Kapitel 8
Intervalle

8.1 Einführung

Die bislang diskutierten Containertypen waren Sammlungen diskreter Elemente. Man kann aber auch andere Typen von Elementen Containern sammeln. Beispielsweise Zeichenketten oder Teilmengen der reellen Zahlen, in der Analysis Intervalle genannt.

Unter Voraussetzung elementarer Kenntnisse der Mathematik kann der Intervallbegriff hier relativ rudimentär erläutert werden. Er setzt Mengen voraus, auf denen die Ordnungsrelation „<“ in irgendeiner Form anwendbar ist, und sei es nur auf einzelnen Koordinaten der Elemente. Im Eindimensionalen beinhaltet ein Intervall

$$(a,b], \quad a < b, \quad a,b \in \mathbb{R}$$

alle Zahlen zwischen den Größen a und b einschließlich b (geschlossene Intervallgrenze), aber ausschließlich a (offene Intervallgrenze), also in Mengenschreibweise

$$(a,b] = \{x : a < x \leq b\}$$

Der Intervallbegriff kann weitere Dimensionen ausgedehnt werden. Ist die Grundmenge beispielsweise die der komplexen Zahlen, so kann ein Intervall beispielsweise durch

- die Angabe einer komplexen und einer reellen Zahl festgelegt werden. Elemente des Intervalls sind alle komplexen Zahlen, die innerhalb einer Kreisscheibe mit dem angebenen Radius um die zentrale Zahl liegen:

$$(z,r) = \left\{x : (x-z)\overline{(x-z)} < r^2\right\}$$

  Man beachte: die notwendige Ordnungsrelation ist hier nicht auf der Menge selbst definiert. Auf dieser existiert aber eine Norm bzw. eine Metrik, auf der

G. Brands, *Das C++ Kompendium*, eXamen.press, 2nd ed.,
DOI 10.1007/978-3-642-04787-9_8, © Springer-Verlag Berlin Heidelberg 2010

wiederum die Ordnungsrelation existiert. Diese Art der Intervallbildung wird in der Theorie recht häufig verwendet,ist für die Praxis aber relativ unbrauchbar.

- die Angabe je zweier Real- und Imaginärteile angegeben werden. Elemente des Intervalls sind alle Zahlen innerhalb des von den Grenzen aufgespanntes Rechtecks liegen:

$$(a,b;c,d) = \{x : (a < x.r < b) \wedge (c < x.i < d)\}$$

Die Ordnungsrelation ist hier auf den beiden Koordinaten eines Mengenelementes definiert.

Werden zwei Intervalle vereinigt oder andere Mengenoperationen mit Intervallen ausgeführt, hängt das Ergebnis sowohl von der Art der Intervallgrenzen als auch von Typ der Mengenelemente ab.

- Es gilt $(a,b) \cup [b,c) = (a,c)$ unabhängig von der Art der Menge.
- Ist die Menge diskret (*mit dem Abstand 1 zwischen den Elementen*), so gilt aber auch

$$[a,b] \cup [b+1,c] = [a,c]$$

- Da es sich um Mengenrelationen handelt, führt eine Vereinigung überlappender Intervalle nicht zu einer Mehrfachzählung der Elemente, sondern es gilt

$$(a,b) \cup (b-1,c) = (min(a,b-1),c)$$

- Der Durchschnitt beinhalten die beiden Intervallen gemeinsamen Elemente

$$a < c < b < d\text{: } (a,b] \cap (c,d) = (c,b]$$

- die Differenz enthält die Elemente, die in einer zweiten Menge nicht vorhanden sind

$$a < c < b < d\text{: } (a,b] - (c-d) = (a,c]$$

Die Beziehungen sind auf mehrdimensionale Intervalle verallgemeinerbar, führen jedoch bei der Differenzbildung häufig zu einem Zerfall in viele Intervalle. Wir werden uns hier auf eindimensionale Intervalle konzentrieren und höherdimensionalen Fällen nicht nachgehen.

