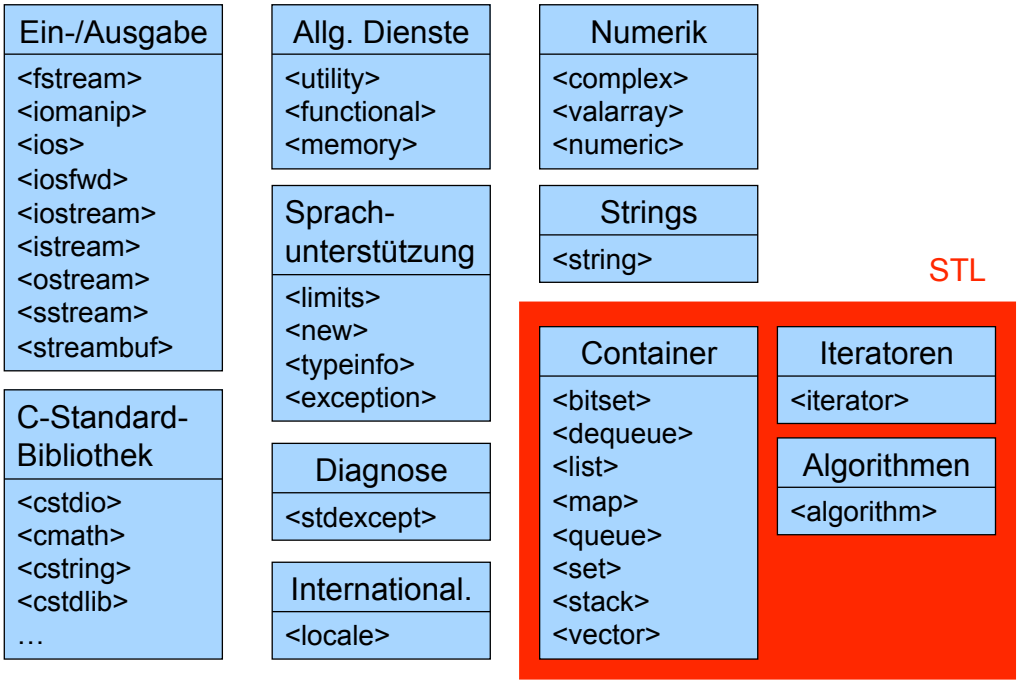
Teil 8:
Standard Template Library - STL

- Übersicht
- Konzepte
- Iteratoren
- Container
- Algorithmen

C++ - Standardbibliothek

- STL bildet einen Teil der C++-Standardbibliothek



Teil 8:
Standard Template Library - STL

- Übersicht
- Konzepte
- Iteratoren
- Container
- Algorithmen

Konzepte – Container

- Container = Datenbehälter mit bestimmter Zugriffsorganisation
  Beispiele:
  - Stack (Last-In First-Out mit Operationen push, pop, top)
  - Queue (First-In First-Out mit Operationen push, pop, front)
  - Vector (wahlfreier Zugriff)

- Typ-Parameter für Container-Elemente (template-Mechanismus)
  Beispiele:
  - stack<int> s;
  - queue<string> q;

- Evtl. weitere template-Parameter für Implementierung und Vergleichsoperator

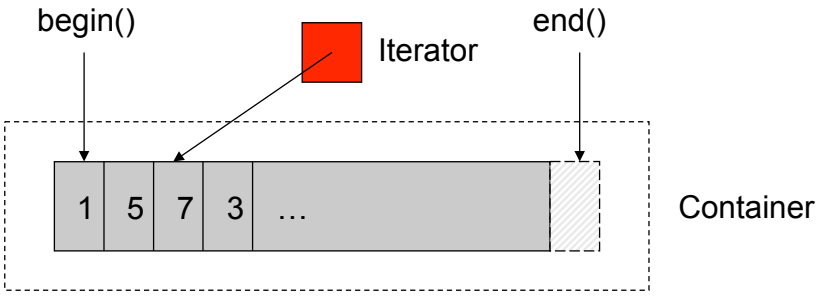
Beispiel mit Stack und Queue

```
#include <queue>
#include <stack>
#include <iostream>

using namespace std;

int main()
{
    // Von Standardeingabe in Queue
    // einlesen:
    queue<int> aQueue;
    int x;

    while(cin >> x)
        aQueue.push(x);
```

```
    // Queue in Stack umkopieren:
    stack<int> aStack;

    while (! aQueue.empty() ) {
        x = aQueue.front();
        aQueue.pop();
        aStack.push(x);
    }

    // Stack ausgeben:
    while (! aStack.empty() ) {
        cout << aStack.top() << endl;
        aStack.pop();
    }
}
```


Konzepte - Iterator

- Iteratoren sind Objekte, mit denen auf die verschiedenen Container in einheitlicher Weise zugegriffen werden kann
- Fast jeder Container bietet Zugriff über Iteratoren an
- Iteratoren verhalten sich ähnlich wie C-Zeiger, mit denen alle Elemente eines Feldes besucht werden können

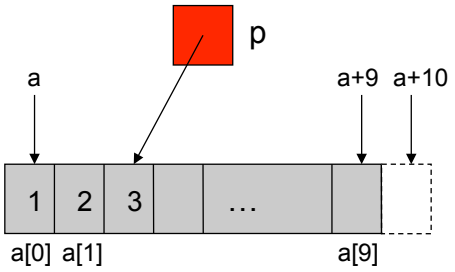


- begin() ist ein Zeiger auf erstes Element im Container und end() ist ein Zeiger auf die Position nach dem letztem Element

Beispiele mit Iteratoren (1)

Durchwandern eines C-Feldes mit einem Zeiger

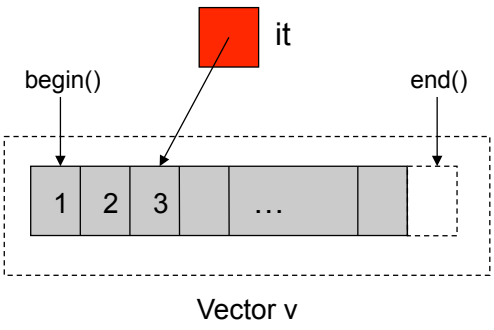
```
#include <iostream>

int main()
{
      const int size = 10;

      // Feld mit Zugriffszeiger definieren:
      int a[size];
      int* p = a;

      // Feld mit Zeigerzugriff beschreiben:
      for (int i = 1; i <= size; i++)
            *p++ = i;

      // Feld mit Zeigerzugriff ausgeben:
      for (p = a; p != a+size; p++)
            cout << *p << ", ";
      cout << endl;
}
```



