Programmieren in C++ (7): Die Standard Template Library STL

Silke Seehusen, Fachhochschule Lübeck

Der Artikel soll ermutigen, die Bibliothek zu nutzen. Er gibt einen Überblick über den Teil der C++-Bibliothek des geplanten ANSI-Standards, der unter dem Namen Standard Template Library (STL) eingeführt wurde. Die Einheitlichkeit und zugleich Mächtigkeit der Bibliothek wird an einem Beispiel demonstriert.

Die Standard-Template-Library (STL) und die Standardisierung von C++

Der Entwurf der Standard-C++-Bibliothek ist relativ spät um einen sehr bedeutenden Teil erweitert worden, der zunächst die Standard-Template-Library von C++, kurz STL, genannt wurde, weil er unter anderem sehr viele Templates zur Verfügung stellt. Der Vorschlag, STL in die Standard-C++-Bibliothek aufzunehmen, kam erst 1994 [1] und wurde relativ schnell in den vorgeschlagenen C++-Standard aufgenommen, wie er in [2] beschrieben ist. Heute ist die STL integraler Bestandteil der Standard-C++-Bibliothek und nicht mehr direkt als eigenständiger Teil zu erkennen.

Weil die STL sehr spät integriert wurde, findet sich in ''älteren'' Büchern wie [3], [4] oder sogar [5] keine Beschreibung der Komponenten, die durch STL zur Verfügung gestellt werden. Deshalb soll in diesem Artikel ein Überblick über wesentliche Komponenten gegeben werden, die ursprünglich aus STL stammen. Da es jetzt keine exakte Grenze mehr zu STL gibt, kann im Einzelfall diese Zuordnung nicht mehr getroffen werden. Wichtig ist nur, daß die Komponenten, insbesondere die zur Verfügung gestellten Templates, bekannt sind und benutzt werden, denn es sind mächtige Werkzeuge der Programmierung.

Motivation

Als ich die Container-Klassen, die aus STL stammen, in einer Lehrveranstaltung präsentierte, war der Kommentar einer Studentin: ''Da hat die Programmiererin ja fast gar nichts mehr zu tun''. Gemeint hat sie wohl, daß viel Routinearbeit wegfällt. Unterschätzt hat sie zum einen die Analyse einer größeren Anwendung und zum anderen die Einarbeitung in die Benutzung einer Bibliothek von der Komplexität der Standard-C++-Bibliothek. Das war eine weitere Motivation für diesen Artikel.

Die STL erhöht die Einsetzbarkeit von C++ enorm, da wesentliche, immer wiederkehrende Templates dort festgelegt sind. Eine Vielfalt von Implementierungen von z.B. Containern wird dadurch unnötig.

Es bleibt nach wir vor jedem überlassen, einige Elemente der Bibliothek noch einmal zu implementieren.

Struktur

Die wesentlichen Komponenten der STL sind:

- **Algorithmen**, wie z.B. Sortieren,
- **Container**, wie z.B. Listen oder Mengen,
- **Iteratoren**, mit deren Hilfe u.a. Container elegant traversiert werden können,
- **Funktionsobjekte**, die eine Funktion einkapseln, die von anderen Komponenten benutzt werden kann, und
- **Adaptoren**, durch die eine andere Schnittstelle von einer Komponente gebildet werden kann.

Container

Ein **Container** ist ein Objekt, das der Speicherung anderer Objekte dient, z.B. eine Liste, ein Feld oder eine Warteschlange. Mit einer Container-Klasse wird der Typ eines solchen Objekts definiert. Die Objekte, die ein Container speichert, nennen wir hier **Elemente**, auch wenn ein Element mehrfach in einem Container auftreten kann. Den Typ der Elemente nennen wir Elementtyp.

Da Container-Klassen für einen beliebigen Elementtyp definiert werden sollten, sind in der Standard-C++-Bibliothek Templates für Container-Klassen definiert. Es wird unterschieden zwischen

- **Sequenzen**, in denen jedes Element eine Position besitzt, wie z.B. Listen,
- **Adaptoren für Sequenzen**, die einer Sequenz eine andere Schnittstelle verleihen, wie z.B. ein Stapel,
- **assoziative Container**, die jeweils eine Menge von Elementen verwalten, auf die über ihren Inhalt (Wert) zugegriffen wird (Beispiel *set* und *map*).

