11

Einführung in die Standard Template Library (STL)

Dieses Kapitel behandelt die folgenden Themen:

- Entstehung und Konzept
- Bezug zur C++-Standardbibliothek
- Wie wirken Container, Iteratoren und Algorithmen zusammen?

Einer der Erfolge von C++ beruht darauf, dass zahlreiche Bibliotheken am Markt vorhanden sind, die die Entwicklung von Programmen erheblich erleichtern, weil sie verlässliche und erprobte Komponenten anbieten. Eine besonders sorgfältig konstruierte Bibliothek ist die *Standard Template Library*, die bei Hewlett-Packard von Alexander Stepanov, Meng Lee und ihren Kollegen entwickelt wurde. Sie ist wegen ihrer überzeugenden Konzeption des Zusammenwirkens von Behälterklassen (englisch *Container*) und Algorithmen vom ISO-Komitee als Teil des C++-Standards akzeptiert worden. Durch das Aufgehen in der C++-Standardbibliothek hat die STL die Rolle als eigenständige Bibliothek verloren.

Der große Vorteil von Templates liegt auf der Hand. Die Auswertung der Templates geschieht zur Compilierzeit, es gibt keine Laufzeiteinbußen – etwa durch polymorphe Funktionszugriffe, falls die Generizität mit Vererbung realisiert wird. Der Vorteil der Standardisierung ist noch größer einzuschätzen. Programme, die eine standardisierte Bibliothek benutzen, sind leichter portierbar, weil jeder Compiler-Hersteller sich am Standard orien-

tieren wird. Darüber hinaus sind sie leichter wartbar, weil das entsprechende Know-how sehr viel verbreiteter ist als das Wissen über eine spezielle Bibliothek.

Der Schwerpunkt liegt auf *Algorithmen*, die mit *Containern* und *Iteratoren* zusammenarbeiten. Durch den Template-Mechanismus von C++ sind die Container für Objekte verschiedenster Klassen geeignet. Ein Iterator ist ein Objekt, das wie ein Zeiger auf einem Container bewegt werden kann, um auf das eine oder andere Objekt zu verweisen. Algorithmen arbeiten mit Containern, indem sie auf zugehörige Iteratoren zugreifen. Diese Konzepte werden weiter unten detailliert dargestellt.

Reduktion der Bibliothekskomponenten durch Templates

Die C++-Standardbibliothek rückt nicht Vererbung und Polymorphismus in den Vordergrund, sondern Container und Algorithmen für viele, auch benutzerdefinierte Datentypen, sofern sie einigen wenigen Voraussetzungen genügen. Die C++-Templates bieten die Grundlage dafür. Der Schwerpunkt liegt also nicht auf der Objektorientierung, sondern auf der generischen Programmierung. Damit ist der große Vorteil verbunden, dass die Anzahl der notwendigen verschiedenen Container- und Algorithmentypen drastisch reduziert wird. Ein weiterer Vorteil ist die Typsicherheit, weil Templates bereits zur Compilationszeit aufgelöst werden.

Nehmen wir zunächst an, dass es keine Templates gäbe und dass wir ein Element eines Datentyps `int` in einem Container vom Typ `vector` finden wollen. Dazu brauchen wir einen Algorithmus `find()`, der den Container durchsucht. Falls wir n verschiedene Container (Liste, Menge ...) haben, brauchen wir für jeden Container einen eigenen Algorithmus, also n `find()`-Algorithmen. Nun könnte es ja sein, dass wir nicht nur ein `int`-Objekt, sondern ein Objekt eines beliebigen, von m möglichen Datentypen suchen wollen. Damit würde die Anzahl der `find()`-Algorithmen auf $n \cdot m$ steigen. Diese Betrachtung soll für k verschiedene Algorithmen gelten, sodass insgesamt $k \cdot n \cdot m$ Algorithmen zu schreiben sind. Die Benutzung von Templates erlaubt es, die Anzahl m auf 1 zu reduzieren. Algorithmen der C++-Standardbibliothek arbeiten jedoch nicht direkt mit Containern, sondern nur mit Schnittstellenobjekten, den Iteratoren, die auf Container zugreifen. Iteratoren sind zeigerähnliche Objekte, die unten genau erklärt werden. Dadurch reduziert sich die notwendige Gesamtzahl auf $n + k$ statt $n \cdot k$, eine erhebliche Ersparnis.