8.2 Funktion eines Intervallcontainers

Ein Intervallcontainer dient zur Verwaltung einer Menge von Intervallen, wobei die Intervalle mittels der Mengenoperationen verwaltet werden:

- Beim Hinzufügen eines Intervalls wird überprüft, ob es mit bereits vorhandenen Intervallen zu einem größeren Intervall vereinigt werden kann. Die im neuen Gesamtintervall aufgehenden kleineren Intervalle werden entfernt. Daher kann die Anzahl der Intervalle beim Hinzufügen eines weiteren kleiner werden.
- Beim Suchen nach einem Intervall wird alternativ auf
  - Enthaltensein (*vollständiges Aufgehen in einem Intervall des Containers*) oder
  - Überlappung (*mindestens ein gemeinsames Element mit einem Intervall des Containers*)

  geprüft (*nicht leere Schnittmenge*). Während das Enthaltensein eine eindeutige und vollständige Relation ist, kann eine Überlappung mit mehreren Containerelementen vorliegen. Vereinbarungsgemäß gibt ein Container bei einer Suche einen Iterator auf das gefundenen Element zurück. Im Überlappungsfall sind die Folgelemente des zurückgegebenen Iterators ebenfalls zu prüfen, um alle Überlappungen zu finden.
- Beim Löschen eines Intervalls (*Differenzbildung*) wird der Teil des übergebenen Intervalls, der im Container vorhanden ist, gelöscht. Eine Löschoperation kann mehrere Intervalle des Containers betreffen, kann aber auch ein großes Intervall in zwei kleine zerlegen, d.h. die Anzahl der im Containter gespeicherten Elemente kann wachsen.

8.3 Intervallimplementation

Um eine einheitliche Arbeitsweise bei unterschiedlichen Grundmengentypen zu gewährleisten, wird vereinbart:

- Intervalle auf kontinuierlichen Mengen wie reellen Zahlen können beliebigen Typs sein (offene oder geschlossene Grenzen).
- Intervalle diskreter Mengen wie den ganzen Zahlen sind immer vom Typ $[a,b)$, d.h. die rechte Intervallgrenze ist immer geschlossen, die linke immer offen.[1]
- Leere Intervall zeichnen sich durch inkompatible Intervallgrenzen aus, d.h.

$$a < b\colon\ (b,a) \vee (a,a)$$

Damit erhalten wir die Intervall-Klassendefinition einschließlich leerer Intervalle

```
template <class T,bool discret=false>
class Intervall_D1 {
public:
  Intervall_D1():lower(Constant<T>::eins()),
                 upper(Constant<T>::null()),
                 l_open(false),u_open(true){}
```

[1] Hierdurch wird vermieden, dass der Abstand von Intervallgrenzen untersucht werden muss. Intervalle werden immer vereinigt, wenn linke und rechte Grenze der Testintervalle übereinstimmen.

```
  Intervall_D1(T const& l, T const& u,
               bool lo=false, bool uo=true):
    lower(l), upper(u), l_open(lo), u_open(uo) {}
  ....
  bool empty() const {
    if(upper<lower) return true;
    if(lower==upper) return l_open || u_open;
    return false;
  }//end function
private:
  T lower,upper;
  bool l_open,u_open;
};//end class

template <class T> class Intervall_D1<T,true> {
public:
  Intervall_D1(T const& l, T const& u,
               bool lo=false, bool uo=true):
  lower(l), upper(u), l_open(false), u_open(true) {
    if(lo) lower++;
    if(!uo) upper++;
  }//end constructor
...
```

Die obere Definition gilt für alle kontinuierlichen Mengen, die untere Spezialisierung für diskrete Mengen mit definierter Inkrementation wie beispeilsweise ganze Zahlen. Für andere diskrete Typen sind ggf. weitere Spezialisierungen vorzunehmen. Dabei ist die Zuordnung einer Menge zu einem Typ nicht unbedingt so trivial, wie dies im ersten Augenblick scheinen mag, wie folgendes Beispiel zeigt:

```
Stringtyp 1:  ["a","ab") , Stringlänge variabel/frei

-- dieser Typ ist formal ein kontinuierlicher Typ, da
   Strings der Form "aazzzzzz..." mit beliebiger Anzahl
   von z zum Intervall gehören

Stringtyp 2: wie vor, Stringlänge begrenzt

-- ist die Stringlänge beispielsweise auf 10 Zeichen
   begrenzt, so wäre der letzte String vor "ab" der
   String "aazzzzzzzz" und die Menge diskret. Eine
   Umwandlung eines Intervalls [a,b] in den Typ [a,c)
   wäre jedoch nicht durch Addition einer „1" zu
   bewerkstelligen.
```

