktion $h(k); 0 \leq k < N$ darf nur Werte zwischen 0 und $M - 1$ annehmen. Eine sehr einfache Hash-Funktion für Zahlenschlüssel ist die Modulo-Funktion

$$h(k) = k \bmod M$$

Dabei wird für die Tabellengröße $M$ eine Primzahl gewählt, um eine gleichmäßige Verteilung zu erreichen. Dennoch hängt die Verteilung sehr von der Art und dem Vorkommen der Schlüssel ab, und es ist manchmal schwierig, eine Funktion zu finden, die zu nur we-

nigen Kollisionen führt. Eine Hash-Funktion für Zeichenketten sollte dafür sorgen, dass ähnliche Zeichenketten nicht zu Ballungen in der Streutabelle führen. Am besten ist es, wenn die Belegung anhand von »realen« Daten kontrolliert wird, um die Hash-Funktion vor dem produktiven Einsatz einer Software geeignet anpassen zu können.

### Kollisionsbehandlung

Was tun, wenn zwei Schlüssel auf derselben Adresse landen? Der zweite hat das Nachsehen, wenn der Platz schon besetzt ist. Zur Lösung dieses Problems legt ein übliches Verfahren die Schlüssel nicht direkt ab. Vielmehr besteht jeder Eintrag in der Tabelle aus einem Verweis auf eine verkettete Liste oder eine andere geeignete Datenstruktur, in der alle Schlüssel mit demselben Hash-Funktionswert abgelegt werden. So ein Tabelleneintrag wird *Bucket* (deutsch Eimer) genannt. Das Verfahren heißt »Streuspeicherung mit Kollisionsauflösung durch Verketten« und ist in Abbildung 28.1 dargestellt. Sie zeigt die mögliche innere Struktur eines `unordered_map`-Containers.

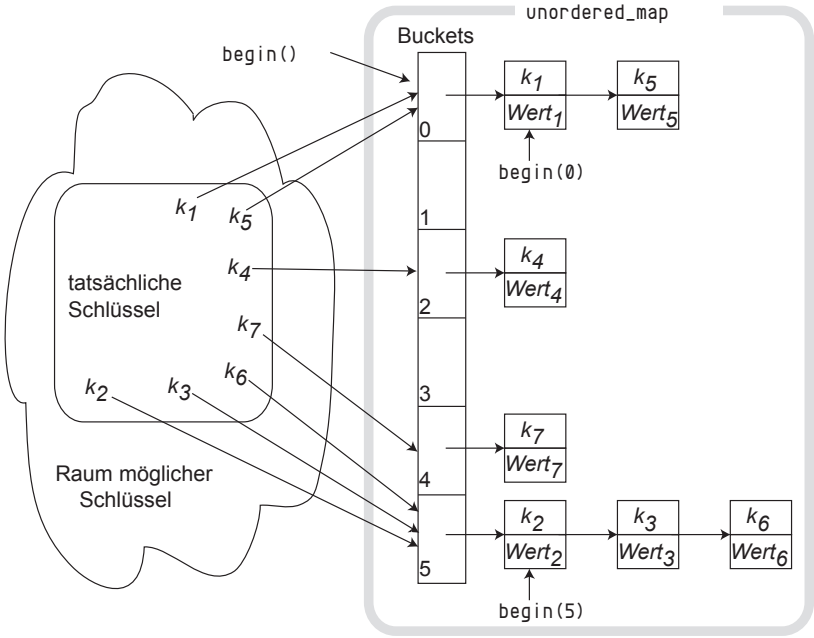


**Abbildung 28.1:** Streuspeicherung mit Kollisionsauflösung durch Verketten

Ein Tabellenelement *bucket*[*i*] verweist auf eine Liste aller Schlüssel, deren Hash-Funktionswert = *i* ist. In Abbildung 28.1 gelten $h(k_1) = h(k_5) = 0$; $h(k_2) = h(k_3) = h(k_6) = 5$; $h(k_4) = 2$ und $h(k_7) = 4$. Der Belegungsgrad (englisch *load factor*) ist das Verhältnis der Anzahl der gespeicherten Elemente zur Anzahl der Buckets. Er sollte erfahrungsgemäß kleiner als 0,7 sein. Das Löschen eines Elements ist einfach, und es können dank der fast beliebigen Länge einer Liste sogar mehr Elemente abgelegt werden, als die Tabelle Positionen hat. Das damit verbundene Anwachsen des Belegungsgrades auf einen Wert

> 1 ist natürlich mit einer Leistungseinbuße verbunden, weil die Such- oder Einfügedauer im schlimmsten Fall proportional zur Länge der längsten Liste ist. Im Fall eines `unordered_set`-Containers sind Schlüssel und Daten dasselbe, sodass die Listenelemente nur Schlüssel enthalten.