11.1 Container, Iteratoren, Algorithmen

Zunächst werden die wichtigsten Elemente skizziert, ehe auf ihr Zusammenwirken eingegangen wird.

Container

Die C++-Standardbibliothek stellt verschiedene Arten von Containern zur Verfügung, die als Template-Klassen formuliert sind. Sie werden detailliert in Kapitel 28 beschrieben. Container sind Objekte, die zur Verwaltung anderer Objekte dienen, wobei es den

Benutzern überlassen bleibt, ob sie die Objekte per Wert oder per Referenz ablegen. Die Ablage per Wert meint, dass jedes Element des Containers ein Objekt eines kopierbaren Typs ist (Wertsemantik). Die Ablage per Referenz heißt, dass die Elemente des Containers Zeiger auf Objekte von möglicherweise heterogenem Typ sind. In C++ müssen die verschiedenen Typen von einer Basisklasse abgeleitet und die Zeiger vom Typ »Zeiger auf Basisklasse« sein.

Ein Mittel, verschiedene Algorithmen mit verschiedenen Containern zusammenarbeiten zu lassen, besteht darin, dass die *Namen*, die zur Compilierzeit ausgewertet werden, für gleichartige Operationen gleich gewählt sind. Zum Beispiel gibt die Methode `size()` die Anzahl der Elemente eines Containers zurück, sei er nun vom Typ `vector`, `list` oder `map`. Ein anderes Beispiel sind die Methoden `begin()` und `end()`, mit denen die Position des ersten und die Position *nach* dem letzten Element ermittelt werden. Die letzte Position ist auch in einem C++-Array stets definiert, wenn auch nicht dereferenzierbar. Ein leerer Container wird durch Gleichheit von `begin()` und `end()` gekennzeichnet.

Iteratoren

Iteratoren arbeiten wie Zeiger. Je nach Anwendungsfall können sie selbst gewöhnliche Zeiger oder Objekte mit zeigerähnlichen Eigenschaften sein. Der Zugriff auf ein Container-Element ist über einen Iterator möglich. Iteratoren können sich von einem Element zum nächsten bewegen, wobei die Art der Bewegung nach außen hin verborgen ist (Kontrollabstraktion). Beispielsweise bedeutet in einem Vektor die Operation `++` das einfache Weiterschalten zur nächsten Position im Speicher, während dieselbe Operation in einem binären Suchbaum mit dem Entlangwandern im Baum verbunden ist.

Hinweis

Die folgende Konvention der C++-Standardbibliothek muss beachtet werden: Wenn zwei Iteratoren einen *Bereich* definieren, zeigt der erste Iterator auf die Position des ersten Elements und der zweite auf die Position *nach* dem letzten Element.

Algorithmen

Die Template-Algorithmen arbeiten mit Iteratoren, die auf Container zugreifen. Da nicht nur benutzerdefinierte Datentypen unterstützt werden, sondern auch die in C++ ohnehin vorhandenen Datentypen wie `int`, `char` usw., wurden die Algorithmen so entworfen, dass sie ebenso gut mit normalen Zeigern arbeiten (siehe Beispiel im folgenden Abschnitt).

Zusammenwirken

Container stellen Iteratoren zur Verfügung, Algorithmen benutzen sie:

Container $\Longleftrightarrow$ Iteratoren $\Longleftrightarrow$ Algorithmen

Dadurch gibt es eine Entkopplung, die ein außergewöhnlich klares Design erlaubt. Im Folgenden soll ein Programm in verschiedenen Varianten zeigen, dass Algorithmen mit C-Arrays genauso gut funktionieren wie mit Template-Klassen der C++-Standardbibliothek.

In diesem Beispiel soll ein per Dialog einzugebender `int`-Wert in einem Array gefunden werden, wozu eine Funktion `find()` benutzt wird, die auch als Algorithmus der C++-

Standardbibliothek vorliegt. Parallel wird `find()` auf verschiedene Arten formuliert, um die Abläufe sichtbar zu machen. Um sich schrittweise der angestrebten Formulierung zu nähern, wird zunächst eine Variante *ohne* Benutzung der C++-Standardbibliothek dargestellt. Der Container ist ein schlichtes C-Array. Um auszudrücken, dass ein Zeiger als Iterator wirkt, wird der Typname `IteratorType` mit `typedef` eingeführt.