Beispiele mit Iteratoren (2)

Durchwandern eines Vektors mit einem Iterator

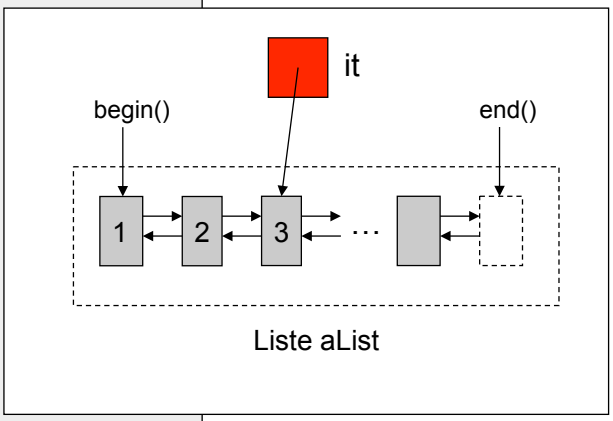
```
#include <vector>
#include <iterator>
#include <iostream>

int main()
{
      const int size = 10;

      // Vektor mit Zugriffs-Iterator definieren:
      vector<int> v(size);
      vector<int>::iterator it = v.begin();

      // Vektor mit Iterator-Zugriff beschreiben:
      for (i = 1; i <= size; i++)
            *it++ = i;

      // Vektor über Iterator-Zugriff ausgeben:
      for (it = v.begin();  it != v.end(); it++)
            cout << *it << ", ";
      cout << endl;
}
```



Beispiele mit Iteratoren (3)

```
#include <list>
#include <iterator>
#include <iostream>

int main()
{
      const int size = 10;

      // Liste mit Zugriffs-Iterator definieren:
      list<int> aList;
      list<int>::iterator it;

      // Liste mit Zugriffsoperation
      //push_back beschreiben:
      for (i = 1; i <= size; i++)
            aList.push_back(i);

      // Liste über Iterator-Zugriff ausgeben:
      for (it = aList.begin();  it != aList.end(); it++)
            cout << *it << ", ";
      cout << endl;
}
```

Durchwandern einer Liste mit einem Iterator



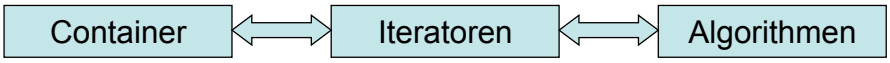
Konzepte - Algorithmen

- Es gibt eine Vielzahl von Algorithmen, die auf alle bzw. mehrere Container-Arten anwendbar sind

  Beispiele:
  - Sortieren
  - Suchen
  - Kopieren
  - Transformieren
  - …

- Algorithmen greifen über Iteratoren auf Container-Daten zu;

  Beispiel:
  - void sort(iterator first, iterator last);



Beispiele mit Algorithmen (1)

Sortieren mit:

sort(first, last);

Es wird der Containerteil von der Iterator-Anfangsposition first einschl. bis zur Iterator-Endposition last ausschließlich sortiert.

Diese Funktion kann nur für Container mit Iteratoren für Direktzugriff verwendet werden.

```
#include <algorithm>
#include <vector>
#include <iostream>

int main()
{
    // vector mit zufälligen Zahlen füllen:
    vector<int> v;

    for (int i = 1; i <= 1000; i++)
        v.push_back( rand()%100 );

    // Vector sortieren:
    sort(v.begin(),v.end());

    // Vector ausgeben:
    vector<int>::iterator it;
    for (it = v.begin();  it != v.end(); it++)
        cout << *it << ", ";
    cout << endl;

}
```


Beispiele mit Algorithmen (2)

Suchen mit:

it = find(first, last, value);

Es wird der Containerteil von der Iterator-Anfangsposition first einschl. bis zur Iterator-Endposition last ausschließlich nach value durchsucht.

Liefert Iterator-Position auf gefundenes Element zurück. Falls Element nicht gefunden wird, wird last zurückgeliefert.

Beispiele mit Algorithmen (3)

Suche in String-Vektor

```
#include <algorithm>
#include <vector>
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main()
{
    // vector mit Strings aus dat.txt füllen:
    vector<string> v;

    ifstream fin("dat.txt");
    string s;

    while (fin >> s)
        v.push_back(s);

    // Gesuchter String:
    s = "Microsoft";

    // String s suchen:
    vector<string>::iterator it;
    it = find( v.begin(), v.end(), s );

    // Suchergebnis ausgeben:
    if (it != v.end())
        cout << s + " ist vorhanden"
             << endl;
    else
        cout << s + " ist nicht vorhanden"
             << endl;

    return 0;
}
```


Beispiele mit Algorithmen (4)

Suche in String-Menge (set<string>)

Unterschiede zur Suche in String-Vector sind blau markiert.

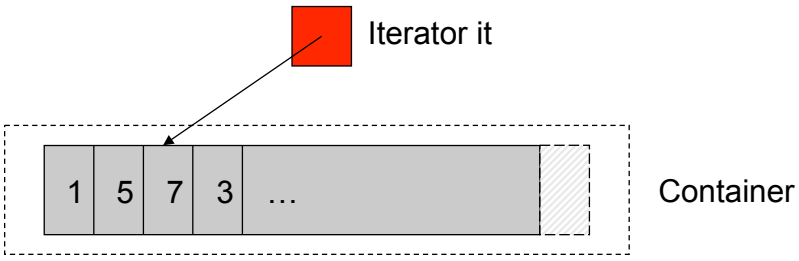
```
#include <algorithm>
#include <set>
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main()
{
    // vector mit Strings aus dat.txt füllen:
    set<string> v;

    ifstream fin("dat.txt");
    string s;

    while (fin >> s)
        v.insert(s);

    // Gesuchter String:
    s = "Microsoft";

    // String s suchen:
    set<string>::iterator it;
    it = find( v.begin(), v.end(), s );

    // Suchergebnis ausgeben:
    if (it != v.end())
        cout << s + " ist vorhanden"
             << endl;
    else
        cout << s + " ist nicht vorhanden"
             << endl;

    return 0;
}
```

