29

Iteratoren

Dieses Kapitel behandelt die folgenden Themen:

- Iterator-Kategorien
- Typinformation mit Traits
- Abstand von Iteratoren
- Iteratoren zum Einfügen
- Stream-Iteratoren

Iteratoren werden in Abschnitt 11.2 besprochen und an Beispielen gezeigt. Hier geht es um die vordefinierten Iteratortypen im Header `<iterator>` der C++-Standardbibliothek, die wie die Standard-Container als Templates realisiert sind und über `traits`-Klassen (*traits*, dt. etwa Eigenschaft) bestimmte öffentliche Typnamen zur Verfügung stellen. Natürlich könnten Typnamen auch direkt von einer Iteratorklasse veröffentlicht werden, aber man geht einen anderen Weg, weil die Algorithmen der C++-Standardbibliothek nicht nur auf Containern, die Typnamen bereitstellen, sondern auch auf einfachen C-Arrays arbeiten können sollen. Die damit arbeitenden Iteratoren sind aber nichts anderes als Zeiger, möglicherweise auf Grunddatentypen wie `int`. Ein Iterator des Typs `int*` kann sicher keine Typnamen zur Verfügung stellen. Aus diesem Grund gibt es eine Spezialisierung der `traits`-Klassen speziell für Zeiger.

```
// vom Iterator abgeleitete öffentliche Typen:
template<class Iterator>
struct iterator_traits {
```

```
  typedef typename Iterator::difference_type difference_type;
  typedef typename Iterator::value_type value_type;
  typedef typename Iterator::pointer pointer;
  typedef typename Iterator::reference reference;
  typedef typename Iterator::iterator_category iterator_category;
};
```

Damit ein Algorithmus, der mit Zeigern arbeitet, die üblichen Typnamen verwenden kann, wird das obige Template für Zeiger spezialisiert:

```
// partielle Spezialisierung (für Zeiger)
template<class T>
struct iterator_traits<T*> {
  typedef ptrdiff_t difference_type;
  typedef T value_type;
  typedef T* pointer;
  typedef T& reference;
  typedef random_access_iterator_tag iterator_category;
};
```

Die Iterator-Kategorie wird im nächsten Abschnitt erläutert, auch konkrete Anwendungsbeispiele für Traits sind dort zu finden. In der C++-Standardbibliothek wird ein Datentyp für Iteratoren angegeben, von dem jeder benutzerdefinierte Iterator erben kann:

```
namespace std {
  template<class Category, class T, class Distance = ptrdiff_t,
           class Pointer = T*, class Reference = T&>
  struct iterator {
    typedef Distance difference_type;
    typedef T value_type;
    typedef Pointer pointer;
    typedef Reference reference;
    typedef Category iterator_category; // siehe Abschnitt 29.1
  };
}
```

Durch `public`-Vererbung sind diese Namen in allen abgeleiteten Klassen sicht- und verwendbar.

29.1 Iterator-Kategorien

Es gibt verschiedene Kategorien von Iteratoren in einer hierarchischen Anordnung.

- **Input-Iterator**: Ein Input-Iterator ist zum sequenziellen Lesen von Daten gedacht, zum Beispiel aus einem Container oder aus einer Datei. Ein Zurückspringen an eine schon gelesene Stelle ist nicht möglich (der ---Operator ist nicht definiert).
- **Output-Iterator**: Ein Output-Iterator kann sequenziell in einen Container oder in eine Datei schreiben, wobei der Dereferenzierungsoperator verwendet wird. Beispiel:

```
// »Ausgabe« ist ein Output-Iterator
*Ausgabe++ = Wert;  // in die Ausgabe schreiben und weiterschalten
```

- **Forward-Iterator**: Wie Input- und Output-Iterator kann der Forward-Iterator sich vorwärts bewegen. Im Unterschied zu den vorgenannten Iteratoren können jedoch Werte des Iterators gespeichert werden, um ein Element des Containers wiederzufinden. Damit ist ein mehrfacher Durchlauf in eine Richtung möglich, zum Beispiel durch eine einfach verkettete Liste, wenn man sich den Anfang gemerkt hat.
- **Bidirectional-Iterator**: Ein Bidirectional-Iterator kann all das, was ein Forward-Iterator kann. Darüber hinaus kann er noch mit dem `--`-Operator *rückwärts* gehen, sodass er zum Beispiel für eine doppelt verkettete Liste geeignet ist.
- **Random-Access-Iterator**: Ein Random-Access-Iterator kann alles, was ein Bidirectional-Iterator kann. Zusätzlich ist ein wahlfreier Zugriff möglich, wie er für einen Vektor benötigt wird. Der wahlfreie Zugriff wird durch den Indexoperator `operator[]()` realisiert.