	Template für Container-Klasse	*Definitions-datei*
Sequenzen	`vector <T>`	`<vector>`
	`list <T>`	`<list>`
	`deque <T>`	`<deque>`
Adaptoren von Sequenzen	`stack <Container>`	`<stack>`
	`queue <Container>`	`<queue> <stack>`
	`priority_queue <Container,Compare>`	`<queue> <stack>`
assoziative Container	`set <T,Compare>`	`<set>`
	`multiset <T,Compare>`	`<set> <multiset.h>`
	`map <Key,T,Compare>`	`<map>`
	`multimap <Key,T,Compare>`	`<map> <multimap.h>`

Abbildung 1: Templates von Container-Klassen

In Abbildung 1 ist eine Übersicht der Templates angegeben, wobei

- *T* jeweils der Elementtyp,
- *Compare* eine Vergleicherklasse,
- *Key* der Typ der Schlüssel der abgespeicherten Elemente und
- *Container* selbst wieder eine Container-Klasse ist.

Jeder der Template-Klassen, die keine Adaptoren sind, kann als weiterer Template-Parameter ein Allokator übergeben werden, der angibt, wie Speicher für den Container reserviert werden soll. Voreingestellt ist die normale Speicherreservierung, die auch beim Standardoperator *new* benutzt wird.

Eine Stärke der STL ist, daß, soweit möglich, alle Container eine gleiche Schnittstelle besitzen. Diese Schnittstelle wird (leider) nicht in einer abstrakten Klasse definiert, sondern durch eine Tabelle, in der die gemeinsamen Eigenschaften aller Container festgelegt werden. Die wichtigsten Eigenschaften sind in Abbildung 2 zusammengefaßt.

Container X		

Konstruktoren	`X u;`
	`X();`
	`X(a);`
	`X u = a;`
		u wird leerer Container.
		Leerer Container wird erzeugt.
		Neuer Container mit X(a) == a wird erzeugt.
		X u; u = a;

Destruktor	`(&a)->~X();`	Inhalt des Containers a wird gelöscht. Danach ist der Container leer.
Zuweisung	`r = a`	Dem Container mit Referenz r wird der Container a zugewiesen.

Traversieren	`a.begin()`
	
	`a.end();`
		Liefert einen Iterator auf das "erste" Element des Containers a zurück.
		Liefert einen Iterator auf den (gedachten) Nachfolger des letzten Elementes des Containers a zurück.

Vergleiche	`a == b`
	
	`a != b`
	`a < b`
	
	`a > b`
	`a <= b`
	`a >= b`
		Prüft, ob die Container a und b elementweise gleich sind.
		!(a == b)
		lexikographischer Vergleich der Elemente von a.begin() bis a.end() mit b.begin() bis b.end()
		b < a
		!(a > b)
		!(a < b)

Größe	`a.size()` `a.empty()`	Anzahl Elemente in a a.size() == 0

Typen	`X::value_type`
	`X::iterator`
	`X::const_iterator`
	
	`X::size_type`
		Elementtyp
		Typ eines Iterators, der auf ein Element zeigt
		Typ eines Iterators, der auf ein konstantes Element zeigt
		Typ nichtnegativer Werte von z.B. Iteratordifferenzen und von size()

Abbildung 2: Wichtige Eigenschaften eines Containers

Da es keinen solchen abstrakten Container gibt, folgt das erste Beispiel erst im nächsten Abschnitt. Weggelassen in Abbildung 2 sind einige im Container definierte Typen und selten verwendete Methoden. Für Abbildung 2 wie auch die weiteren Abbildungen, in denen wichtige Eigenschaften von Containern und Klassen zusammengefaßt werden, gilt, daß eine vollständige Beschreibung [2] zu entnehmen ist. Eine tabellarische Übersicht der definierten Templates ist auch in [6] zu finden.

Sequenzen

Sequenzen besitzen zusätzlich zu den Eigenschaften aus Abbildung 2 im wesentlichen Methoden zum Einfügen (*insert*) und Löschen (*erase*) von Elementen, siehe Abbildung 3. Positionen in Sequenzen sind Iteratoren. Eine genauere Beschreibung von Iteratoren findet sich weiter unten.