Teil 8:
Standard Template Library - STL

- Übersicht
- Konzepte
- Iteratoren
- Container
- Algorithmen

Iteratoren

- Mit Iteratoren können Elemente eines Containers iterativ (d.h. mittels einer Schleife) besucht werden
- Jeder Iterator zeigt auf eine Position (Element) des Containers
- Iteratoren bieten - ähnlich wie C-Zeiger - folgende Operationen an:
  - Adress-Arithmetik: it+n, it-n, it+= n, it -=n, it[n] (entspr. *(it+n))
  - Inkr./Dekr. ++it, it++, --it, it--
  - Dereferenzierung *it
  - Gleichheit i1 == i2, it1 != it2
  - Ordnungsrelation it1 < it2, it1 <= it2, it1 > it2, it1 >= it2



Iterator-Kategorien

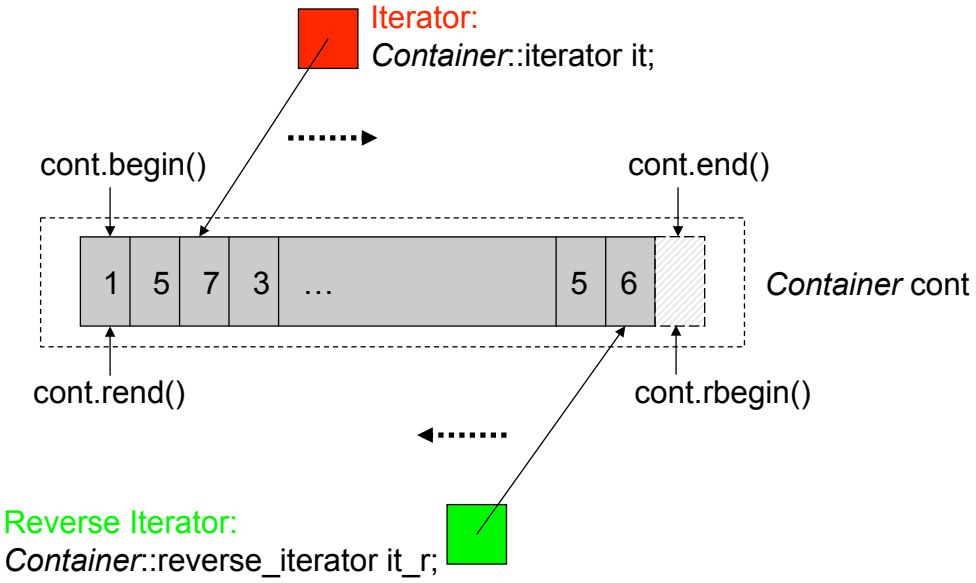
	Wahlfreier Zugriff (Random Access) Iterator	Bidirektionaler (Bidirectional) Iterator	Vorwärts- (Forward) Iterator	Eingabe- (Input) Iterator	Ausgabe- (Output) Iterator
Adress-Arithmetik	X				
Inkr./Dekr.	X	X	Inkr.	Inkr.	Inkr.
Derefenz-ierung	X	X	X	Nur lesend … = *it;	Nur schreibend *it = …;
Gleichheit	X	X	X	X	X
Ordnungs-relation	X				

Beachte: Damit ist ein Iterator mit wahlfreiem Zugriff auch ein bidirektionaler Iterator und ein bidirektionaler Iterator ist auch ein Vorwärts-Iterator usw.

Container und Iteratoren (1)

- Alle Container außer stack, queue und priority_queue bieten eine einheitliche Iterator-Schnittstelle an.
- **Iterator-Typen:**
  - iterator — vorwärts laufender Iterator
  - const_iterator — derefenziertes Element darf nicht verändert werden
  - reverse_iterator — Reversions-Iterator (rückwärts laufender Operator)
  - const_reverse_iterator — derefenziertes Element darf nicht verändert werden
- **Methoden zum Positionieren der Iteratoren:**
  - begin(); Zeiger auf erstes Element im Container
  - end(); Zeiger auf Position nach letztem Element
  - rbegin(); wie end() jedoch für Reverse-Iterator
  - rend(); wie begin() jedoch für Reversions-Iterator
- **Besonderheiten:**
  - Reversions-Iteratoren werden nur von den Containern angeboten, die auch Random-Access- und bidirektionale Iteratoren anbieten
  - wird ein Reversions-Iterator derefenziert, dann wird das unmittelbar davor stehende Element angesprochen (aus Symmetriegründen; siehe auch Programm-Beispiel)

Container und Iteratoren (2)



Beispiele mit Iteratoren (1)

```
#include <iostream>
#include <iterator>
#include <list>

int main()
{
      list<int> tab;
      list<int>::iterator it;
      list<int>::reverse_iterator it_r;

      const int size = 10;

      // Liste mit Quadratzahlen füllen:
      for (int i = 1; i <= size; i++)
            tab.push_back(i*i);
```

```
// Liste mit Iterator ausgeben:
      for (it = tab.begin(); it != tab.end(); ++it)
            cout << *it << ", ";
      cout << endl;
```

Ausgabe: 1, 4, 9, 16, 25, 36, 49, 64, 81, 100,

```
// Liste mit Reversions-Iterator ausgeben:
      for (it_r = tab.rbegin(); it_r != tab.rend(); ++it_r)
            cout << *it_r << ", ";
      cout << endl;
```

Ausgabe: 100, 81, 64, 49, 36, 25, 16, 9, 4, 1,

```
      return 0;
}
```

Beachte die Symmetrie der beiden Ausgabe-Schleifen. Mit dem derefenzierten Reversions-Iterator *it_r wird auf das unmittelbar vor it_r stehende Element zugegriffen.