Tabelle 29.1 zeigt eine Übersicht über mögliche Operationen einer Kategorie.

Tabelle 29.1: Fähigkeiten der Iterator-Kategorien

Operation	Input	Output	Forward	Bidirectional	Random Access
`=`	●		●	●	●
`==`	●		●	●	●
`!=`	●		●	●	●
`*`	1)	2)	●	●	●
`->`	●		●	●	●
`++`	●	●	●	●	●
`--`				●	●
`[ ]`					3)
arithmetisch					4)
relational					5)

1) Dereferenzierung ist nur lesend möglich.
2) Dereferenzierung ist nur auf der linken Seite einer Zuweisung möglich.
3) `iter[n]` bedeutet `*(iter+n)` für einen Iterator `iter`
4) `+ += - -=` in Analogie zur Zeigerarithmetik
5) `< > <= >=` relationale Operatoren

Um einen Iterator mit einer Marke (englisch *tag*) zu versehen, gibt es die folgenden Markierungsklassen:

```
struct input_iterator_tag {};

struct output_iterator_tag {};

struct forward_iterator_tag
  : public input_iterator_tag {};

struct bidirectional_iterator_tag
  : public forward_iterator_tag {};

struct random_access_iterator_tag
 : public bidirectional_iterator_tag {};
```

29.1.1 Anwendung von Traits

In diesem Abschnitt wird gezeigt, wie mithilfe von Traits der Typ bestimmt und wie abhängig vom Typ der passende Algorithmus ausgewählt werden kann. Der Compiler wählt im folgenden Beispiel dazu unter überladenen Funktionen aus. Deren Parameter, die Iterator-Kategorie, wird aus dem Iterator mithilfe der Funktion `getIteratortyp()` ermittelt:

Listing 29.1: Iteratortyp bestimmen

```
// cppbuch/k29/ityp.cpp
#include<string>
#include<fstream>
#include<vector>
#include<iterator>
#include<iostream>
using namespace std;

// Funktions-Template zur Ermittlung des Typs (iterator-tag) des Iterators
template<class Iterator>
typename iterator_traits<Iterator>::iterator_category
getIteratortyp(const Iterator&) {
   typename iterator_traits<Iterator>::iterator_category
     typeobject;
   return typeobject;
}

// überladene Funktionen
void welcherIterator(const input_iterator_tag&) {
    cout << "Input-Iterator!" << endl;
}

void welcherIterator(const output_iterator_tag&) {
    cout << "Output-Iterator!" << endl;
}

void welcherIterator(const forward_iterator_tag&) {
    cout << "Forward-Iterator!" << endl;
}

void welcherIterator(const random_access_iterator_tag&) {
    cout << "Random-Access-Iterator!" << endl;
}

int main( ) {     // Anwendung
    // Bei Grunddatentypen (hier: ein Zeiger) wird das partiell spezialisierte
    // iterator_traits<T*>- Template von Seite 806 benutzt.
    int *ip;                      // Random-Access-Iterator
    // Anzeige des Iteratortyps
    welcherIterator(getIteratortyp(ip)); // oder:
    welcherIterator(iterator_traits<int*>::iterator_category());

    // Definition eines Datei-Objekts zum Lesen (eine tatsächliche Datei ist nicht
    // erforderlich, es geht nur um den Typ)
```

```
    ifstream Source;
    // Ein istream_iterator ist ein Input-Iterator
    istream_iterator<string> IPos(Source);
    // Anzeige des Iteratortyps
    welcherIterator(getIteratortyp(IPos)); // oder:
    welcherIterator(iterator_traits<istream_iterator<string> >
           ::iterator_category());

    // Definition eines Datei-Objekts zum Schreiben
    ofstream Destination;
    // Ein ostream_iterator ist ein Output-Iterator
    ostream_iterator<string> OPos(Destination);
    // Anzeige des Iteratortyps
    welcherIterator(getIteratortyp(OPos)); // oder:
    welcherIterator(iterator_traits<ostream_iterator<string> >
           ::iterator_category());

    vector<int> v(10);
    // Anzeige des Iteratortyps
    welcherIterator(getIteratortyp(v.begin())); // oder end()
    welcherIterator(iterator_traits<vector<int>::iterator>
           ::iterator_category());
}
```