Sequenz X (zusätzlich zu Container)		
Einfügen	`a.insert(p,t)`	Fügt t vor der Position p in a ein. Liefert Iterator auf eingefügtes Element zurück.
Löschen	`a.erase(q)`	Löscht das Element an Position q.
bestimmte Elemente	`a.front()` `a.back()`	*a.begin(), erstes Element *(a.end() - 1), letztes Element
nur vector und deque	`a[n]`	n-tes Element

Abbildung 3: Wichtige Eigenschaften einer Sequenz

list

Die wohl berühmteste Sequenz ist die Liste. Mit den bisher definierten Eigenschaften können wir sehr einfach eine Liste von Widerständen verwalten, die zu einer Schaltung gehören. Ein Beispiel einer solchen Schaltung ist in Abbildung 4 dargestellt.

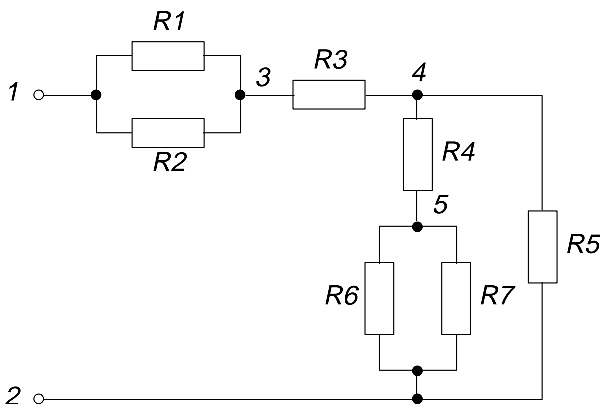
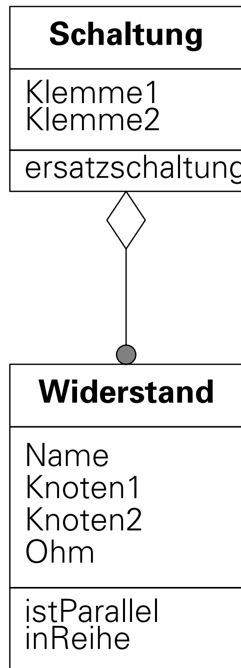


Abbildung 4: Schaltung

Ein Widerstand wird durch seinen Namen, zwei Knoten, an denen der Widerstand in der Schaltung angeklemmt ist, und seinen Widerstandswert beschrieben. Eine Schaltung hat zwei Klemmen und besteht aus einer Menge von Widerständen:



Eine solche Aggregation kann als Liste von Widerständen implementiert werden:

```
class Schaltung
{
 private:
  Klemme _klemme1;
  Klemme _klemme2;
  typedef list<Widerstand>
          List;
  List  _widerstaende;
 // ...
};
```

Die Liste wird automatisch bei Erzeugung eines Objekts der Klasse *Schaltung* als leere Liste erzeugt. Aufgebaut wird sie z.B. beim Einlesen der Widerstände:

```
istream &
Schaltung::read (istream & s)
{
 s >> _klemme1 >> _klemme2;
 int  anzahl; s >> anzahl;
 Widerstand  w;
 for ( int i=0;
       i<anzahl && s; i++ )
 {
   s >> w;
   _widerstaende.insert
    (_widerstaende.end(), w);
 }
 return s;
}
```

In einer Liste wird immer vor bestimmten Positionen eingefügt. Typische Positionen einer Liste *a* sind die Position des ersten Elementes, die von *a.begin()* zurückgegeben wird, und die Position hinter dem letzten Element der Liste, die von *a.end()* zurückgegeben wird. Positionen sind Iteratoren, auf denen auch der Nachfolgeoperator *++* und der Durchgriff auf das Element, auf das ein Iterator *i* zeigt, mit **i* definiert sind.