Beispiele mit Iteratoren (2)

```
#include <iostream>
#include <iterator>
#include <vector>

int main()
{
      const int size = 10;

      vector<int> tab;
      vector<int>::iterator it;

      // Liste mit Quadratzahlen füllen:
      it = tab.begin();
      for (int i = 1; i <= size; i++)
            *it++ = i*i;

      // Liste mit Iterator ausgeben:
      for (it = tab.begin(); it != tab.end(); ++it)
            cout << *it << ", ";
      cout << endl;
}
```

Beachte: Iterator-Position muss definiert sein! D.h. es muss Speicherplatz an der Iterator-Position vorhanden sein.

Da noch kein Speicherplatz für Vektor tab angelegt ist, führt Schleife zum Absturz.

Frage: Wie muss das Programm verändert werden, um den Fehler zu vermeiden?

Iteratoren für Ein-/Ausgabeströme (streams)

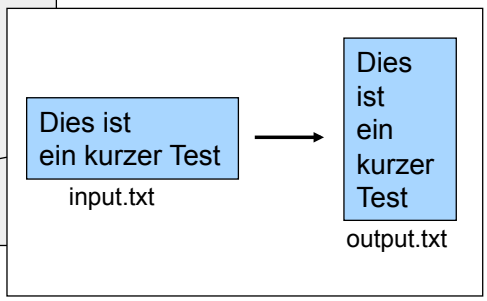
- Eingabestrom- bzw. Ausgabstrom-Iteratoren (istream and ostream iterators) sind spezielle Input- bzw. Output-Iteratoren
- Sie unterstützen das Einlesen von einem Eingabestrom bzw. das Schreiben auf einem Ausgabestrom
- **Eingabestrom-Iterator:**
  - istream_iterator<T> it(istr);
    Definition eines Iterators it zum Einlesen von Elementen vom Typ T vom Eingabestrom istr.
    Wird istr weggelassen (Default Constructor), dann wird it mit Stream-Ende initialisiert.
- **Ausgabestrom-Iterator:**
  - ostream_iterator<T> it(ostr, char* seperator = "");
    Iterator it zum Schreiben von Elementen vom Typ T auf dem Ausgabestrom ostr.
    Die Ausgaben werden durch die Zeichenkette seperator getrennt

Beispiele mit Stream-Iteratoren (1)

```
#include <iostream>
#include <fstream>
#include <iterator>
#include <string>

int main()
{
      // Eingabestrom fin:
      ifstream fin("input.txt");

      // Ausgabestrom fout:
      ofstream fout("output.txt");

      istream_iterator<string> inIt(fin), end;
      ostream_iterator<string> outIt(fout,"\n");

      while (inIt != end)
            *outIt++ = *inIt++;
}
```

Definition des Iterators inIt für den Eingabestrom fin zum Einlesen von Strings.
end ist ein istream-Iterator, der mit „Strom-Ende“ initialisiert wird.

Definition des Iterators outIt für den Ausgabestrom fout zum Schreiben von Strings.



Beispiele mit Stream-Iteratoren (2)

Frage: Was leistet das Programm?

```
#include <iostream>
#include <fstream>
#include <iterator>
#include <string>
#include <list>
#include <algorithm>

int main()
{
      // Eingabestrom-Iterator:
      ifstream fin("input.txt");
      istream_iterator<string> inIt(fin), end;

      // Ausgabestrom-Iterator:
      ofstream fout("output.txt");
      ostream_iterator<string> outIt(fout,"\n");

      // Liste als Container
      list<string> aList;
      list<string>::iterator it;

      // Kopiere Eingabestrom nach Vektor v:
      while(inIt != end) {
            aList.push_back(*inIt);
            inIt++;
      }

      aList.sort();
      copy(aList.begin(),aList.end(),outIt);
}
```

copy(first,last,outIt);

Kopiert den Bereich first einschl. bis last ausschließlich in den Bereich ab outIt.
first u. last sind Input-Iteratoren.
outIt ist ein Output-Iterator.

Aufgaben

Aufgabe 8.1

Schreiben Sie ein Programm, das von der Tastatur
eine Folge von Gleitkomma-Zahlen in ein Vektor-Container einliest
und danach alle positiven Zahlen (in umgekehrter Reihenfolge) ausgibt.

Aufgabe 8.2

Schreiben Sie ein Programm, das von einer Textdatei input.txt
alle Wörter (Strings) einliest und in einen Listen-Container abspeichert.

Dann sollen alle großgeschriebenen Wörter des Containers durch entsprechend kleingeschriebene ersetzt werden.

Anschließend soll der Container-Inhalt auf eine Datei output.txt geschrieben werden.

Die Reihenfolge der Wörter ist beizubehalten.

Sie können für das Einlesen und das Ausgeben geeignete Stream-Iteratoren verwenden.

Iteratoren zum Einfügen in Container (1)

- **Problem:**
  - Falls ein Container mit einem herkömmlichen Iterator gefüllt wird, dann muss die Iterator-Position definiert sein. Ansonsten kann es im schlimmsten Fall zu einem Programmabsturz kommen.

- **Beispiele:**

```
#include <iterator>
#include <vector>

int main()
{
      vector<int> tab;
      vector<int>::iterator it;

      it = tab.begin();

      for (int i = 1; i <= 10; i++)
            *it++ = i*i;
      // …
}
```

Programmabsturz, da Iterator-Position nicht definiert ist!

```
#include <iterator>
#include <list>

int main()
{
      list<int> aList;
      list<int>::iterator it;

      it = aList.begin();

      for (int i = 1; i <= 10; i++)
            *it++ = i*i;
      // …
}
```

Container wird nicht gefüllt, da Iterator-Pos. nicht definiert ist!