Wie üblich, können auch diese Container mithilfe der von `begin()` bzw. `end()` gelieferten Iteratoren durchwandert werden. Um alle Elemente, die auf denselben Hash-Funktionswert abgebildet werden, erreichen zu können, gibt es die beiden Funktionen `begin(size_t n)` bzw. `end(size_t n)`, die einen lokalen Iterator auf die zum Bucket mit der Nummer `n` gehörende Liste zurückgeben (siehe `begin(0)` und `begin(5)` in Abbildung 28.1).

### 28.4.1 unordered_map

Die Klasse `unordered_map`, eingebunden durch den Header `<unordered_map>`, speichert wie `map` von Seite 782 Paare von Schlüsseln und zugehörigen Daten. Im Gegensatz zur `map` ist die Speicherung nicht sortiert. Die Deklaration der Klasse ist

```
template<class Key,                 // Typ der Schlüssel
         class T,                   // Typ der Daten
         class Hash = hash<Key>,    // Hash-Funktion
         class Pred = std::equal_to<Key> > // für Elementvergleich,
                                           // siehe Seite 753
class unordered_map;
```

`hash<Key>` ist der Typ für ein Funktionsobjekt, das eine Zahl des Typs `size_t` als Hash-Wert zurückgibt. Für alle Grunddatentypen wie `int`, `double`, `char` usw. sowie für Zeiger und `string` liegen Spezialisierungen vor, für andere Typen muss man die Hash-Funktion nach dem folgenden Muster selbst schreiben:

```
// eigene Hash-Funktionsklasse für den Typ meinKey
struct meinHash {
   std::size_t operator()(meinKey wert) const {
     // hier Hash-Wert aus wert berechnen und zurückgeben
   }
}
```

Dieser Typ kann dann bei der Anlage einer `unordered_map` übergeben werden:

```
unordered_map<meinKey, TypDerDaten, meinHash> eineAbbildung;
```

Wenn der Schlüssel selbst mehrere Attribute hat, für die es die vordefinierten Funktionen gibt, kann man sie natürlich nutzen. Zum Beispiel sollen Objekte einer Klasse `Name`, die die Attribute `vorname` und `nachname` hat, als Schlüssel dienen. Der Hash-Wert könnte dann wie folgt für ein Objekt `einName` dieser Klasse berechnet werden:

```
hash<string> hashFunktion;  // hash<string> ist vordefiniert.
return hashFunktion(einName.getVorname())
     + hashFunktion(einName.getNachname());
```

Ein `unordered_map`-Container stellt unter anderem die folgenden öffentlichen Datentypen und Methoden bereit:

- Datentypen der Tabelle 28.2 (Seite 765)
- Methoden der Tabelle 28.3 (Seite 766)

Relationale Operatoren sind nicht definiert. Es gibt weitere öffentliche Datentypen, die in der Tabelle 28.8 aufgelistet sind.

**Tabelle 28.8:** Zusätzliche Datentypen für `unordered_map<Key, T>`

| Datentyp | Bedeutung |
|---|---|
| `pointer` | Zeiger auf Map-Element |
| `const_pointer` | nur lesend verwendbarer Zeiger |
| `key_type` | entspricht `Key` |
| `value_type` | entspricht `pair<const Key, T>` |
| `mapped_type` | entspricht `T` |
| `hasher` | Hash-Funktionsobjekt |
| `key_equal` | Vergleichs-Funktionsobjekt, entspricht `Pred` in der Deklaration oben |
| `local_iterator` | lokaler Iterator für ein Bucket |
| `const_local_iterator` | dito (siehe Text) |

Die in einer `unordered_map` abgelegten Elemente sind Schlüssel/Wert-Paare. Weil der Schlüssel zur Berechnung der Adresse verwendet wird, kann mit einem Iterator, ob `const` oder nicht, auf das erste Element des Paars (den Schlüssel) *nur lesend* zugegriffen werden. Andernfalls wäre die Adressberechnung hinfällig. Das zweite Element, z.B. `(*einIterator).second`, ist veränderbar, wenn `einIterator` vom nicht-`const`-Typ `iterator` oder `local_iterator` ist. Das Beispiel unten ist eine Variation des Programms von Seite 786 für eine `unordered_map`.