Im zweiten Beispiel wird der am besten passende Algorithmus automatisch zur Compilierzeit ausgewählt. Es soll das mittlere Element eines Containers zurückgegeben werden. Bei einer Liste müssen dazu N/2 Elemente abgeklappert werden, es ist also eine Schleife notwendig. Bei einem Vektor nimmt man einfach das Element [N/2]. Dabei wird einer Funktion `mittleresElement(Iterator anfang, size_t n)` der Anfang des Containers als Iterator und die Anzahl der Elemente mitgeteilt. Aus dem Typ des Iterators wird die passende überladene aufzurufende Funktion ermittelt:

Listing 29.2: Algorithmus typabhängig auswählen

```
// cppbuch/k29/algorithmenwahl.cpp
#include<iostream>
#include<list>
#include<vector>
#include<iterator>

template<class Iterator>   // aufrufende Funktion
int mittleresElement(Iterator anfang, size_t n) {
   typename std::iterator_traits<Iterator>::iterator_category
      typeobject;
   return holeMittleres(anfang, n, typeobject);
}

template<class Iterator>   // erste überladene Funktion
int holeMittleres(Iterator anfang, size_t n,
                  std::bidirectional_iterator_tag) {
   for(size_t i=0; i < n/2; ++i) { // Schleife
      ++anfang;
   }
```

```
   return *anfang;
}

template<class Iterator>   // zweite überladene Funktion
int holeMittleres(Iterator anfang, size_t n,
                  std::random_access_iterator_tag) {
   return *(anfang + n/2);   // Arithmetik
}

using namespace std;
int main( ) { // Hauptprogramm
    list<int> lis;             // mit Werten füllen
    for(size_t i=0; i < 10; ++i) {
        lis.push_back(i);
    }
    vector<int> vec(10);       // mit Werten füllen
    for(size_t i = 0; i < vec.size(); ++i) {
        vec[i] = 10*i;
    }
    // Aufruf der ersten Implementierung für die Liste
    cout << mittleresElement(lis.begin(), lis.size()) << endl;
    // Aufruf der zweiten Implementierung für den Vektor
    cout << mittleresElement(vec.begin(), vec.size()) << endl;
}
```


29.2 distance() und advance()

Weil nur Random-Access-Iteratoren die Operationen + und - erlauben, gibt es die Funktionen `distance()` zum Ermitteln eines Iteratorabstands und `advance()` zum Weiterschalten. Diese Funktionen benutzen intern + und - für Random-Access-Iteratoren und in anderen Fällen ++ bzw. --. Die Deklarationen sind:

```
// advance() schaltet i um n Positionen vor bzw. zurück, falls n < 0 ist.
template<class InputIterator, class Distance>
void advance(InputIterator& i, Distance n);

// distance(first, last) gibt den Abstand zwischen zwei Iteratoren zurück.
// Dabei muss last von first aus erreichbar sein.
template<class InputIterator>
typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);
```

29.3 Reverse-Iteratoren

Ein Reverse-Iterator ist bei einem bidirektionalen Iterator immer möglich. Ein Reverse-Iterator durchläuft einen Container *rückwärts* mit der ++-Operation. Beginn und Ende eines Containers für Reverse-Iteratoren werden durch `rbegin()` und `rend()` markiert. Dabei verweist `rbegin()` auf das letzte Element des Containers und `rend()` auf die (ggf. fiktive) Position *vor* dem ersten Element. Einige Container stellen Reverse-Iteratoren zur Verfügung. Sie werden mit der vordefinierten Klasse

```
template<class Iterator> class reverse_iterator;
```

realisiert. Ein Objekt dieser Klasse wird mit einem bidirektionalen oder einem Random-Access-Iterator initialisiert, entsprechend dem Typ des Template-Parameters. Ein Reverse-Iterator arbeitet intern mit diesem Iterator und legt eine Hülle (englisch *wrapper*) mit bestimmten zusätzlichen Operationen um ihn herum. Für einen existierenden Iterator wird eine neue Schnittstelle geschaffen, um sich verschiedenen Gegebenheiten anpassen (englisch *to adapt*) zu können. Deshalb werden Klassen, die eine Klasse in eine andere umwandeln, *Adapter* genannt. Ein Beispiel sehen Sie unten. Zu den in der Tabelle 29.1 auf Seite 807 angegebenen Operationen für bidirektionale oder Random-Access-Iteratoren gibt es die Methode `base()`, die den gekapselten Iterator zurückgibt.