Iteratoren zum Einfügen in Container (2)

- **Einfüge-Iteratoren (insert iterators):**
  Spezielle Ausgabe-Iteratoren zum Einfügen in einem Container
- Einfüge-Iteratoren übernehmen beim Einfügen eines neues Elements in den Container eine eventuell notwendige Speicherplatz-Reservierung.
- Folgende **Iterator-Typen** stehen zur Verfügung:
  - **back_insert_iterator<ContainerType>**

    Einfügen am Container-Ende;
    Benutzt dafür die Container-Methode push_back
  - **front_insert_iterator<ContainerType>**

    Einfügen am Container-Anfang;
    Benutzt dafür die Container-Methode push_front
  - **insert_iterator<ContainerType>**

    Einfügen an einer beliebigen Position des Containers;
    Benutzt dafür die Container-Methode insert()

Beispiel mit Einfüge-Iteratoren (1)

```
#include <iostream>
#include <iterator>
#include <list>

int main()
{
      // Liste mit Einfüge-Iterator
      list<int> v;
      back_insert_iterator< list<int> > aBackInserter(v);

      // Container mit Qudratzahlen füllen:
      for (int i = 1; i <= 10; i++)
            *aBackInserter++ = i*i;

      // Container ausgeben:
      list<int>::iterator it = v.begin();
      while(it != v.end())
            cout << *it++ << ", ";
      cout << endl;

}
```

Ausgabe: 1, 4, 9, 16, 25, 36, 49, 64, 81, 100,

Beispiel mit Einfüge-Iteratoren (2)

```
#include <iostream>
#include <iterator>
#include <list>

int main()
{
      // Liste mit Einfüge-Iterator
      list<int> v;
      front_insert_iterator< list<int> > aFrontInserter(v);

      // Container mit Qudratzahlen füllen:
      for (int i = 1; i <= 10; i++)
            *aFrontInserter++ = i*i;

      // Container ausgeben:
      list<int>::iterator it = v.begin();
      while(it != v.end())
            cout << *it++ << ", ";
      cout << endl;

}
```

Ausgabe: 100, 81, 64, 49, 36, 25, 16, 9, 4, 1,

Beispiel mit Einfüge-Iteratoren (3)

```
#include <iostream>
#include <iterator>
#include <list>
#include <algorithm>

int main()
{
      // Liste mit Einfüge-Iterator
      list<int> v;
      insert_iterator< list<int> >  aInserter(v,v.begin());

      // Container mit Qudratzahlen füllen:
      for (int i = 1; i <= 5; i++)
            *aInserter++ = i*i;

      // Container ausgeben: …

      // v in der Mitte weiter füllen:
      it = find(v.begin(),v.end(),9);
      insert_iterator< list<int> >  aMiddleInserter(v,it);
      for (i = 1; i <= 5; i++)
            *aMiddleInserter++ = i;

       // Container ausgeben: …
}
```

1, 4, 9, 16, 25

1, 4, 1, 2, 3, 4, 5, 9, 16, 25,

Teil 8:
Standard Template Library - STL

- Übersicht
- Konzepte
- Iteratoren
- Container
- Algorithmen

Übersicht über Container

- Sequentielle Container
  - Vektoren, Felder: <vector>
  - Deque (double ended queue): <deque>
    nach beiden Seiten offenes dynamisches Feld
  - Liste: <list>
- Container-Adapter
  - Keller: <stack>
  - Schlange: <queue>
  - Vorrangswarteschlange: <priority_queue>
- Assoziative Container
  - Mengen: <set>, <multiset>
  - Relationen: <map>, <multimap>

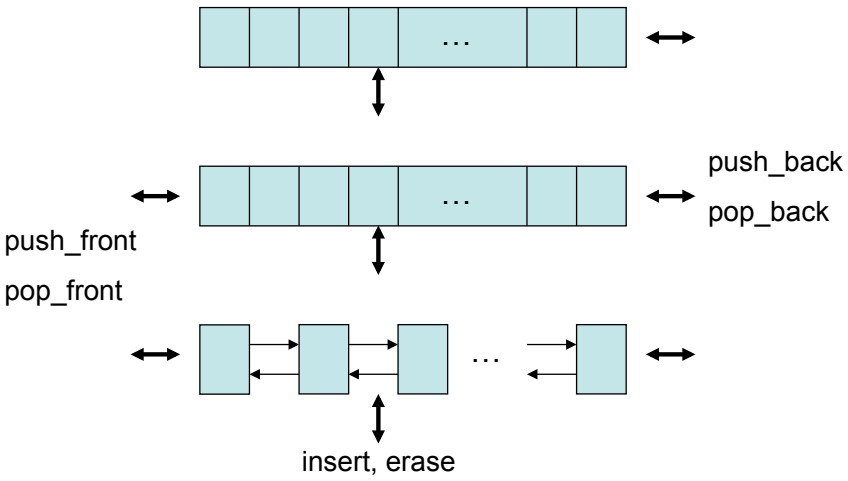
Sequentielle Container

- Datenelemente sind linear (sequentiell) angeordnet.
  Die Position jedes Elements ist durch Folge von
  Einfüge/Lösch-Operationen eindeutig festgelegt.

- Vektor

- Deque

- Liste



Iteratoren u. sequentielle Container

- Iteratoren für Vector und Deque haben wahlfreien Zugriff
  (Random-Access Iterator)

- Iteratoren für List sind bidrektionale Iteratoren.
  Sie erlauben keinen wahlfreien Zugriff.

- Iteratoren können vorwärtslaufend
  - z.B.: list<int>::iterator it;

  als auch rückwärtslaufend (Reversions-Iteratoren) sein
  - z.B.: list<int>::reverse_iterator it_r;

- Einfüge-Iteratoren (insert iterators):
  - Deque, List: front_insert_iterator, back_insert_iterator, insert_iterator
  - Vector: back_insert_iterator, insert_iterator

Konstr./Destr. für sequentielle Container

SeqCont<*TypName*> cont;	Standard-Konstruktor
SeqCont<*TypName*> cont(n); *SeqCont*<*TypName*> cont(n, x); *SeqCont*<*TypName*> cont(first, last);	Container mit n Elementen Container mit n Elementen mit x initialisiert Container, initialisiert mit allen Elementen aus dem Bereich [first, last) d.h. von IteratorPos. first einschl. bis IteratorPos. last ausschließlich
SeqCont<*TypName*> cont2 = cont1; *SeqCont*<*TypName*> cont2(conts1); cont2 = cont1;	Kopierkonstruktor Kopierkonstruktor Zuweisungsoperator
~*SeqCont*<*TypName*>();	Destruktor

Beispiele

```
vector<int>   v1(100);
vector<double>   v2(20, 0.0);
…
list<int>   list1( v1.begin(), v1.end() );
list<int>   list2 = list1;
…
vector< list<string> >   synonymeTab(10);
```


Methoden für sequentielle Container (1)

- Größe und Kapazität

n = cont.size();	Anzahl Elemente
n = vec.capacity();	Maximale Anzahl von Elementen (nur für Vektoren)
vec.resize(n);	Vergrößern eines Vektors um n Elemente
b = cont.empty();	Prüfen, ob Container leer ist

- Zugriff auf Elemente

x = cont.front();	Vorderstes Element
x = cont.back();	Letztes Element
cont[i] = x; x = cont[i];	Indizierter Zugriff; nur für Vector und Deque. Vorsicht: Speicherplatz muss vorhanden sein!
x = cont.at(i); cont.at(i) = x;	Zugriff auf das i-te Element; nur für Vector und Deque. Im Gegensatz zum indiziertem Zugriff wird bei Indexüberschreitung eine „out_of_range exception“ geworfen.