Aufgabe. Der Einfachheit halber wird in der Intervallklasse die Relation ′<′ mit dem dazugehörenden Operator verwendet. Diese Lösung ist natürlich etwas unschön, wenn Objekte verwaltet werden sollen, auf denen diese Relation

natürlicherweise gar nicht definiert ist. Komplexe Zahlen sind ein Beispiel dafür. Hierfür einfach einen `operator<` zu definieren, birgt die Gefahr in sich, an anderer Stelle komplexe Zahlen in Rechnungen einzusetzen, in denen sie nichts zu suchen haben, ein Meckern des Compilers aber unterbleibt.

Gehen Sie dieses Problem in der bekannten Weise an, die Parameterliste der Intervallklasse durch eine Policyklasse für die Relation zu erweitern. Die Standardpolicyklasse, die Sie ebenfalls entwerfen, greift auf `operator<` zurück. Wird jetzt ein Datentyp eingesetzt, der nicht über diesen Operator verfügt, lässt sich eine Spezialisierung dafür implementieren, die Probleme an anderer Stelle vermeidet.

8.4 Relationen zwischen Intervallen

Sämtliche Relationen und Operationen sind relativ einfach zu implementieren. Etwas Aufwand entsteht durch die Unterscheidung offener und geschlossener Grenzen bei gleichen Werten.

8.4.1 Überlappung/Durchschnitt

Eine Überlappung von Intervallen (a,b),(c,d) bzw. ein nichtleerer Durchschnitt liegt vor, wenn $a < c < b < c$ gilt. Ebenso überlappen die Intervalle $[a,b]$,$[b,c]$, wenn auch nur in einem Element. Der Code ist aufgrund der nicht festliegenden Reihenfolge der Intervalle und der Berücksichtigung leerer Intervalle etwas komplexer:

```
template <class T, bool discret> inline
bool overlap(Intervall_D1<T,discret> const& i1,
             Intervall_D1<T,discret> const& i2){
  if(i1.empty() || i2.empty()) return false;
  if(i1.right()<i2.left() || i2.right()<i1.left())
    return false;
  if(i1.right()==i2.left())
    return !(i1.right_open() && i2.left_open());
  if(i1.left()==i2.right())
    return !(i1.left_open() && i2.right_open());
  return true;
}//end function
```

Die Berechnung des Durchschnitts – in Mengennotation trivial – muss ebenfalls eine Reihe Fälle berücksichtigen und erfolgt getrennt für die linke und die rechte Grenze des neuen Intervalls.

```
template <class T, bool b> Intervall_D1<T,b>
schnitt(Intervall_D1<T,b> const& i1,
        Intervall_D1<T,b> const& i2){
  T l,r; bool l_o,r_o;
  if(!overlap(i1,i2))
    return empty<T,b>();
  if(i1.left()<i2.left()){
    l=i2.left();
    l_o=i2.left_open();
  }else if(i1.left()==i2.left()){
    l=i1.left();
    l_o=i1.left_open() || i2.left_open();
  }else{
    l=i1.left();
    l_o=i1.left_open();
  }//endif

  if(i1.right()>i2.right()){
    r=i2.right();
    r_o=i2.right_open();
  }else if(i1.right()==i2.right()){
    r=i1.right();
    r_o=i1.right_open() || i2.right_open();
  }else{
    r=i1.right();
    r_o=i2.right_open();
  }//endif
  return Intervall_D1<T,b>(l,r,l_o,r_o);
}//end function
```

Aufgabe. „Enthaltensein“ und „Gleichheit“ zweier Intervalle sind einfacher darzustellen, so dass die Implementation Ihnen überlassen bleibe. Formulieren Sie die Algorithmen so um, dass Ihre Policyklasse anstelle der $'<'$-Relation zum Einsatz kommt.