So sieht die Suche eines Widerstandspaares, das durch einen Ersatzwiderstand

ersetzt werden kann, wie in Abbildung 5 aus.

```
bool Schaltung::regelAngwendet()
{
  for( List::iterator ir1=_widerstaende.begin();
       ir1 != _widerstaende.end(); ir1++ )
  {
    List::iterator ir2=ir1;
    for(++ir2; ir2 != _widerstaende.end(); ir2++)
      if ( ersatzParallel(*ir1, *ir2) ||
           ersatzInReihe (*ir1, *ir2)   )
        return true;
  }
  return false;
}
```

Abbildung 5: Anwendung einer Ersetzungsregel

Ziel ist es, durch wiederholte Anwendung von Regeln die Schaltung in Abbildung 4 durch einen Ersatzwiderstand wie in Abbildung 6 zu ersetzen. Zwei parallel geschaltete und zwei in Reihe geschaltete Widerstände, zwischen denen kein anderer Widerstand abgeht, können nach den bekannten Regeln ersetzt werden.

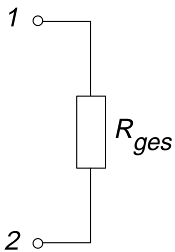


Abbildung 6: Ersatzwiderstand

Sind zwei Widerstände gefunden, die ersetzt werden können, werden sie gelöscht. Im folgenden wird als Beispiel der Widerstand *r1* gelöscht:

```
List::iterator i =
 find(_widerstaende.begin(),
      _widerstaende.end(),
      r1);
if( i!=_widerstaende.end() )
  _widerstaende.erase(i);
```

Hier wird zuerst der Widerstand *r1* im halboffenen Intervall *[b, e)* von *_widerstaende* gesucht, wobei *b* die Position des ersten und *e* die Position nach dem letzten abzuprüfenden Element ist. *find* liefert einen Iterator auf das gesuchte Element zurück oder *end()*, wenn kein Element gefunden wurde. Beim Löschen eines Elements in einer Sequenz muß seine Position angegeben werden.

find ist übrigens ein Funktions-Template, das auf Eingabeiteratoren (input_iterator, siehe unten) definiert und in der Definitionsdatei *<algorithm>* neben vielen anderen nützlichen Algorithmen wie *find_if*, *for_each*, *copy*, *transform* und *sort* zu finden ist.

Um alle Widerstände einer Schaltung auszugeben, bedarf es unter Verwendung des Funktions-Templates *copy* und dem Iterator-Template *ostream_iterator* einer Anweisung:

```
copy(_widerstaende.begin(),
     _widerstaende.end(),
     ostream_iterator
       <Widerstand>(s,"\n"));
```

Für alle Widerstände *w* von *begin()* bis *end()* wird *s << w* aufgerufen, wobei *s* ein Ausgabestrom, z.B. *cout*, ist.

vector

Die Sequenz *vector* ist zunächst genauso zu benutzen wie eine Liste. Alle bisher vorgestellten Methoden tragen denselben Namen und haben dieselben Parameter, nur daß sie jetzt auf *vector* bezogen sind. Im obigen Beispiel muß nur in der Typdefinition von *List* statt *list* jetzt *vector* stehen, und alle Programme haben denselben Effekt wie vorher.

Auf Vektoren haben wir, wie in Abbildung 3 auch angegeben, eine Zugriffsmöglichkeit mehr. Wir können mit *a[n]* auf das n-te Element des Vektors *a* zugreifen, sowohl lesend als auch schreibend. Als Beispiel kann der Rumpf der Methode *regelAngwendet* aus Abbildung 5 jetzt folgendermaßen formuliert werden:

```
for(int i=0;
    i<_widerstaende.size();
    i++ )
  for(int j=i+1;
      j<_widerstaende.size();
      j++ )
  {
    if(ersatzParallel
        (_widerstaende[i],
         _widerstaende[j]) ||
       ersatzInReihe
        (_widerstaende[i],
         _widerstaende[j])  )
      return true;
  }
return false;
```

Aber aufgepaßt: Das Einfügen und Löschen funktioniert wie bei einer Liste. Das bedeutet, daß an jeder Position eingefügt werden kann. Auch der Vektor verändert dabei seine Größe (Anzahl Elemente). Indizes wie auch Iteratoren und Referenzen sind nur so lange gültig, wie keine Elemente eingefügt oder gelöscht werden.