Außerdem: Zugriff über Iteratoren

Methoden für sequentielle Container (2)

- Löschen und Einfügen

cont.push_back(x);	Am Ende x einfügen
cont.pop_back();	Letztes Element löschen
cont.push_front(x);	Am Anfang x einfügen; nicht für Vector
cont.pop_front();	Erstes Element löschen; nicht für Vector
cont.insert(itPos, x);	An Iterator-Position itPos Element x einfügen
cont.insert(itPos, first, last);	An Iterator-Position itPos alle Elemente aus [first, last) einfügen;
cont.erase(itPos);	Element an Iterator-Position itPos löschen
cont.erase(first, last);	Alle Elemente mit Iterator-Pos. in [first, last) löschen
cont.clear();	Alle Elemente löschen

Der Aufwand von insert und erase ist bei Vector und Deque O(n) und bei List O(1).

Spezielle Methoden für Liste

cont.remove(x);	Alle Elemente mit Wert x entfernen; O(n)
cont.remove_if(pred);	Alle Elemente x, für die pred(x) gilt, entfernen.; O(n)
cont.unique();	Aufeinander folgende identische Elemente werden gelöscht. Bei einer sortierten Liste wird also erreicht, dass Elemente nur einfach vorkommen. O(n)
cont.merge(list);	Sortierte Liste cont mit einer anderen sortierten Liste list verschmelzen; O(n+m), wobei m = list.size().
cont.sort();	Container sortieren; O(n log n)
cont.sort(cmp);	Container bzgl. der Vergleichsoperation cmp sortieren
cont.reverse();	Reihenfolge umkehren; O(n)
cont.splice(itPos, list);	(splice = zusammenkleben, verbinden) Fügt den Inhalt von list vor die Position itPos; danach ist list leer. O(m), wobei m = list.size().
cont.splice(itPos,list,it);	Nur Element mit Iterator-Position it wird eingefügt.
cont.splice(itPos,list,first,last)	Alle Elem. mit Iterator-Pos. in [first, last) werden eingefügt

Beispiel mit Deque

```
#include <iostream>
#include <iterator>
#include <deque>


int main()
{
    // Deque definieren:
    deque<int> cont;

    // Elemente 1 bis 5 jeweils vorne
    // und hinten anfügen:
    for (int i = 0; i < 6; ++i)
    {
        cont.push_front(i);
        cont.push_back(i);
    }
```

```
    // Deque mit indiziertem Zugriff
    // ausgeben:
    for (i = 0; i < cont.size(); ++i)
        cout << cont[i] << ", ";
    cout << endl;

    // Deque mittels Iterator ausgeben:
    deque<int>::iterator it = cont.begin();
    while ( it != cont.end() )
        cout << *it++ << ", ";
    cout << endl;
}
```

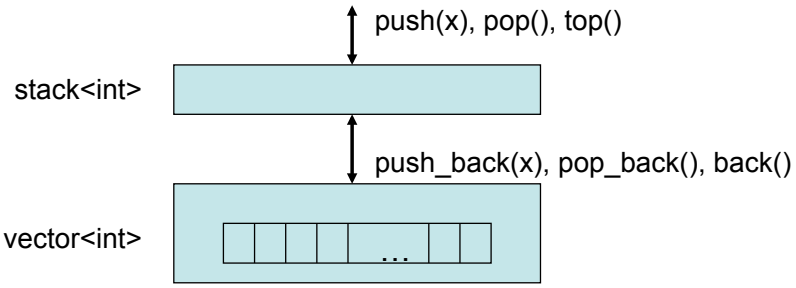
Ausgabe:

5, 4, 3, 2, 1, 0, 0, 1, 2, 3, 4, 5,

Stack, Queue und Priority_Queue

- Stack, Queue und Priority_Queue sind in der STL als sogenannte **Adapter-Klassen** definiert.
  Sie definieren keine eigene Datenstruktur sondern sie werden auf Basis der sequentiellen Container realisiert.
  Damit adaptieren sie die Schnittstellen der sequentiellen Container.

  Beispiel:



- Stack und Priority_queue werden als Vector und Queue als Deque implementiert. Über einen Template-Parameter ließe sich auch eine andere Realisierung einstellen.
- Stack, Queue und Priority_Queue bieten keine Iteratoren an.

Stack

stack<*TypeName*> cont; …	Konstruktor, Kopierkonstruktor, Destruktor, Zuweisungsoperator
n = cont.size();	Anzahl Elemente im Keller
cont.empty();	Prüft, ob Keller leer ist
cont.push(x);	Element x einkellern
cont.pop();	Oberstes Element aus Keller entfernen
x = cont.top();	Oberstes Element