Der Belegungsfaktor und die Anzahl der Buckets lassen sich mit den entsprechenden Funktionen ermitteln. Mit Hilfe eines `local_iterator`-Objekts können alle Schlüssel mit demselben Hash-Wert angezeigt werden. Das Programmfragment zeigt dies, indem es den Inhalt aller nicht-leeren Buckets anzeigt und nur nach jedem Bucket eine neue Zeile anfängt. Mehrfache Einträge pro Bucket stehen also in einer Zeile:

**Listing 28.9:** Hash-Map

```
// cppbuch/k28/unorderedmap.cpp
#include<unordered_map>
#include<string>
#include<iostream>
using namespace std;

// Zwei typedefs zur Abkürzung
typedef unordered_map<string, long> MapType;
typedef MapType::value_type ValuePair;

int main() {
   MapType aMap;
   aMap.insert(ValuePair("Thomas", 5192835));
   aMap.insert(ValuePair("Werner", 24439404));
   aMap.insert(ValuePair("Manfred", 535353));
   aMap.insert(ValuePair("Heiko", 635352723));
   aMap.insert(ValuePair("Andreas", 42536347));
   aMap.insert(ValuePair("Karin", 9273539));
   // 2. Einfügen von Heiko mit einer anderen Telefonnummer wird
   // NICHT ausgeführt, weil der Schlüssel schon existiert.
```

```
   aMap.insert(ValuePair("Heiko", 1000000));

   // Wegen der internen Hash-Struktur ist die Ausgabe unsortiert.
   cout << "Ausgabe:\n";
   auto iter = aMap.begin();
   while(iter != aMap.end()) {
      cout << (*iter).first << ':'   // Name
           << (*iter).second << endl; // Nummer
        ++iter;
   }
   cout << "Ausgabe der Nummer nach Eingabe des Namens\n"
        << "Name: ";
   string derName;
   cin >> derName;
   cout << "Suche mit Iterator: ";
   iter = aMap.find(derName);      // O(1)
   if(iter != aMap.end()) {
      cout << (*iter).second << endl;
   }
   else {
      cout << "Nicht gefunden!" << endl;
   }
   cout << "Belegungsfaktor = " << aMap.load_factor() << endl;
   cout << "Anzahl der Buckets = " << aMap.bucket_count()
        << ". Belegt sind:" << endl;
   for(size_t i=0; i < aMap.bucket_count(); ++i) {
      if(aMap.bucket_size(i) > 0) {
         cout << "Bucket " << i << " : " ;
         auto locIter = aMap.begin(i); // auto: MapType::const_local_iterator
         while(locIter != aMap.end(i)) {
            cout << (*locIter).first << ' '
                 << (*locIter).second << " ";
            ++locIter;
         }
         cout << endl;
      }
   }
}
```

Hier ist `MapType` wie oben definiert. Außer den Methoden der Tabelle 28.3 auf Seite 766 gibt es noch die folgenden – wenn nicht anders angegeben, ist die Zeitkomplexität $O(1)$ und mit $O(n)$ ist $O(size())$ gemeint:

- `unordered_map(size_type n = X, const hasher& hf = hasher(), const key_equal& eql = key_equal())`
  Konstruktor. Die vorgegebenen Parameter haben folgende Bedeutung:
  - `n` ist die Anzahl der Buckets. Wenn für `n` nichts angegeben wird, ist `X` eine implementationsabhängige Zahl.
  - `hf`: Hash-Funktionsobjekt
  - `eql`: Funktionsobjekt zum Prüfen zweier Elemente auf Gleichheit.

Wenn der Schlüssel einer der Grunddatentypen, ein Zeiger oder vom Typ `string` ist, ist die einfachste Erzeugung nur durch Angabe von Schlüssel- und Datentyp möglich, wie oben im Beispiel gezeigt.