deque

Wo bisher *vector* benutzt wurde, kann auch die Schlange *deque* verwendet werden. Der wesentliche Unterschied zwischen den einzelnen Sequenzarten liegt im Laufzeitaufwand (siehe Abbildung 7). Beim Vektor benötigt das Einfügen und Löschen am Ende des Vektor konstante Zeit, ansonsten ist der Aufwand linear abhängig von der Anzahl *n* der Elemente im Vektor. Der Vektor wie auch die Schlange erlauben einen wahlfreien Zugriff auf Elemente über einen Index in konstanter Zeit.

	vector	*deque*	*list*
Einfügen und Löschen am Ende	O(1)	O(1)	O(1)
Einfügen und Löschen am Anfang	O(n)	O(1)	O(1)
Einfügen und Löschen sonst	O(n)	O(n)	O(1)
wahlfreier Zugriff	O(1)	O(1)	O(n)

Abbildung 7: Laufzeitaufwand der Methoden von Sequenzen

Die Schlange garantiert auch konstante Zeit beim Einfügen und Löschen am Anfang und die Liste überall. Die Liste erlaubt jedoch keinen schnellen wahlfreien Zugriff.

Es bleibt zu erwähnen, daß es für *vector<bool>* eine spezielle Implementierung in der Bibliothek gibt, die deutlich effizienter ist, als wenn ‘‘nur’’ das Template benutzt würde.

Anforderungen an Elementtyp

Alle Sequenzen und auch die assoziativen Container, die weiter unten beschrieben werden, setzen einige Eigenschaften vom Elementtyp voraus. Es muß folgendes vom Elementtyp *T* sinnvoll definiert und auch öffentlich sein:

- *T()* Konstruktor ohne Argumente,
- *T(const T&)* Kopierkonstruktor,
- *˜T()* Destruktor,
- *T& operator = (const T &)*
  Zuweisung und
- *static void * operator new*
  (size_t).

Doch das ist für jeden in diesem Zusammenhang sinnvollen Typ selbstverständlich.

Adaptoren von Sequenzen

Ein Template, das einem Container (als Template-Parameter) eine in der Regel eingeschränkte Schnittstelle gibt, wird ein **Adaptor der Sequenz** genannt. In der Bibliothek gibt es die Adaptoren für einen Stapel, *stack*, für eine Warteschlange, *queue*, und für eine Warteschlange mit Priorität, *priority_queue*.

stack

Eines der berühmtesten Beispiele für abstrakte Datentypen ist der Stapel (stack). Auf einem Stapel sind die bekannten Operationen

- *void push(const value_type&)*,
- *void pop()*,
- *value_type& top()* und *const value_type& top()*,
- *bool empty()* und
- *size_type size()*

definiert.

Vielleicht etwas ungewöhnlich ist, daß der Template-Parameter von *stack* eine Sequenz sein muß. Als Beispiel wird ein Stapel von Widerständen definiert durch:

```
stack< list<Widerstand> >
                    stapel;
```

Damit wird **kein** Stapel von Listen von Widerständen definiert, sondern (nur) ein Stapel von Widerständen. Benutzt wird der Stapel wie erwartet:

```
Widerstand  w;
while ( cin >> w )
  stapel.push(w);
while ( !stapel.empty() )
{
  cout << stapel.top();
  stapel.pop();
}
```

Ein Stapel hat dann vom Zugriff her dasselbe Verhalten wie der Template-Parameter des Stapels.

queue und priority_queue

Für die Definition von Warteschlangen gilt das gleiche Prinzip wie für den Stapel. Die Methoden, die zur Verfügung gestellt werden, sind auch klassisch. Es werden die gleichen Methoden wie beim Stapel zur Verfügung gestellt, nur daß *push* am Ende anhängt und *pop* das erste Element löscht. Außerdem liefert *front()* das erste Element und *back()* das letzte Element der Warteschlange zurück.

Bei einer Warteschlange mit Priorität, *priority_queue*, muß neben der Sequenz als weiterer Template-Parameter eine Vergleicherklasse übergeben werden, nach der die Elemente in der Warteschlange sortiert werden. Als Sequenz ist nur ein Typ wie ein Vektor (vector) oder eine Schlange (deque) erlaubt, der die Methoden *front()*, *push_back* und *pop_back* zur Verfügung stellt, z.B.:

```
priority_queue
   < deque<Widerstand>,
     less<Widerstand>   >  q;
```

Der Zugriff auf das erste Element geschieht mit *top()* und nicht mit *front()*.