Queue

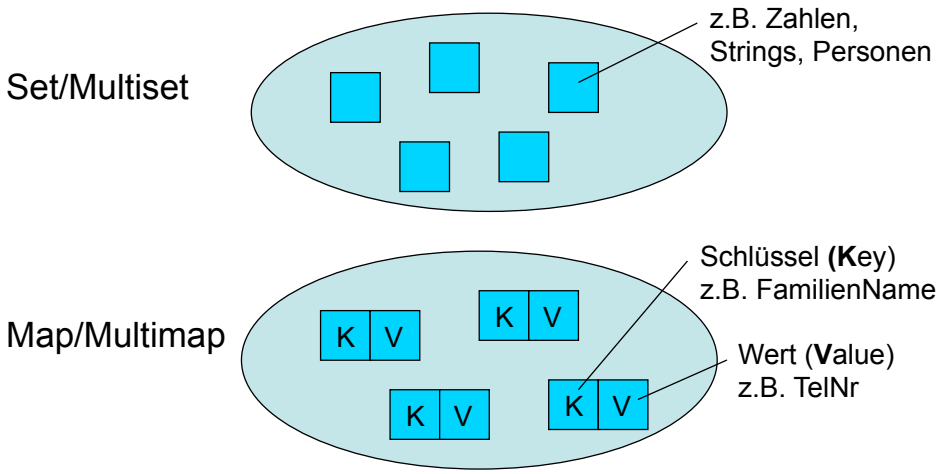
queue<*TypeName*> cont; …	Konstruktor, Kopierkonstruktor, Destruktor, Zuweisungsoperator
n = cont.size();	Anzahl Elemente in der Schlange
cont.empty();	Prüft, ob Schlange leer ist
cont.push(x);	Element x an Schlange hinten anfügen
cont.pop();	Vorderstes Element aus Schlange entfernen
x = cont.front();	Vorderstes Element
x = cont.back();	Letztes Element

Priority_Queue

priority_queue<*TypeName*> cont; priority_queue<*TypeName*, *Comp*> cont; …	Konstruktor, Konstruktor mit Comp als Vergleichsoperator Kopierkonstruktor, Destruktor, Zuweisungsoperator
n = cont.size();	Anzahl Elemente in der PrioQueue
cont.empty();	Prüft, ob PrioQueue leer ist
cont.push(x);	Element x in PrioQueue einfügen; O(log n)
cont.pop();	Größtes Element aus PrioQueue entfernen; O(log n)
x = cont.top();	Größtes Element; O(1)

Assoziative Container (1)

- Assoziative Container sind sortiert.
- Bei Set/Multiset sind die Elemente und bei Map/Multimap die Schlüssel sortiert.
- Die assoziativen Container dienen vor allem zum schnellen Suchen, Löschen und Einfügen von Elementen.



Assoziative Container (2)

- Muliset gestatten im Gegensatz zu Set das mehrfache Vorkommen von Elementen.
- In einer Multimap können - im Gegensatz zu Map – zu einem Schlüssel (Key) mehrere Werte (Value) assoziiert sein.
- Assoziative Container bieten bidrektionale Iteratoren an.
  Sie erlauben keinen wahlfreien Zugriff.
  Iteratoren können sowohl vorwärtslaufend
  - z.B.: set<int>::iterator it;

  als auch rückwärtslaufend (Reversions-Iteratoren) sein
  - z.B.: set<int>::reverse_iterator it_r;
- Als Einfüge-Iteratoren gibt es den insert_iterator:
  - z.B.
    set<int> cont;
    insert_iterator< set<int> > insIt(cont, cont.begin());
- Die üblichen Mengenfunktionen (Vereinigung, Schnitt, Differenz, …) und die Sortierfunktion sind nicht als Methoden sondern als Funktionen realisiert (siehe <algorithm>).

Konstr./Destr. für Set/Multiset

AssCont<*TypName*> cont;	Standard-Konstruktor
AssCont<*TypName*> cont(comp);	Container mit comp als Sortierkriterium
AssCont<*TypName*> cont(first, last);	Container, initialisiert mit allen Elementen aus dem Bereich [first, last) d.h. von IteratorPos. first einschl. bis IteratorPos. last ausschließlich
AssCont<*TypName*> cont(first, last, comp);	Wie oben mit comp als Sortierkriterium
AssCont<*TypName*> cont2 = cont1;	Kopierkonstruktor
AssCont<*TypName*> cont2(conts1);	Kopierkonstruktor
cont2 = cont1;	Zuweisungsoperator
~*AssCont*<*TypName*>();	Destruktor

Beispiele

```
set<int>          m1;
// m1 füllen: …

multiset<int>     m2( m1.begin(), m1.end() );
set<int>          m3( m2.begin(), m2.end() );
```

Methoden für Set/Multiset (1)

- Größe

n = cont.size();	Anzahl Elemente
b = cont.empty();	Prüfen, ob Container leer ist

- Suche von Elementen

n = cont.count(x);	Anzahl Vorkommen von x
it = cont.find(x);	Sucht x und liefert Iterator auf gefundene Position bzw. cont.end(), falls nicht gefunden, zurück.
it = cont.lower_bound(x);	Sucht x und liefert Iterator auf erste gefundene Position zurück.
it = cont.upper_bound(x);	Sucht x und liefert Iterator auf Position unmittelbar nach letztem x zurück
pair = cont.equal_range(x);	Liefert lower_bound und upper_bound als Paar zurück (siehe Beispiel)

Der Aufwand für alle Operationen ist O(log n).

Methoden für Set/Multiset (2)

- Löschen und Einfügen

pair = setCont.insert(x);	Element x in set-Container einfügen. Liefert Wert vom Typ pair<iterator, bool> zurück. Erste Komponente gibt die Iteratorposition an und die zweite Komponente gibt an, ob eingefügt wurde.
it = multisetCont.insert(x);	Element x in multiset-Container einfügen. Liefert Iteratorposition zurück.
it = cont.insert(first, last);	Alle Elemente aus [first, last) einfügen; liefert Iterator auf erstes eingefügtes Element zurück
cont.erase(itPos);	Element an Iterator-Position itPos löschen
cont.erase(first, last);	Alle Elemente mit Iterator-Pos. in [first, last) löschen
n = cont.erase(x);	Alle Vorkommen von x löschen; Anzahl gelöschter Elemente wird zurückgeliefert.
cont.clear();	Alle Elemente löschen

Der Aufwand für insert(x) und erase(x) ist O(log n).