8.4.2 Vereinigung und Differenz

Die Vereinigung zweier Intervalle muss wieder eine Reihe Fälle leerer Intervalle oder verschiedener Intervalltypen berücksichtigen, was den Code etwas aufbläht.

```
template <class T, bool discret> inline
Intervall_D1<T,discret> vereinen(Intervall_D1<T,discret>
const& i1,
         Intervall_D1<T,discret> const& i2){
  T l,r; bool l_o,r_o;
```

```
  if(i1.empty()) return i2;
  if(i2.empty()) return i1;
  if(!overlap(i1,i2)) return empty<T,discret>();
  if(i1.left()<i2.left()){
    l=i1.left();
    l_o=i1.left_open();
  }else if(i1.left()==i2.left()){
    l=i1.left();
    l_o=i1.left_open() && i2.left_open();
  }else{
    l=i2.left();
    l_o=i2.left_open();
  }//endif
  if(i1.right()>i2.right()){
    r=i1.right();
    r_o=i1.right_open();
  }else if(i1.right()==i2.right()){
    r=i1.right();
    r_o=i1.right_open() && i2.right_open();
  }else{
    r=i2.right();
    r_o=i2.right_open();
  }//endif
  return Intervall_D1<T,discret>(l,r,l_o,r_o);
}//end function
```

Etwas komplizierter ist die Differenz, da hier in dem Fall, dass eines der Intervalle vollständig im Inneren des anderen liegt, zwei Ergebnisintervalle resultieren

$$a < c < d < b\colon\ (a,b) - (c,d) = (a,c] \cup [d,b)$$

```
template <class T, bool b>
Intervall_D1<T,b> differenz(Intervall_D1<T,b>& i1,
                    Intervall_D1<T,b> const& i2){
  Intervall_D1<T,b> li=empty<T,b>(),
                    ri=empty<T,b>();
  if(!overlap(i1,i2)) return empty<T,b>();
  if(i1.left()<i2.left()){
    li=Intervall_D1<T,b>(i1.left(),i2.left(),
                         i1.left_open(),
                        !i2.left_open());
  }else if(i1.left()==i2.left() &&
           !i1.left_open() && i2.left_open()){
    li=Intervall_D1<T,b>(i1.left(),i1.left(),
                         false,false);
  }//endif
```

```
  if(i2.right()<i1.right()){
    ri=Intervall_D1<T,b>(i2.right(),i1.right(),
                         !i2.right_open(),
                          i1.right_open());
  }else if(i1.right()==i2.right()&&
           !i2.right_open() && i1.right_open()){
    ri=Intervall_D1<T,b>(i1.right(),i1.right(),
                          false,false);
  }//endif
  if(!li.empty()){
    i1=li;
  }else{
    i1=ri;
    ri=empty<T,b>();
  }//endif
  return ri;
}//end function
```

Neben dem Rückgabewert, der nur dann vom leeren Intervall verschieden ist, wenn zwei Ergebnisintervalle entstehen, wird daher das erste Intervall ebenfalls verändert.

8.5 Intervallcontainer

8.5.1 Relationen zwischen Intervallen

Ein speziell für Intervalle gestalteter Container muss die Intervallrelationen beherrschen, denn überlappende oder aneinander anschließende Intervall sind zu vereinigen bzw. bei der Herausnahme eines Intervall ein größeres, das dieses Intervall enthält, aufzuspalten.[2]

Grundlage eines solchen Containers ist zweckmäßigerweise ein Standardbaumcontainer, denn er bietet aufgrund der internen Sortierung die Möglichkeit, schnell Kandidaten für Intervalloperationen zu finden und besitzt außerdem günstige Laufzeiteigenschaften hinsichtlich der zu erwartenden Intervallvereinigungen und Intervallaufteilungen, die zu Lösch- und Einfügeoperationen führen.

Für sortierte Container ist eine Relationen `key_comp(x,y)` notwendig, die den Rückgabewert ′true′ liefert, wenn im übertragenen Sinn `x<y` gilt, wobei sich die Relationen aber in diesem Fall nicht auf den vom Intervalltyp verwalteten Datentyp beziehen (das haben wir ja schon behandelt), sondern auf die Intervalle selbst. Wir legen fest:

---

[2] Will man diese aktive Arbeit mit den gespeicherten Intervallen nicht, kann man natürlich auch einen normalen Container verwenden.

- Die Relation `key_comp(x,y)` ist dann erfüllt, wenn zwischen oberer Intervallgrenze des Intervalls `x` und unterer des Intervalls `y` die „<“ -Relation erfüllt ist.