Listing 11.1: Variante noch ohne Nutzung der STL

```
// cppbuch/k11/einf/ohneSTL.cpp
#include<iostream>
using namespace std;
// neuer Typname IteratorType für 'Zeiger auf const int' :
typedef const int* IteratorType; // hier noch Zeiger

// Prototyp des Algorithmus
IteratorType find(IteratorType begin, IteratorType end, const int& Value);

int main() {
   const int ANZAHL = 100;
   int einContainer[ANZAHL];       // Container definieren
   IteratorType begin = einContainer; // Zeiger auf den Anfang
   // Position NACH dem letzten Element
   IteratorType end = einContainer + ANZAHL;
   // Container mit beliebigen Werten füllen (hier: gerade Zahlen)
   for(int i = 0; i < ANZAHL; ++i) {
      einContainer[i] = 2*i;
   }
   int zahl = 0;
   while(zahl != -1) {
      cout << " gesuchte Zahl eingeben (-1 = Ende):";
      cin >> zahl;
      if(zahl != -1) {         // weitermachen?
        IteratorType position = find(begin, end, zahl);
        if (position != end) {
           cout << "gefunden an Position "
                << (position - begin) << endl;
        }
        else {
           cout << zahl << " nicht gefunden!" << endl;
        }
      }
   }
}

// Implementation
IteratorType find(IteratorType begin, IteratorType end, const int& Value) {
   while(begin != end      // Zeigervergleich
         && *begin != Value) // Dereferenzierung und Objektvergleich
        ++begin;           // nächste Stelle
   return begin;
}
```

Im Fall eines leeren Containers haben beide Iteratoren denselben Wert. Man sieht, dass der Algorithmus `find()` selbst nichts über den Container wissen muss. Er benutzt nur Zeiger (Iteratoren), die einige wenige Fähigkeiten benötigen:

- Der Operator `++` dient zum Weiterschalten auf die nächste Position.
- Der Operator `*` dient zur Dereferenzierung. Angewendet auf einen Zeiger (Iterator) gibt er eine Referenz auf das dahinterstehende Objekt zurück.
- Die Zeiger müssen mit dem Operator `!=` vergleichbar sein.

Die Objekte im Container werden hier mit dem Operator `!=` verglichen. Im nächsten Schritt streiche ich die Implementierung der Funktion `find()` und ersetze den Prototyp durch ein Template:

Listing 11.2: Variante 2: Algorithmus als Template

```
// Auszug aus cppbuch/k11/einf/ohneSTLmitTemplate.cpp
template<typename Iterator, typename T>
Iteratortype find(Iterator begin, Iterator end, const T& Value) {
   while(begin != end      // Zeigervergleich
        && *begin != Value) // Dereferenzierung und Objektvergleich
       ++begin;            // nächste Position
   return begin;
}
```

Der Rest des Programms bleibt *unverändert*. Der Platzhalter `IteratorType` für den Datentyp des Iterators kann jeden beliebigen Namen haben. Im dritten Schritt benutze ich einen Container der STL. Die Iteratoren `begin` und `end` werden im `main()`-Programm durch die Methoden der Klasse `vector<T>` ersetzt, die einen entsprechenden Iterator zurückgeben.

Listing 11.3: Variante 3: mit Template-Container vector

```
// cppbuch/k11/einf/mitVector.cpp
#include<iostream>
#include<vector>
using namespace std;

// Prototyp des Algorithmus ersetzt durch Template
template<typename Iterator, typename T>
Iterator find(Iterator begin, Iterator end, const T& Value) {
  while(begin != end      // Iterator-Vergleich
       && *begin != Value) // Dereferenzierung und Objektvergleich
    ++begin;              // nächste Position
  return begin;
}

// neuer Typname zur Verbesserung der Lesbarkeit; vector<int>::const_iterator ist vordefiniert.
typedef vector<int>::const_iterator IteratorType;

int main() {
  const int ANZAHL = 100;
  vector<int> einContainer(ANZAHL); // Container definieren
  // Container mit beliebigen Werten füllen (hier: gerade Zahlen)
  for(int i = 0; i < ANZAHL; ++i) {
    einContainer[i] = 2*i;
```

```
  }
  int zahl = 0;
  while(zahl != -1) {
    cout << " gesuchte Zahl eingeben (-1 = Ende):";
    cin >> zahl;
    if(zahl != -1) {          // weitermachen?
      // globales find() von oben benutzen
      IteratorType position =
        ::find(einContainer.begin(), // Container-Methoden
               einContainer.end(), zahl);
      if (position != einContainer.end())
        cout << "gefunden an Position "
             << (position - einContainer.begin()) << endl;
    }
    else {
      cout << zahl << " nicht gefunden!" << endl;
    }
  }
}
```