- `template<class InputIterator>`
  `unordered_map(InputIterator first, InputIterator last,`
  `size_type n = X, const hasher& hf = hasher(),`
  `const key_equal& eql = key_equal())`
  Eine Hash-Map kann mit Daten initialisiert werden, wenn dem Konstruktor ein Bereich von Schlüssel/Wert-Paaren in Form von Iteratoren übergeben wird. Die Zeitkomplexität ist im Durchschnitt $O(n)$ mit $n = last - first$, sie kann im schlechtesten Fall $O(n^2)$ sein. Ursache wäre eine für die reale Schlüsselverteilung gänzlich ungeeignete Hash-Funktion, die alle vorkommenden Schlüssel demselben Bucket zuordnet.
- `hasher hash_function() const`
  gibt eine Kopie des Funktionsobjekts zurück, das zur Hash-Berechnung verwendet wird.
- `key_equal key_eq() const`
  gibt eine Kopie des Funktionsobjekts zurück, das zum Test auf Gleichheit zweier Elemente verwendet wird.
- `pair<iterator, bool> insert(const value_type& x)`
  fügt das Schlüssel/Daten-Paar `x` ein, sofern ein Element mit dem entsprechenden Schlüssel noch nicht vorhanden ist. Der Iterator des zurückgegebenen Paares zeigt auf das eingefügte Element bzw. auf das Element mit demselben Schlüssel wie `x`. Der Wahrheitswert zeigt, ob überhaupt ein Einfügen stattgefunden hat – es könnte ja sein, dass der Schlüssel bereits existiert. Die Zeitkomplexität ist im Durchschnitt $O(1)$, im schlechtesten Fall $O(n)$.
- `iterator insert(iterator p, const value_type& x)`
  wie `insert(const value_type& x)`, wobei der Iterator `p` ein Hinweis sein soll, wo die Suche zum Einfügen beginnen soll. Der zurückgegebene Iterator zeigt auf das eingefügte Element bzw. auf das mit demselben Schlüssel wie `x`. Die Zeitkomplexität ist im Durchschnitt $O(1)$, im schlechtesten Fall $O(n)$.
- `void insert(InputIterator i, InputIteratorj)`
  fügt die Elemente aus dem Iteratorbereich `[i, j)` ein. Die Zeitkomplexität ist im Durchschnitt $O(j - i)$, im schlechtesten Fall $O(n(j - i))$.
- `size_type erase(const key_type& k)`
  alle Elemente mit einem Schlüssel gleich `k` löschen. Es wird die Anzahl $N$ der gelöschten Elemente (hier: 0 oder 1, bei einer `unordered_multimap` auch mehr) zurückgegeben. Die Zeitkomplexität ist im Durchschnitt $O(1)$, im schlechtesten Fall $O(n)$.
- `iterator erase(iterator p)`
  das Element löschen, auf das der Iterator `p` zeigt. Der zurückgegebene Iterator zeigt anschließend auf das `p` folgende Element, wenn es existiert, ansonsten wird ein End-Iterator (`end()`) zurückgegeben.
- `iterator erase(iterator p, iterator q)`
  alle Elemente im Iteratorbereich `[p, q)` löschen. Der zurückgegebene Iterator zeigt anschließend auf das `q` folgende Element, wenn es existiert, ansonsten wird ein End-Iterator (`end()`) zurückgegeben. Die Zeitkomplexität ist im Durchschnitt $O(q - p)$.

- `const_iterator find(const key_type& k) const` und
  `iterator find(const key_type& k)`
  geben einen Iterator auf ein Element mit dem Schlüssel `k` zurück, falls vorhanden. Andernfalls wird `end()` zurückgegeben. Die Zeitkomplexität ist im Durchschnitt $O(1)$, im schlechtesten Fall $O(n)$.
- `size_type count(const key_type& k) const`
  gibt die Anzahl der Elemente (hier: 0 oder 1, bei einer `unordered_multimap` auch mehr) mit dem Schlüssel `k` zurück. Die Zeitkomplexität ist im Durchschnitt $O(1)$, im schlechtesten Fall $O(n)$.
- `pair<const_iterator, const_iterator>`
  `equal_range(const key_type& k) const` und
  `pair<iterator, iterator> equal_range(const key_type& k)`
  geben ein Paar von Iteratoren zurück, zwischen denen die Schlüssel gleich `k` sind. Die Differenz der Iteratoren kann (anders als bei `unordered_multimap`) nur 0 oder 1 sein. Die Zeitkomplexität ist im Durchschnitt $O(1)$, im schlechtesten Fall $O(n)$.
- `T& operator[](const key_type& k)`
  gibt eine Referenz auf die zum Schlüssel `k` gehörenden Daten zurück. Falls der Schlüssel nicht existiert, wird zu diesem Schlüssel ein Objekt mit dem Konstruktor `T()` für den Wertteil angelegt. Die Zeitkomplexität ist im Durchschnitt $O(1)$, im schlechtesten Fall $O(n)$.
- `size_type bucket_count() const`
  gibt die aktuelle Anzahl der Buckets zurück.
- `size_type max_bucket_count() const`
  gibt die maximal mögliche Anzahl der Buckets zurück.
- `size_type bucket_size(size_type n) const`
  gibt die Anzahl der im Bucket `n` gespeicherten Elemente zurück. Die Funktion kann hilfreich sein, um eine geeignete Hash-Funktion zu finden.
- `size_type bucket(const key_type& k) const`
  gibt die Nummer des Buckets zurück, in dem die Elemente mit dem Schlüssel `k` gespeichert sind.
- `const_local_iterator begin(size_type n) const` und
  `local_iterator begin(size_type n)`
  geben einen Iterator auf den Bucket mit der Nummer `n` zurück.
- `const_local_iterator end(size_type n) const` und
  `local_iterator end(size_type n)`
  geben den End-Iterator des Buckets mit der Nummer `n` zurück.
- `float load_factor() const`
  gibt den Belegungsfaktor zurück.
- `void max_load_factor(float f) const`
  setzt den maximal gewünschten Belegungsfaktor. Falls dieser Faktor überschritten zu werden droht, wird die Anzahl der Buckets erhöht.
- `float max_load_factor() const`
  gibt den maximal gewünschten Belegungsfaktor zurück.

- `void rehash(size_type n)`
  reorganisiert den Container, sodass er danach mindestens `n` Buckets hat und der Belegungsfaktor unter dem maximal gewünschten liegt. Die Zeitkomplexität ist im Durchschnitt $O(n)$, im schlechtesten Fall $O(n^2)$.