Listing 29.3: Reverse-Iterator

```
// cppbuch/k29/reverse.cpp
#include<vector>
#include<iostream>
#include<iterator>
using namespace std;

int main() {
  vector<int> v(10);
  for(size_t i = 0; i < v.size(); ++i) {
     v[i] = i;
  }
  reverse_iterator<vector<int>::iterator> revIter(v.rbegin());
  // Alternativ: vector<int>::reverse_iterator revIter(v.rbegin());
  while(revIter != v.rend()) { // Zahlen verdoppelt in umgekehrter Folge ausgeben
    *revIter *= 2;             // Wert über den Iterator ändern
    cout << *revIter++ << ' '; // nur lesender Zugriff
  }
  cout << endl;
  // Falls Werte NICHT geändert werden sollen, kann der von der Klasse vector
  // bereitgestellte Typ const_reverse_iterator verwendet werden. Eine eigene
  // Klasse const_reverse_iterator gibt es nicht [ISOC++].
  vector<int>::const_reverse_iterator constRevIter = v.rbegin(),
                                      constRevIterEnd(v.rend());
  while(constRevIter != constRevIterEnd) {
    cout << *constRevIter++ << ' ';
  }
}
```

29.4 Insert-Iteratoren

Mit normalen Iteratoren bewirkt der Code

```
while(first != last) {
   *result++ = *first++;
}
```

das Kopieren des Bereichs `[first, last)` an die Stelle `result`, wobei der vorherige Inhalt an der Stelle *überschrieben* wird. Derselbe Code bewirkt jedoch das *Einfügen* in einen Container, wenn `result` ein Insert-Iterator ist. Je nach gewünschter Position zum Einfügen gibt es drei Varianten:

1. `front_insert_iterator` (Beispiel siehe *cppbuch/k29/finsert.cpp*)
   Dieser Insert-Iterator fügt etwas am Anfang eines Containers ein. Der Container muss die Methode `push_front()` zur Verfügung stellen. Anwendungsbeispiel:

   ```
   list<int> dieListe;
   // ....
   front_insert_iterator<list<int> > frontInsIter(dieListe);
   int i = 1;
   while(i < 5) {
      *frontInsIter++ = i++;        // Zahlen vorne einfügen
   }
   ```

   `front_inserter()` ist eine Funktion, die einen `front_insert_iterator` zurückgibt. Ein Beispiel mit dem Standardalgorithmus `copy()` (siehe auch *cppbuch/k29/finsert.cpp*):

   ```
   // Einfügen aller Elemente im Bereich [a,b) am Anfang von dieListe.
   // a und b sind Iteratoren eines anderen Containers.
   copy(a, b, front_inserter(dieListe));
   ```

2. `back_insert_iterator` (Beispiel siehe *cppbuch/k29/binsert.cpp*)
   Dieser Insert-Iterator fügt etwas am Ende eines Containers ein. Der Container muss die Methode `push_back()` zur Verfügung stellen. Anwendungsbeispiel:

   ```
   list<int> dieListe;
   // ....
   back_insert_iterator<list<int> > backInsIter(dieListe);
   int i = 1;
   while(i < 5) {
      *backInsIter++ = i++;        // Zahlen anhängen
   }
   ```

   `back_inserter()` ist eine Funktion, die einen `back_insert_iterator` zurückgibt. Ein Beispiel mit dem Standardalgorithmus `copy()`:

   ```
   // Anhängen aller Elemente im Bereich [a,b) an das Ende von dieListe.
   // a und b sind Iteratoren eines anderen Containers.
   copy(a, b, back_inserter(dieListe));
   ```

3. `insert_iterator` (Beispiel siehe *cppbuch/k29/insert.cpp*)
   Dieser Insert-Iterator fügt etwas an einer ausgewählten Position in den Container ein. Der Container muss die Methode `insert()` zur Verfügung stellen. Die Anwendung

ist ähnlich wie vorher, nur dass dem Insert-Iterator die gewünschte Einfügeposition mitgegeben werden muss:

```
list<int> dieListe;
list<int>::iterator pos;
// .... hier pos an die gewünschte Stelle bringen
insert_iterator<list<int> > iter(dieListe, pos);
int i = 1;
while(i < 5) {
   *iter++ = i++;          // Zahlen bei pos einfügen
}
```