Man sieht, wie der Standard-Container mit unserem Algorithmus zusammenarbeitet und wie Arithmetik mit Iteratoren möglich ist (Differenzbildung). Im letzten Schritt verwende ich den in der C++-Standardbibliothek vorhandenen `find()`-Algorithmus und ersetze das gesamte Template durch eine weitere `#include`-Anweisung:

Listing 11.4: Variante 4: Algorithmus `find()` der C++-Standardbibliothek

```
// Auszug aus cppbuch/k11/einf/mitSTL.cpp
#include<algorithm>
// ... Rest wie Variante 3, aber ohne find()-Template. Der Aufruf ::find() wird
// durch find() ersetzt (d.h. Namespace std).
```

Darüber hinaus ist es nicht erforderlich, mit `typedef` einen Iteratortyp zu definieren, weil jeder Container der STL einen entsprechenden Typ liefert. Anstatt `IteratorType position` kann im obigen Programm `vector<int>::iterator position` geschrieben werden – oder noch einfacher: `auto position`.

Interessant ist, dass der Algorithmus mit *jeder* Klasse von Iteratoren zusammenarbeiten kann, die die Operationen `!=` zum Vergleich, `*` zur Dereferenzierung und `++` zur Weiterschaltung auf das nächste Element zur Verfügung stellt. Dies ist ein Grund für die Mächtigkeit des Konzepts und für die Tatsache, dass jeder Algorithmus nur in *einer* Form vorliegen muss, womit Verwaltungsprobleme minimiert und Inkonsistenzen ausgeschlossen werden. Durch Verwendung der vorhandenen Algorithmen und Container werden Programme nicht nur kürzer, sondern auch zuverlässiger, weil Programmierfehler vermieden werden. Die Produktivität der Softwareentwicklung steigt damit.

11.2 Iteratoren im Detail

Die sequenzielle Bearbeitung einer Datenstruktur, die keine Liste sein muss, durch eine Methode, die nach außen hin *die innere Struktur der Daten verbirgt*, heißt *Kontrollabstraktion*. Das »Durchwandern« der Datenstruktur in einer bestimmten Reihenfolge kann sinnvoll sein, um auf jedes Element eine bestimmte Operation anzuwenden, zum Beispiel Ausdrucken des Inhalts. Der Benutzer soll auf die Elemente der Reihe nach zugreifen können, ohne Kenntnis über die verborgene Implementierung haben zu müssen. Ein Prinzip zur Kontrollabstraktion dieser Art wird *Iterator* genannt.

Ein Iterator ist ein Konzept, das auf eine Menge von Klassen und Typen zutrifft, die bestimmten Anforderungen entsprechen. Ein Iterator kann auf verschiedene Weise mit Grunddatentypen oder Klassenobjekten realisiert werden. Weil ein Iterator dazu dient, auf Elemente eines Containers zuzugreifen, ist er eine Art verallgemeinerter Zeiger. Mehr zu den Iteratoren der C++-Standardbibliothek lesen Sie ab Seite 805. Wesentlich für alle Iteratoren sind die bereits genannten Fähigkeiten des Weiterschaltens (`++`), der Dereferenzierung (`*`) und der Vergleichsmöglichkeit (`!=` bzw. `==`). Falls der Iterator nicht ein gewöhnlicher Zeiger, sondern ein Objekt einer Iterator-Klasse ist, werden diese Eigenschaften durch die entsprechenden Operatorfunktionen realisiert:

Listing 11.5: Schema eines einfachen Iterators

```
template<typename T>
class Iteratortyp {
   public:
      // Konstruktoren, Destruktor weggelassen
      bool operator==(const Iteratortyp<T>&) const;
      bool operator!=(const Iteratortyp<T>&) const;
      Iteratortyp<T>& operator++();     // präfix
      Iteratortyp<T> operator++(int);   // postfix
      T& operator*() const;
      T* operator->() const;
   private:
      // Verbindung zum Container ...
};
```

Der Operator `->` erlaubt es, einen Iterator wie einen Zeiger zu verwenden. Man kann sich bei einem Vector-Container natürlich vorstellen, dass der Iterator auch eine Methode `operator--()` haben sollte. Die wesentlichen Eigenschaften eines Iterators sind also:

1. Ein Iterator kann wie ein Zeiger benutzt werden.
2. Wie das Vorangehen mit dem `++`- bzw. `--`-Operator intern funktioniert, ist dem Benutzer verborgen. So wird ein Iterator `I` nur mit dem Befehl `++I` um eine Position weitergeschaltet. Die möglicherweise damit verbundene komplexe Operation, zum Beispiel den nächsten Eintrag in einem binären Suchbaum zu finden, ist für den Benutzer nicht sichtbar.