### 28.4.2 unordered_multimap

Die Klasse `unordered_multimap` unterscheidet sich von der Klasse `unordered_map` dadurch, dass *mehrfache* Einträge von Elementen mit identischen Schlüsseln möglich sind. Die Datentypen und Methoden entsprechen denen der oben dargestellten Klasse `unordered_map` mit folgenden wenigen Unterschieden:

- Es gibt keinen Index-Operator `T& operator[](const key_type& x)`.
- Die `insert()`-Methoden fügen Schlüssel/Daten-Paare ein, auch wenn ein Element mit dem entsprechenden Schlüssel schon vorhanden ist.
- Die Zeitkomplexität mancher Methoden (wie zum Beispiel `equal_range(const key_type& k)`) erhöht sich im Vergleich zu einer `unordered_map`, wenn es mehrere Einträge mit identischen Schlüsseln gibt.

### 28.4.3 unordered_set

Die Klasse `unordered_set` für Mengen entspricht der Klasse `unordered_map`, nur dass Schlüssel und Daten zusammenfallen. Das heißt, es werden nur Schlüssel gespeichert. Dabei ist der Schlüssel eindeutig, es kann also keine zwei Datensätze zu demselben Schlüssel geben. Im Gegensatz zu `set` ist die Speicherung nicht sortiert. Die Deklaration ist

```
template<class Value,                    // Typ der Schlüssel
         class Hash = hash<Key>,         // Hash-Funktion
         class Pred = std::equal_to<Key> > // für Elementvergleich
                                         // siehe Seite 753
class unordered_set;
```

`hash<Key>` ist ein Typ für ein Funktionsobjekt, das eine Zahl des Typs `size_t` als Hash-Wert zurückgibt, vgl. Seite 793. Ein `unordered_set`-Container stellt unter anderem die folgenden öffentlichen Datentypen und Methoden bereit:

- Datentypen der Tabelle 28.2 (Seite 765)
- Methoden der Tabelle 28.3 (Seite 766)

Relationale Operatoren sind nicht definiert. Es gibt weitere öffentliche Datentypen, die in der Tabelle 28.9 aufgelistet sind. Außer den Methoden der Tabelle 28.3 gibt es noch die folgenden – wenn nicht anders angegeben, ist die Zeitkomplexität $O(1)$ und mit $O(n)$ ist $O(size())$ gemeint:

- `unordered_set(size_type n = X, const hasher& hf = hasher(),`
  `const key_equal& eql = key_equal())`
  Konstruktor. Die vorgegebenen Parameter haben die auf Seite 795 angegebene Bedeutung.
- `template<class InputIterator>`
  `unordered_set(InputIterator first, InputIterator last,`
  `size_type n = X, const hasher& hf = hasher(),`
  `const key_equal& eql = key_equal())`

**Tabelle 28.9:** Zusätzliche Datentypen für `unordered_set<Key>`

| Datentyp | Bedeutung |
|---|---|
| `pointer` | Zeiger auf Set-Element |
| `const_pointer` | nur lesend verwendbarer Zeiger |
| `key_type` | entspricht `Value` |
| `value_type` | dito |
| `hasher` | Hash-Funktionsobjekt |
| `key_equal` | Vergleichs-Funktionsobjekt, entspricht `Pred` in der Deklaration oben |
| `local_iterator` | lokaler Iterator für ein Bucket |
| `const_local_iterator` | dito (vgl. Seite 794) |

Ein Hash-Set kann mit Daten initialisiert werden, wenn dem Konstruktor ein Bereich von Schlüsseln in Form von Iteratoren übergeben wird. Die Zeitkomplexität ist im Durchschnitt $O(n)$ mit $n = last - first$, sie kann wie bei der `unordered_map` im schlechtesten Fall $O(n^2)$ sein.