Diese Relationen ist zusammen mit der Gleichheitsrelation nicht geeignet, Intervalle in einem normalen Container zu verwalten, denn man vollzieht leicht nach, dass die anderen Relationen aus diesen beiden nicht sinnvoll abgeleitet werden können. Außerdem kann man der $'<'$-Relation auch andere sinnvolle Bedeutungen beilegen. Aus mengentheoretischer Sicht sind beispielsweise die Größen von Intervallen unabhängig von deren Lage wesentlich sinnvollere Ordnungskriterien, die auch aus den beiden Standardrelationen die anderen korrekt ergeben. Die Festlegungen gelten daher ausschließlich für die innere Verwaltung der Intervalle im Intervallcontainer.

Aufgabe. Implementieren Sie die interne Relationen als spezielle Templateklasse `key_less`. Implementieren Sie außerdem die beiden Standardrelationen als Operatoren im Sinne der mengentheoretischen Aussagen.

8.5.2 Containerimplementation

Die Vorarbeiten ermöglichen uns nun die Definition einer Intervallcontainerklasse:

```
template <class T, bool discret=true,
          template<class,bool>
              class Intervall = Intervall_D1 >
class IntervallContainer:
        public set<Intervall<T,discret>,key_less >{
private:
  typedef set<Intervall<T,discret>,key_less >  ptype;

public:
  typedef typename ptype::iterator   iterator;
  typedef typename ptype::const_iterator
            const_iterator;
  typedef typename ptype::value_type  value_type;
```

Aufgrund der Übernahme der set-Methoden durch Vererbung müssen nur wenige Methoden überschrieben werden. Wir betrachten hier drei Methoden.[3]

(1) Bei Einfügen eines neuen Intervalls werden nur nicht-leere Intervalle berücksichtigt. Alle überlappenden Intervalle werden mit dem neuen Intervall vereinigt und aus dem Container entfernt, abschließend wird das neue Intervall hinzugefügt. Das neue Intervall kann bei der Suche mit der Methode

[3] Möglicherweise ist das Überschreiben weiterer Methoden in speziellen Anwendungen sinnvoll, jedoch wurde keine intensive Analyse in diese Richtung vorangetrieben.

lower_bound mit dem letzten Intervall vor dem gefundenen sowie ggf. mit weiteren folgenden überlappen.

```
iterator insert(value_type const& t){
  iterator it;
  if(t.empty()) return ptype::end();
  Intervall<T,discret> tt(t);
  it=lower_bound(tt);
  if(it!=ptype::begin()){
    it--;
    if(overlap(tt,*it){
      tt=vereinen(tt,*it);
      ptype::erase(it);
    }
  }
  while((it=ptype::lower_bound(tt))!=ptype::end() &&
        overlap(tt,*it){
    tt=vereinen(tt,*it);
    ptype::erase(it);
  }//endwhile
  return ptype::insert(tt).first;
}//end function
```

(2) Beim Suchen wird ein Intervall gesucht, dass das gesuchte vollständig enthält. Bei der Suche mit lower_bound ist dies entweder das erste Intervall oder es existiert kein Intervall, das die Relation erfüllt.

```
const_iterator find(value_type const& t) const {
    const_iterator it = ptype::lower_bound(t);
    if(it==ptype::end())
        return it;
    if(schnitt(tt,*it)==tt)
        return it;
    else
        return ptype::end();
}//end function
```

(3) Die letzte Funktion, die wir hier betrachten, ist die Löschfunktion, die alle Schnittmengen mit dem Testintervall entfernt.

```
const_iterator erase(value_type const& t){
    value_type v1;
    iterator it;
    while((it=ptype::lower_bound(t))!=ptype::end()){
        v1=*it;
        ptype::erase(it);
```

```
            insert(differenz(v1,t));
            insert(v1);
        }//endwhile
        return ptype::end();
    }//end function
```

Aufgabe. Ein Intervall wird bei dieser Implementation nur dann gefunden, wenn es vollständig in einem gespeicherten enthalten ist. Man könnte auch nach den Teilintervallen fragen, die im Container zur Verfügung stehen. Entwerfen Sie eine Methode, die dies bewerkstelligt. Beachten Sie: von einem Intervall im Container darf nur der Teil geliefert werden, der tatsächlich zur Verfügung steht, und der kann eine Teilmenge des Containerintervalls sein.