Anzumerken ist an dieser Stelle noch, daß sich die STL in der C++-Bibliothek von GNU noch nicht ganz an den Standard hält. So finden sich z.B. die Definitionen der Warteschlangen nicht in *<queue>* sondern in *<stack>*.

Mengen und Abbildungen

Assoziative Container dienen der Verwaltung einer Menge von Elementen, auf die über ihren Inhalt (Wert) zugegriffen wird. Neben dem Elementtyp muß einem Template eines assoziativen Containers auch die Information mitgegeben werden, wann zwei Elemente gleich sind (siehe Abbildung 1). Das geschieht, indem ein Vergleicher mitgegeben wird, der der Kleiner-Relation < entspricht. Sei der Vergleicher *kleiner*, dann gelten zwei Elemente *a* und *b* als gleich, wenn weder *kleiner()(a,b)* noch *kleiner()(b,a)* gilt. Der Vergleicher muß eine totale Ordnung auf den Elementen festlegen.

set

Das Template *set<T,Compare>* dient der Verwaltung einer Menge von Elementen, wobei die Elemente entsprechend der Definition der Gleichheit nur einmal in der Menge vorkommen.

Da Widerstände einer Schaltung eindeutig sind, können wir die Widerstände in einer Schaltung jetzt als Menge definieren:

```
typedef
  set< Widerstand,
       less<Widerstand> >
     Set;
  Set   _widerstaende;
```

Als Vergleicherklasse wird die Kleinerrelation auf Widerständen übergeben, die hier aus dem Template *less<T>* generiert wird, das die Relation auf die Anwendung der Operators < zurückführt. Der Operator muß dann für *T* existieren. Wir haben ihn auf Widerständen festgelegt als:

```
bool Widerstand::operator <
     ( const Widerstand & p )
     const
{ return (
   _name    < p._name ||
  (_name   == p._name &&
```

```
    _knoten1 < p._knoten1 ) ||
   (_name   == p._name &&
    _knoten1== p._knoten1 &&
    _knoten2 < p._knoten2 ) ||
   (_name   == p._name &&
    _knoten1== p._knoten1 &&
    _knoten2== p._knoten2 &&
    _ohm     < p._ohm ) );
}
```

Das Template *less<T>* ist in der Standard-C++-Bibliothek in *<functional>* definiert.

Menge X<T,Compare> (zusätzlich zu Container)		
Einfügen	`a.insert(t)`	Fügt t in a ein.
Löschen	`a.erase(t)`	Löscht alle Elemente in a, die gleich t sind.
	`a.erase(p)`	Löscht das Element in a, auf das der Iterator p zeigt.
Suchen	`a.find(t)`	Liefert einen Iterator auf ein Element aus a zurück, das gleich t ist.
Zählen	`a.count(t)`	Liefert die Anzahl Elemente in a zurück, die gleich t sind.

Abbildung 8: Wichtige Methoden von set und multiset

Die wichtigsten Methoden einer Menge sind in Abbildung 8 aufgeführt. Dabei ist *a* ein Mengenobjekt, *t* jeweils ein Objekt des Elementtyps und *p* ein Iterator. Auf Mengen sind keine Positionen wie auf Sequenzen definiert. Zum Traversieren gibt es nach wie vor Iteratoren, so daß die Methode *Schaltung::regelAngwendet()* genauso aussieht wie die Listenvariante in Abbildung 5, wenn wir dort *List* durch *Set* ersetzen.

Beim Einfügen und Löschen muß jetzt kein Iterator, sondern es kann gleich das entsprechende Element angegeben werden. Das Ersetzen zweier paralleler Widerstände sieht jetzt folgendermaßen aus:

```
bool Schaltung::
  ersatzParallel(
        const Widerstand r1,
        const Widerstand r2 )
{
 if ( r1.parallel(r2) )
 {
   Widerstand ersatz
   ( "("+r1.name()+"||"+
      r2.name()+")",
     r1.knoten1(),
     r1.knoten2(),
     (r1.ohm()*r2.ohm()) /
       (r1.ohm()+r2.ohm()) );
   _widerstaende.erase(r1);
   _widerstaende.erase(r2);
   _widerstaende.insert
                (ersatz);
   return true;
 }
 else return false;
}
```

multiset

Ein Multiset ist eine Menge, in der Elemente mehrfach vorkommen können. Bei jedem Einfügen wird ein neues Element eingefügt, und beim Löschen mit *a.erase(t)* werden alle Elemente, die gleich *t* sind, gelöscht. Mit *a.erase(p)* wird nur das Element, auf das *p* zeigt, gelöscht.