- `hasher hash_function() const`
  gibt eine Kopie des Funktionsobjekts zurück, das zur Hash-Berechnung verwendet wird.
- `key_equal key_eq() const`
  gibt eine Kopie des Funktionsobjekts zurück, das zum Test auf Gleichheit zweier Schlüssel verwendet wird.
- `pair<iterator, bool> insert(const value_type& x)`
  fügt den Schlüssel `x` ein, sofern er noch nicht vorhanden ist. Der Iterator des zurückgegebenen Paares zeigt auf das eingefügte Element bzw. auf den ggf. schon vorhandenen Schlüssel. Der Wahrheitswert zeigt, ob überhaupt ein Einfügen stattgefunden hat – es könnte ja sein, dass der Schlüssel bereits existiert. Die Zeitkomplexität ist im Durchschnitt $O(1)$, im schlechtesten Fall $O(n)$.
- `iterator insert(iterator p, const value_type& x)`
  wie `insert(const value_type& x)`, wobei der Iterator `p` ein Hinweis ist, wo die Suche zum Einfügen beginnen soll. Der zurückgegebene Iterator zeigt auf das eingefügte Element bzw. auf den schon vorhandenen Schlüssel. Die Zeitkomplexität ist im Durchschnitt $O(1)$, im schlechtesten Fall $O(n)$.
- `void insert(InputIterator i, InputIteratorj)`
  fügt die Schlüssel aus dem Iteratorbereich `[i, j)` ein. Die Zeitkomplexität ist im Durchschnitt $O(j - i)$, im schlechtesten Fall $O(n(j - i))$.
- `size_type erase(const key_type& k)`
  alle Elemente mit einem Schlüssel gleich `k` löschen. Es wird die Anzahl $N$ der gelöschten Elemente (hier: 0 oder 1, bei einem `unordered_multiset` auch mehr) zurückgegeben. Die Zeitkomplexität ist im Durchschnitt $O(1)$, im schlechtesten Fall $O(n)$.
- `iterator erase(iterator p)`
  das Element löschen, auf das der Iterator `p` zeigt. Der zurückgegebene Iterator zeigt anschließend auf das `p` folgende Element, wenn es existiert, ansonsten wird ein End-Iterator (`end()`) zurückgegeben.

- `iterator erase(iterator p, iterator q)`
  alle Elemente im Iteratorbereich `[p, q)` löschen. Der zurückgegebene Iterator zeigt anschließend auf das `q` folgende Element, wenn es existiert, ansonsten wird ein End-Iterator (`end()`) zurückgegeben. Die Zeitkomplexität ist im Durchschnitt $O(q - p)$.
- `const_iterator find(const key_type& k) const` und
  `iterator find(const key_type& k)`
  geben einen Iterator auf den Schlüssel `k` zurück, falls vorhanden. Andernfalls wird `end()` zurückgegeben. Die Zeitkomplexität ist im Durchschnitt $O(1)$, im schlechtesten Fall $O(n)$.
- `size_type count(const key_type& k) const`
  gibt die Anzahl der Schlüssel (hier: 0 oder 1, bei einem `unordered_multiset` auch mehr) mit dem Wert `k` zurück. Die Zeitkomplexität ist im Durchschnitt $O(1)$, im schlechtesten Fall $O(n)$.
- `pair<const_iterator, const_iterator>`
  `equal_range(const key_type& k) const` und
  `pair<iterator, iterator> equal_range(const key_type& k)`
  geben ein Paar von Iteratoren zurück, zwischen denen die Schlüssel gleich `k` sind. Die Differenz der Iteratoren kann (anders als bei `unordered_multiset`) nur 0 oder 1 sein. Die Zeitkomplexität ist im Durchschnitt $O(1)$, im schlechtesten Fall $O(n)$.
- `size_type bucket_count() const`
  gibt die aktuelle Anzahl der Buckets zurück.
- `size_type max_bucket_count() const`
  gibt die maximal mögliche Anzahl der Buckets zurück.
- `size_type bucket_size(size_type n) const`
  gibt die Anzahl der im Bucket `n` gespeicherten Elemente zurück.
- `size_type bucket(const key_type& k) const`
  gibt die Nummer des Buckets zurück, in dem die Elemente mit dem Schlüssel `k` gespeichert sind.
- `const_local_iterator begin(size_type n) const` und
  `local_iterator begin(size_type n)`
  geben einen Iterator auf den Bucket mit der Nummer `n` zurück.
- `const_local_iterator end(size_type n) const` und
  `local_iterator end(size_type n)`
  geben den End-Iterator des Buckets mit der Nummer `n` zurück.
- `float load_factor() const`
  gibt den Belegungsfaktor zurück.
- `void max_load_factor(float f) const`
  setzt den maximal gewünschten Belegungsfaktor. Falls dieser Faktor überschritten zu werden droht, wird die Anzahl der Buckets erhöht.
- `float max_load_factor() const`
  gibt den maximal gewünschten Belegungsfaktor zurück.

- `void rehash(size_type n)`
  reorganisiert den Container, sodass er danach mindestens `n` Buckets hat und der Belegungsfaktor unter dem maximal gewünschten liegt. Die Zeitkomplexität ist im Durchschnitt $O(n)$, im schlechtesten Fall $O(n^2)$.

### 28.4.4 unordered_multiset

Die Klasse `unordered_multiset` unterscheidet sich von der Klasse `unordered_set` dadurch, dass *mehrfache* Einträge identischer Schlüssel möglich sind. Die Datentypen und Methoden entsprechen denen der Klasse `set` mit folgender Ausnahme:

- Die `insert()`-Methoden fügen Schlüssel/Daten-Paare ein, auch wenn ein Element mit dem entsprechenden Schlüssel schon vorhanden ist.