`inserter()` ist eine Funktion, die einen `insert_iterator` zurückgibt. Die Anwendung wird dadurch manchmal einfacher. Ein Beispiel mit dem Standardalgorithmus `copy()` von Seite 717:

```
// Einfügen aller Elemente im Bereich [a,b) an die Stelle pos von dieListe.
// a und b sind Iteratoren eines anderen Containers.
copy(a, b, inserter(dieListe, pos));
```

29.5 Stream-Iteratoren

Stream-Iteratoren dienen zum sequenziellen Lesen und Schreiben von Strömen mit den bekannten Operatoren << und >>. Der Istream-Iterator ist ein Input-Iterator, der Ostream-Iterator ein Output-Iterator. Beispiele:

```
// Anzeige aller durch Zwischenraumzeichen getrennten Zeichenfolgen:
ifstream quelle("Datei.txt");
istream_iterator<string> pos(quelle), ende;
while(pos != ende) {
  cout << *pos++ << endl;
}
```

Die Dereferenzierung von `pos` in der Schleife gibt den gelesenen Wert zurück. Durch Erben von der Klasse `istream_iterator` und Redefinieren einiger Methoden lassen sich eigene Istream-Iteratoren mit besonderen Eigenschaften schreiben. Dem Konstruktor eines OStream-Iterators kann wahlweise eine Zeichenkette zur Trennung von Elementen mitgegeben werden.

```
// Anzeige aller durch Zwischenraumzeichen getrennten Zeichenfolgen,
// wobei in der Ausgabe jede Zeile mit einem * versehen wird:
ifstream quelle("Datei.txt");
ofstream ziel("Ergebnis.txt");
istream_iterator<string> iPos(quelle), Ende;
ostream_iterator<string> oPos(ziel, "*\n");
while(iPos != Ende) {
   *oPos++ = *iPos++;
}
```

IStream-Iterator für Bezeichner

Im Folgenden wird als konkretes Beispiel ein IStream-Iterator, der die Bezeichner (englisch *identifier*) einer Datei liest. Weil der Eingabeoperator >> für die Klasse `string` schon existiert, kann ihm keine neue Bedeutung zugewiesen werden. Deshalb wird eine Klasse `Identifier` angelegt, die von `string` erbt und für die `operator>>()` definiert wird:

Listing 29.4: Leere Klasse Identifier

```
// cppbuch/k29/bezeichnerlesen/identifier.h
#ifndef IDENTIFIER_H
#define IDENTIFIER_H
#include<cctype>
#include<iostream>
#include<string>

class Identifier : public std::string { };

inline std::istream& operator>>(std::istream& is, Identifier& identifier) {
   identifier.assign(""); // alten Inhalt löschen
   char c = '\0';
   while(is && !(isalpha(c) || '_' == c)) { // Wortanfang finden
      is.get(c);
   }
   identifier += c;
   // Wenn der Anfang gefunden wurde, werden alle folgenden Unterstriche und alphanu-
   // merische Zeichen gesammelt. Whitespace oder Sonderzeichen beenden das Einlesen.
   while(is && (isalnum(c) || '_' == c)) {
      is.get(c);
      if(isalnum(c) || '_' == c)
         identifier += c;
   }    // Das letzte Zeichen gehört nicht zum Identifier und wird deshalb
   is.putback(c); // in den Stream zurückgeschrieben.
   return is;
}
#endif
```

In der Anwendung wird ein `istream_iterator` für diesen Typ erzeugt, der mit dem `ifstream` initialisiert wird. Das folgende Programm gibt die Bezeichner auf der Konsole aus:

Listing 29.5: Anwendung für IStream-Iterator

```
// cppbuch/k29/bezeichnerlesen/main.cpp
#include<iterator>
#include<fstream>
#include"identifier.h"
using namespace std;

int main( ) {
   ifstream datei("main.cpp"); // Eingabedatei öffnen
   istream_iterator<Identifier> iter(datei), end;
   while(iter != end) {
      cout << *iter++ << endl;
   }
}
```