Zustände

Iteratoren erlauben es, mit verschiedenen Containern auf gleichartige Weise zu arbeiten. Ein Iterator kann verschiedene Zustände haben.

- Ein Iterator kann erzeugt werden, auch ohne dass er mit einem Container verbunden ist. Die Verbindung zu einem Container wird dann erst nachträglich hergestellt. Ein solcher Iterator ist nicht dereferenzierbar. Ein vergleichbarer C++-Zeiger könnte zum Beispiel den Wert 0 haben.
- Ein Iterator kann während der Erzeugung oder danach mit einem Container verbunden werden. Typischerweise – aber nicht zwingend – zeigt er nach der Initialisierung auf den Anfang des Containers. Die Methode `begin()` eines Containers liefert die Anfangsposition. Wenn der Container nicht leer ist, ist der Iterator in diesem Fall dereferenzierbar. Man kann also über ihn auf ein Element des Containers zugreifen. Der Iterator ist mit Ausnahme der `end()`-Position (siehe nächster Punkt) für alle Werte dereferenzierbar, die mit der Operation `++` erreicht werden können.
- In C++ ist der Wert eines Zeigers, der auf die Position direkt *nach* dem letzten Element eines C-Arrays zeigt, stets definiert. In Analogie dazu gibt die Methode `end()` eines Containers einen Iterator mit eben dieser Bedeutung zurück, auch wenn der Container kein Array, sondern zum Beispiel eine Liste ist. Damit können Iteratorobjekte und Zeiger auf C++-Grunddatentypen gleichartig behandelt werden. Der Vergleich eines laufenden Iterators mit diesem Nach-dem-Ende-Wert signalisiert, ob das Ende eines Containers erreicht wurde. Ein Iterator, der auf die Position nach dem Ende eines Containers verweist, ist natürlich nicht dereferenzierbar.

Auf die verschiedenen sinnvollen, möglichen Fähigkeiten eines Iterators und weitere Eigenschaften wird in Kapitel 29 eingegangen. Im nächsten Abschnitt wird ein Beispiel mit einem Iterator, der mehr können muss als ein Zeiger auf ein `int`-Array, gezeigt

11.3 Beispiel verkettete Liste

Wie funktioniert das Zusammenwirken von Containern, Iteratoren und Algorithmen genau? Um dies im Einzelnen zu zeigen, verwende ich das `main`-Programm aus Abschnitt 11.1 in leicht variierter Form. Damit ein Iterator dieser Klasse nicht einfach einem Zeiger gleichgesetzt werden kann, muss die Komplexität des Beispiels geringfügig erhöht werden: Eine einfach verkettete Liste tritt an die Stelle des Vektors. Die Klasse werde `Liste` genannt. Eine einfach verkettete Liste ist wohl die bekannteste dynamische Datenstruktur. Sie besteht aus Elementen, die miteinander über Zeiger verbunden sind (siehe Abbildung 11.1).

Ein Element einer einfach verketteten Liste besteht aus Daten, zum Beispiel einer Adresse, und einem Verweis auf das nächste Element. Der Nachteil der einfach verketteten Liste besteht darin, dass man sie nur in Vorwärtsrichtung bearbeiten kann, weil die Information über das Vorgängerelement in den Elementen der Liste nicht vorhanden ist. Es gibt verschiedene Möglichkeiten, auf eine Liste zuzugreifen:

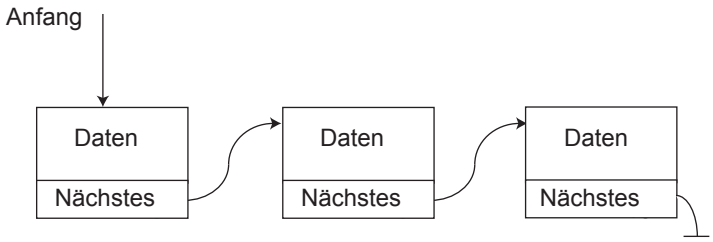

Abbildung 11.1: Einfach verkettete Liste

a) Einfügen eines Elements am Anfang
b) Einfügen eines Elements am Ende
c) Einfügen eines Elements dazwischen
d) Lesen und Entfernen eines Elements am Anfang
e) Lesen und Entfernen eines Elements am Ende

und andere mehr. Eine Liste, bei der nur die Operationen a) und e) zugelassen sind, heißt *Warteschlange* (englisch *queue*) und arbeitet nach dem *fifo*-Prinzip (fifo = first in, first out). Falls nur die Operationen a) und d) (beziehungsweise b) und e)) erlaubt sind, heißt die Liste *Stapel* oder Kellerspeicher (englisch *stack*). Auf einen Stapel kann man nur von oben etwas legen oder wegnehmen, er arbeitet nach dem *lifo*-Prinzip (last in, first out). In eine (allgemeine) Liste kann auch mittendrin eingefügt oder entnommen werden.

Im folgenden Beispiel beschränke ich mich auf a). Es gibt keinen Indexoperator und somit keinen wahlfreien Zugriff auf die Elemente. Ferner wird keinerlei Rücksicht auf Laufzeitverhalten genommen, um die Klasse so einfach wie möglich zu gestalten. Der vorhandene Algorithmus `std::find()` wird verwendet, um zu zeigen, dass sich die selbstgeschriebene Klasse wirklich genau wie eine Klasse der C++-Standardbibliothek verhält, wenigstens bezüglich des `main`-Programms unten.

Hinweis

Die Klasse für eine einfache Liste ist nicht vollständig und daher nicht sicher benutzbar! Sie stellt nur das zur Verfügung, was im Beispiel benötigt wird. Konsequenz: Nachdem Ihnen die Wirkungsweise klar ist, benutzen Sie für weitere Zwecke bitte die Klasse `list` der C++-Standardbibliothek (siehe Seite 772).

Die Liste besteht aus Elementen, deren Typ innerhalb der Klasse als geschachtelte `public`-Klasse (`struct`) definiert ist. In einem `struct`-Objekt ist der Zugriff auf interne Daten möglich. Dies stellt hier aber kein Problem dar, weil die Klasse innerhalb der `private`-Sektion der Klasse `Liste` definiert ist. Jedes Listenelement trägt die Daten (zum Beispiel eine Zahl) mit sich sowie einen Zeiger auf das nächste Listenelement. Die Klasse `Liste` stellt einen öffentlichen Iteratortyp `iterator` zur Verfügung. Ein Iterator-Objekt verweist auf die aktuelle Position in der Liste (Attribut `aktuellesElement` der Klasse `Liste::iterator`) und ist geeignet, darüber auf die in der Liste abgelegten Daten zuzugreifen, wie das Beispiel unten zeigt. Die Iteratormethoden erfüllen die auf Seite 401 beschriebenen Anforderungen.

Listing 11.6: Klasse für eine Liste (unvollständig)

```
// cppbuch/k11/liste/liste.t
// Unvollständig! Nur für das Beispiel notwendige Funktionen sind implementiert!
#ifndef LISTE_T
#define LISTE_T
#include<cstddef> // ptrdiff_t

template<typename T>
class Liste {
 public:
   Liste() : anfang(0), anzahl(0) { }
   // Kopierkonstruktor, Destruktor und Zuweisungsoperator weggelassen! (siehe Aufgaben)
   // push_front() erzeugt ein neues Listenelement und fügt es am Anfang der Liste ein:
   void push_front(const T& wert) {
      anfang = new ListElement(wert, anfang);
      ++anzahl;
   }
 private:
    struct ListElement {
       T daten;
       ListElement *naechstes;
       ListElement(const T& wert, ListElement* p)
            : daten(wert), naechstes(p) {}
    };
    ListElement *anfang;
    size_t anzahl;
 public:
    class iterator {
      public:
         // von find() geforderte Schnittstelle:
         typedef std::forward_iterator_tag iterator_category;
         typedef T value_type;
         typedef T* pointer;
         typedef T& reference;
         typedef ptrdiff_t difference_type;
         iterator(ListElement* init = 0) // Konstruktor
            : aktuellesElement(init){}
            T& operator*() {          // Dereferenzierung
               return aktuellesElement->daten;
            }
            const T& operator*() const { // Dereferenzierung
               return aktuellesElement->daten;
            }
            iterator& operator++() {  // Präfix
               if(aktuellesElement) // noch nicht am Ende angelangt?
                  aktuellesElement = aktuellesElement->naechstes;
               return *this;
            }
            iterator operator++(int) { // Postfix
               iterator temp = *this;
               ++*this;
               return temp;
```

```
        }
        bool operator==(const iterator& x) const {
          return aktuellesElement == x.aktuellesElement;
        }
        bool operator!=(const iterator& x) const {
          return aktuellesElement != x.aktuellesElement;
        }
    private:
        ListElement* aktuellesElement;
    }; // iterator
  // Einige Methoden der Klasse Liste benutzen die Klasse iterator:
  iterator begin() const { return iterator(anfang);}
  iterator end() const { return iterator();}
};
#endif
```