Auch hier ist anzumerken, daß die Definition von *multiset* in der libg++ in der Definitionsdatei *<multiset.h>* zu finden ist. Analog ist *multimap* (siehe unten) in *<multimap.h>* zu finden.

map und multimap

Inhaltsadressierte Container im engeren Sinne sind Abbildungen (maps). Ein Schlüssel wird dabei auf ein Datum abgebildet. Der eigentliche Elementtyp ist hier jeweils ein Paar (Tupel), bestehend aus Schlüssel und Datum. Entsprechend muß beim Einfügen ein solches Paar als Parameter übergeben werden (siehe Abbildung 9).

Abbildung X<Key,T,Compare> (zusätzlich zu Container)		
Einfügen	`a.insert(pair(k,t))`	Fügt pair(k,t) in a ein.
Löschen	`a.erase(k)`	Löscht alle Elemente in a, deren Schlüssel gleich k sind.
	`a.erase(p)`	Löscht das Element in a, auf das der Iterator p zeigt.
Suchen	`a.find(k)`	Liefert einen Iterator auf ein Element aus a zurück, dessen Schlüssel gleich k ist.
Zählen	`a.count(k)`	Liefert die Anzahl Elemente in a zurück, deren Schlüssel gleich k sind.

Abbildung 9: Wichtige Methoden von map und multimap

Ein solches Paar wird mit dem Konstruktor *pair(key,t)* erzeugt, der wiederum aus einem Template generiert wird, das in der Definitionsdatei *<utility>* zu finden ist. Auf die erste Komponente eines Paares *x* kann mit *x.first* und auf die zweite Komponente mit *x.second* zugegriffen werden.

Das Schaltungsbeispiel kann jetzt folgendermaßen umformuliert werden. Ein Widerstand sei eindeutig über seinen Namen identifiziert. Dann können die Widerstände einer Schaltung verwaltet werden mit:

```
typedef
  map< string,Widerstand,
       less<string> >
  Map;
Map     _widerstaende;
```

Das Einfügen und Löschen sieht jetzt etwas anders aus. In unserem Beispiel wird der Name eines Widerstandes einmal beim Widerstand selbst als Attribut und eine Kopie davon als Schlüssel in der Abbildung benutzt. Das Ersetzen zweiter paralleler Widerstände *r1* und *r2* durch den Widerstand *ersatz* wird damit formuliert als:

```
_widerstaende.
        erase(r1.name());
_widerstaende.
        erase(r2.name());
_widerstaende.insert(
        Map::value_type
              (ersatz.name(),
```

```
                ersatz) );
```

Dabei ist *Map::value_type* der Typ der abgespeicherten Paare (pair<const Key, T>).

Iteratoren

Bei allen Containern wurden Iteratoren verwendet, um einen Container zu traversieren oder eventuell auch, um auf ein bestimmtes Element zu verweisen. In der Standard-C++-Bibliothek gibt es mehrere Typen von Iteratoren. Die einzelnen Container stellen bidirektionale Iteratoren oder Iteratoren zum wahlfreien Zugriff zur Verfügung, siehe Abbildung 10.

```
vector<T>::iterator                  random_access_iterator
list  <T>::iterator                  bidirectional_iterator
deque <T>::iterator                  random_access_iterator

set     <T,Comp>::iterator           bidirectional_iterator
multiset<T,Comp>::iterator           bidirectional_iterator

map     <Key,T,Comp>::iterator       bidirectional_iterator
multimap<Key,T,Comp>::iterator       bidirectional_iterator
```

Abbildung 10: Typen der Iteratoren der Container

Die Eigenschaften der verschiedenen Iteratortypen und deren Vererbungshierarchie sind in Abbildung 11 dargestellt.