## 28.5 bitset

Der Header `<bitset>` definiert eine Template-Klasse und zugehörige Funktionen zur Darstellung und Bearbeitung von Bitfolgen fester Größe. Die Deklaration der Klasse ist

```
template<size_t N> class bitset;
```

Wie bei dem Beispiel eines Stacks mit statisch festgelegter Größe in Abschnitt 6.3.2 ab Seite 249 ist `N` die Anzahl der zu speichernden Bits. Es gibt die folgenden `std`-globalen Operatoren, die den Bitoperationen mit `int`-Zahlen entsprechen:

```
template <size_t N>
  bitset<N> operator&(const bitset<N>&, const bitset<N>&);
template <size_t N>
  bitset<N> operator|(const bitset<N>&, const bitset<N>&);
template <size_t N>
  bitset<N> operator^(const bitset<N>&, const bitset<N>&);
```

Dazu kommen Operatoren zur Ein- und Ausgabe:

```
template <size_t N>
  istream& operator>>(istream& is, bitset<N>& x);
template <size_t N>
  ostream& operator<<(ostream& os, const bitset<N>& x);
```

Die beiden letzten Deklarationen sorgen dafür, dass die Eingabe und die Ausgabe mit Streams auf die bekannte einfache Art möglich sind, etwa:

```
bitset<13817> B;   // 13817 Bits
cin >> B;          // Eingabe
cout << B;         // Ausgabe
```

Die Klasse `bitset` stellt den öffentlichen Datentyp `reference` für Manipulationen an einzelnen Bits bereit, ähnlich wie die Klasse `vector<bool>`:

```
class reference {
   friend class bitset;
   reference();
 public:
   ~reference();
   reference& operator=(bool x);       // für b[i] = x;
   reference& operator=(const reference&); // für b[i] = b[j];
   bool operator~() const;             // negiert das Bit
   operator bool() const;              // für x = b[i];
   reference& flip();                  // für b[i].flip();
};
```

Die Klasse `bitset` kennt die folgenden Konstruktoren und Methoden:

- `bitset()` ist der Standardkonstruktor. Alle Bits werden zu 0 initialisiert.
- `bitset(unsigned long val)`
  Dieser Konstruktor initialisiert die ersten *m* Positionen entsprechend der Bitwerte im Parameter `val`. *m* ist 8· `sizeof(unsigned long)`, falls ein Byte 8 Bits entspricht. Falls *m* < `N` ist, werden die restlichen Bits zu 0 initialisiert. Falls *m* > `N` ist, werden nur `N` Positionen initialisiert.
- `bitset(const string& str, size_t pos = 0, size_t n = string::npos)`
  Der Bitset wird mit dem String `str`, beginnend an der Position `pos`, initialisiert. Der String darf nur aus den Zeichen '0' und '1' bestehen. Der Parameter `n` bestimmt die maximale Anzahl der auszuwertenden Zeichen. Auch hier werden restliche Positionen zu 0 gesetzt, und es werden insgesamt nicht mehr als `N` Positionen initialisiert.
- `bitset<N>& operator&=(const bitset<N>& B)`
  löscht jedes Bit, dessen Entsprechung im Bitset `B` nicht gesetzt ist. Die anderen Bits bleiben unverändert. Es wird das aufrufende Objekt zurückgegeben (`*this`).
- `bitset<N>& operator|=(const bitset<N>& B)`
  setzt jedes Bit, dessen Entsprechung im Bitset `B` gesetzt ist. Die anderen Bits bleiben unverändert. Es wird `*this` zurückgegeben.
- `bitset<N>& operator^=(const bitset<N>& B)`
  negiert jedes Bit, dessen Entsprechung im Bitset `B` gesetzt ist. Die anderen Bits bleiben unverändert. Es wird `*this` zurückgegeben.
- `bitset<N>& operator<<=(size_t n)`
  verschiebt alle Bits um `n` Positionen nach links. Dabei werden von rechts Null-Bits nachgezogen. Es wird `*this` zurückgegeben.
- `bitset<N> operator<<(size_t n) const`
  gibt einen Bitset zurück, in dem relativ zum aufrufenden Objekt alle Bits um `n` Positionen nach links verschoben sind. Im zurückgegebenen Ergebnis sind `n` Bits rechts auf Null gesetzt.
- `bitset<N>& operator>>=(size_t n)`
  verschiebt alle Bits um `n` Positionen nach rechts. Dabei werden von rechts Null-Bits nachgezogen. Es wird `*this` zurückgegeben.