Das zugehörige `main`-Programm verwendet die Listenklasse. Eine gefundene Zahl wird nicht mit der Variablen `zahl` angezeigt, was auch möglich wäre, sondern mit dem dereferenzierten Iterator.

Listing 11.7: Anwendung der Liste

```
// cppbuch/k11/liste/main.cpp
#include<iostream>
#include<algorithm>
#include"liste.t"
using namespace std;

int main() {
  const int ANZAHL = 100;
  Liste<int> einContainer; // Container definieren

  // Container mit beliebigen Werten füllen (hier: gerade Zahlen)
  for(int i = 0; i < ANZAHL; ++i) {
    einContainer.push_front(2*i);
  }

  int zahl = 0;
  while(zahl != -1) {
    cout << " gesuchte Zahl eingeben (-1 = Ende):";
    cin >> zahl;
    if(zahl != -1) {          // weitermachen?
      // std::find() benutzen
      auto position =              // siehe Text unten
        find(einContainer.begin(), // Container-Methoden
             einContainer.end(), zahl);
      if (position == einContainer.end()) {
        cout << " nicht gefunden!";
      }
      else {
        cout << *position // Dereferenzierung des Iterators
             << " gefunden!" << endl;
      }
    }
```

```
  }
  cout << "Liste ausgeben:" << endl;
  for(auto iter = einContainer.begin();     // siehe Text unten
      iter != einContainer.end(); ++iter) {
     cout << *iter << endl;
  }
}
```

In diesem Programm kommt das Schlüsselwort `auto` von Seite 90 zur Geltung, um nicht `Liste<int>::iterator` schreiben zu müssen. Statt zum Beispiel

```
for(Liste<int>::iterator iter = einContainer.begin();
                         iter != einContainer.end(); ++iter) {
     cout << *iter << endl;
```

heißt es einfach

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

Was der Klasse `Liste` noch fehlt, um sie zu einer vollwertigen Klasse im Sinne der C++-Standardbibliothek zu machen, können Sie Abschnitt 28.2.3 entnehmen.

Diese Einführung zeigt die nur Funktionsweise der STL. Mehr Informationen zu den Containern, Iteratoren und Algorithmen der C++-Standardbibliothek finden Sie in weiteren Kapiteln:

- Container: Kapitel 28
- Iteratoren: Kapitel 29
- Algorithmen: Kapitel 24 und 30

Übung

11.1 Ergänzen Sie die Klasse `Liste` wie folgt und versuchen Sie, Lösungen der Teilaufgaben wiederzuverwenden.

- Kopierkonstruktor und Zuweisungsoperator. Der letztere kann vorteilhaft den ersteren einsetzen, indem erst eine temporäre Kopie der Liste erzeugt wird und dann die Verwaltungsinformationen vertauscht werden.
- Destruktor.
- Methode `iterator erase(iterator p)`, die das Element, auf das der Iterator `p` zeigt, aus der Liste entfernt. Der zurückgegebene Iterator soll auf das nach `p` folgende Element zeigen, sofern es existiert. Andernfalls soll `end()` zurückgegeben werden. Ergänzen Sie vorher die Klasse `iterator` um `friend class Liste;`, damit `erase()` auf das zu löschende Element zugreifen darf.
- Methode `void pop_front()`, die das erste Element löscht.
- Methode `void clear()`, die die ganze Liste löscht.
- Methode `bool empty()`, die anzeigt, ob die Liste leer ist.
- Methode `size_t size()`, die die Anzahl der Elemente zurückgibt.