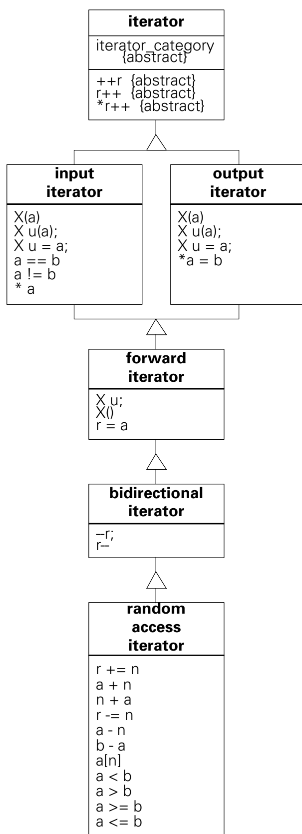


Abbildung 11: Vererbung der Eigenschaften von Iteratoren

Wenn ein Container einen Iterator mit wahlfreiem Zugriff (random_access_iterator) besitzt wie z.B. ein Vektor, dann können die Elemente des Containers einfach sortiert werden:

```
  sort( _widerstaende.begin(),
```

```
        _widerstaende.end()  );
```

Das Funktions-Template *sort<first,last>* sortiert den Container im Intervall *[first,last)* nach der Relation <. *first* und *last* müssen Iteratoren mit wahlfreiem Zugriff sein. Soll eine andere Vergleichsfunktion verwendet werden, so kann das Funktions-Template *sort<first,last,compare>* verwendet werden. Sollen die Widerstände absteigend sortiert werden:

```
  sort( _widerstaende.begin(),
        _widerstaende.end(),
        greater<Widerstand>());
```

Diese Funktions-Templates sind in *<algorithm>* definiert, wo noch weitere interessante Funktions-Templates stehen.

Die Funktions-Templates setzen unterschiedliche Typen von Iteratoren voraus, wobei die normale Typverträglichkeit von Basisklassen mit abgeleiteten Klassen gilt.

Zusammenfassung

Die Standard-C++-Bibliothek gewinnt durch die dort definierten Templates erheblich an Mächtigkeit.

Da viele Routinedefinitionen in der Bibliothek vorhanden sind, können diese in Programmen einheitlich benutzt werden. Die Programme werden dadurch verständlicher, da auf allgemein bekannte Lösungen zurückgegriffen werden kann. Insbesondere ist die Namensgebung vereinheitlicht. (Wie viele Namen für die Methoden insert und erase von Containern kennen Sie?)

Aber die Benutzung von Templates aus der Standard-C++-Bibliothek erfordert eine sorgfältige Programmierung. Einfache Programmierungsfehler erzeugen eine Unmenge von Fehlermeldungen des Übersetzers. Das Prinzip, erst zu kontrollieren und dann den Übersetzer zu bemühen, bekommt wieder seine Berechtigung.

Die Bibliotheken, die mit Übersetzern mitgeliefert werden, halten sich zur Zeit nicht immer ganz an den Standard.

Insgesamt kann ich jedoch nur empfehlen, die Templates der Standard-C++-Bibliothek wirklich zu nutzen.

Quellen

Die Beispielprogramme befinden sich auf dem FTP-Server *ftp.informatik.uni-osnabrueck.de* unter *pub/hanser/um.*

Literatur

[1] A. Stepanov, M. Lee, *The Standard Template Library*, Hewlett-Packard Laboratories, Palo Alto, CA, 1994.

[2] A. Koenig (Ed.), *Working Paper for Draft Proposed International Standard for Information Systems — Programming Language C++*, Dokumentnr. X3J16/95-0087, WG21/N0687, 1995.

[3] B. Stroustrup *Die Programmiersprache C++ (2. Auflage)*, Addison-Wesley, Bonn, 1992.

[4] N. Josuttis, *Objektorientiertes Programmieren mit C++. Von der Klasse zur Klassenbibliothek*, Addison-Wesley, 1994.

[5] P. J. Plauger *The Draft Standard C++ Library*, Prentice Hall, Englewood Cliffs, NJ, 1995.

[6] D. Louis, *C und C++, Programmierung und Referenz*, Markt & Technik, Haar, 1996.