- `bitset<N> operator>>(size_t n) const`
  gibt einen Bitset zurück, in dem relativ zum aufrufenden Objekt alle Bits um `n` Positionen nach rechts verschoben sind. Im zurückgegebenen Ergebnis sind `n` Bits links auf Null gesetzt.
- `bitset<N>& set()` setzt alle Bits. Es wird `*this` zurückgegeben.
- `bitset<N>& set(size_t n, int val = 1)`
  setzt das Bit an der Position `n`, falls `val` nicht angegeben wird oder ungleich Null ist. Ist `val` gleich 0, wird das Bit auf 0 zurückgesetzt. Es wird `*this` zurückgegeben.
- `bitset<N>& reset()` löscht alle Bits. Es wird `*this` zurückgegeben.
- `bitset<N>& reset(size_t n)`
  löscht das Bit an der Position `n`. Es wird `*this` zurückgegeben.
- `bitset<N>& flip()` negiert alle Bits. Es wird `*this` zurückgegeben.
- `bitset<N>& flip(size_t n)`
  negiert das Bit an der Position `n`. Es wird `*this` zurückgegeben.
- `bitset<N> operator~() const`
  gibt eine Kopie des aufrufenden Objekts zurück, in der alle Bits negiert sind.
- `reference operator[](size_t n)`
  gibt das Bit Nr. `n` als Objekt der Klasse `bitset<N>::reference` zurück (siehe Seite 801).
- `unsigned long to_ulong() const`
  gibt das aufrufende `bitset`-Objekt als `unsigned long`-Zahl zurück. Dabei darf `N` nicht größer als die Anzahl der Bits sein, die eine `unsigned long`-Zahl aufnehmen kann.
- `string to_string() const`
  gibt das aufrufende `bitset`-Objekt als String, bestehend aus Nullen und Einsen, zurück. Das erste Zeichen des Strings (Position 0) entspricht dem höchstwertigen Bit.
- `size_t count() const` gibt die Anzahl der gesetzten Bits zurück.
- `size_t size() const` gibt die Anzahl aller Bits (d.h. `N`) zurück.
- `bool operator==(const bitset<N>& B) const`
  gibt zurück, ob alle Bits mit denen von `B` übereinstimmen.
- `bool operator!=(const bitset<N>& B) const`
  gibt zurück, ob wenigstens ein Bit mit denen von `B` nicht übereinstimmt.
- `bool test(size_t n) const` gibt zurück, ob das Bit `n` gesetzt ist.
- `bool any() const` gibt zurück, ob wenigstens ein Bit gesetzt ist.
- `bool none() const` gibt zurück, ob alle Bits 0 sind.

Das folgende kleine Programm gibt *111111111101111* und *000011111111110* aus:

**Listing 28.10:** Bitset

```
// cppbuch/k28/bitset.cpp
#include<bitset>
#include<iostream>
using namespace std;
int main() {
  bitset<15> einBitset;
  einBitset.set();          // alles 1
  einBitset[4].flip();      // Bit 4 wird 0
  cout << einBitset << endl;
```

```
   einBitset >>= 4; // Rechtsverschiebung um 4 Positionen
   cout << einBitset << endl;
}
```

## Übungen

**28.1** Wenn eine Funktion aufgerufen wird, werden die lokalen Variablen auf dem C++-Laufzeitstack abgelegt. Nach Rückkehr der Funktion werden die Variablen wieder vom Stack geholt, und damit wird die Umgebung des Aufrufers wiederhergestellt. Daraus ergibt sich, dass jeder Funktionsaufruf auch mithilfe eines eigenen Stacks simuliert werden kann: 1. lokale Variablen auf dem Stack sichern; 2. Code mit Variablen der Funktion ausführen; 3. lokale Variablen restaurieren. Also lassen sich auch jegliche rekursive Aufrufe einer Funktion simulieren und damit ersetzen. Aufgabe: Eliminieren Sie die noch verbliebene Rekursion in der Lösung zu Aufgabe 3.2 von Seite 111, indem Sie ein Objekt der Klasse `stack` zur Verwaltung der aktuellen Variablen verwenden.

**28.2** Tragen Sie Prominente als Rang/Name-Kombination in eine Priority-Queue `promis` ein, etwa

```
promis.push(make_pair(8, "Peter Jackson"));          // oder
promis.push(pair<int, string>(10, "Tina Turner")); // ... usw.
```

Zu `make_pair()` und `pair` siehe Abschnitt 27.3. Die Zahl soll den Rang oder die vermutete Wichtigkeit des jeweiligen Stars in den einschlägigen Illustrierten bedeuten: je größer, desto wichtiger. Leeren Sie die Queue und zeigen Sie dabei die Prominenten auf dem Bildschirm an, geordnet nach ihrem Rang, und die wichtigsten zuerst.

**28.3** Wie muss die Deklaration der Priority-Queue `promis` lauten, wenn zuerst die weniger wichtigen Promis ausgegeben werden sollen?

**28.4** Lösen Sie das Problem der beiden vorstehenden Aufgaben, indem Sie anstelle einer Priority-Queue ein Objekt der Klasse `multimap` verwenden.