Beispiel für Set/Multiset (1)

```
#include <set>
…

int main()
{
      // Feld a:
      const int size_a = 12;
      int a[size_a] = {10,2,4,1,2,4,3,4,5,6,7,10};

      // Set s mit Feld a initialisiert:
      set<int> s(a, a+size_a);

      // Set s über Iterator-Zugriff ausgeben:
      set<int>::iterator it;
      for (it = s.begin();  it != s.end(); it++)
            cout << *it << ", ";
      cout << endl;

      // Multiset m mit Feld a initialisiert:
      multiset<int> m(a, a+size_a);

      // Multiset m über Iterator-Zugriff ausgeben:
      multiset<int>::iterator it_m;
      for (it_m = m.begin();  it_m != m.end(); it_m++)
            cout << *it_m << ", ";
      cout << endl;
```

1, 2, 3, 4, 5, 6, 7, 10

1, 2, 2, 3, 4, 4, 4, 5, 6, 7, 10, 10

Beispiel für Set/Multiset (2)

m = 1, 2, 2, 3, 4, 4, 4, 5, 6, 7, 10, 10

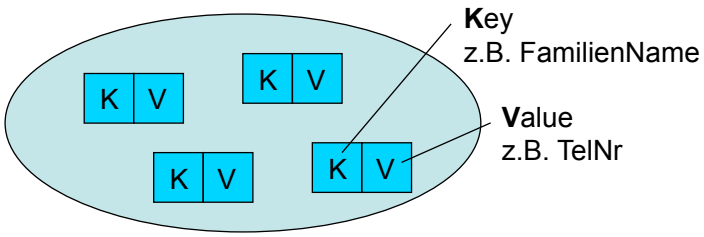
```
      // Alle Vorkommen von 4 in m suchen:
      pair<multiset<int>::iterator,multiset<int>::iterator> it_pair;
      it_pair = m.equal_range(4);

      for (it_m = it_pair.first;  it_m != it_pair.second; it_m++)
            cout << *it_m << ", ";
      cout << endl;


      // Alle Vorkommen von 4 bis auf eines löschen:
      m.erase(++it_pair.first,it_pair.second);

      for (it_m = m.begin();  it_m != m.end(); it_m++)
            cout << *it_m << ", ";
      cout << endl;

      return 0;
}
```

4, 4, 4

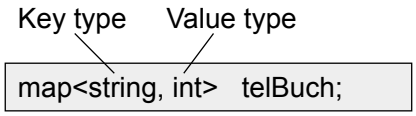
1, 2, 2, 3, 4, 5, 6, 7, 10, 10

Map/Multimap

- Bei einer Map/Multimap wird über einen Schlüssel (Key) auf den eigentlichen Wert eines Elements (Value) zugegriffen.



- Bei der Definition einer Map/Multimap muss der Schlüssel- und Element-Typ festgelegt werden; z.B.:



- Bei einer Map kann der Elementzugriff (wie bei Vektor) auch indiziert erfolgen:

```
telBuch ["Maier"] = 411;
telBuch ["Anton"] = 513;
cout << telBuch ["Maier"];
```


Konstr./Destr. für Map/Multimap

AssCont<KeyName,ValueName> cont;	Standard-Konstruktor
AssCont<KeyName,ValueName> cont(comp);	Container mit comp als Sortierkriterium
AssCont<KeyName,ValueName> cont(first, last);	Container, initialisiert mit allen Elementen aus dem Bereich [first, last) d.h. von IteratorPos. first einschl. bis IteratorPos. last ausschließlich
AssCont<KeyName,ValueName> cont(first, last, comp);	Wie oben mit comp als Sortierkriterium
AssCont<KeyName,ValueName> cont2 = cont1;	Kopierkonstruktor
AssCont<KeyName,ValueName> cont2(conts1);	Kopierkonstruktor
cont2 = cont1;	Zuweisungsoperator
~*AssCont<KeyName,ValueName>*();	Destruktor

Beispiel

```
typedef string Name;
typedef string TelNr;

// Telefonbuch definieren:
map<Name,TelNr> telBuch;
```

Methoden für Map/Multimap (1)

- Größe

n = cont.size();	Anzahl Elemente
b = cont.empty();	Prüfen, ob Container leer ist

- Suche von Elementen

n = cont.count(key);	Anzahl Vorkommen von Elemente mit Schlüssel key
it = cont.find(key);	Sucht Element mit Schlüssel key und liefert Iterator auf gefundene Position bzw. cont.end(), falls nicht gefunden, zurück.
it = cont.lower_bound(key);	Sucht Element mit Schlüssel key und liefert Iterator auf erste gefundene Position zurück.
it = cont.upper_bound(key);	Sucht Element mit Schlüssel key und liefert Iterator auf Position unmittelbar nach letztem x zurück
pair = cont.equal_range(key);	Liefert lower_bound und upper_bound als Paar zurück

Der Aufwand für alle Operationen ist O(log n).

Methoden für Map/Multimap (2)

- Zugriff auf Elemente

cont[key] = value; value = cont[key];	Indizierter Zugriff nur für Map. O(log n). Existiert ein Element mit key bereits, dann wird mit cont[key] = new_value, der alte Wert überschreiben. Vorsicht: Falls ein Element mit Schlüssel key nicht existiert, dann wird mit dem Zugriff cont[key] ein neues Element angelegt. Das geschieht auch dann, wenn cont[key] auf der rechten Seite einer Zuweisung benutzt wird.

Außerdem: Zugriff über Iteratoren.

Elemente des Containers sind key-value-Paare, die als Strukturen mit den Komponenten first und second realisiert sind. (siehe Beispiel)

Methoden für Map/Multimap (3)

- Einfügen und Löschen:

pair = mapCont.insert(key-value-pair);	Element (key,value) in eine Map einfügen. Bei einer Map wird das Element nur eingefügt, falls ein Element mit dem Schlüssel key noch nicht existiert. Liefert Wert vom Typ pair<iterator, bool> zurück. Erste Komponente gibt die Iteratorposition an und die zweite Komponente gibt an, ob eingefügt wurde.
it = multimapCont.insert(key-value-pair);	Element (key,value) in eine Multimap einfügen. Liefert Iterator zurück.
it = cont.insert(first, last);	Alle Elemente aus [first, last) einfügen; liefert Iterator auf erstes eingefügtes Element zurück

cont.erase(itPos);	Element an Iterator-Position itPos löschen
cont.erase(first, last);	Alle Elemente mit Iterator-Pos. in [first, last) löschen
n = cont.erase(key);	Alle Vorkommen von Elementen mit Schlüssel key löschen; Anzahl gelöschter Elemente wird zurückgeliefert.
cont.clear();	Alle Elemente löschen

Der Aufwand für insert und erase ist O(log n).

Beispiel mit Map: Telefonbuch (1)

```
#include <map>
#include <iostream>
#include <iterator>
#include <string>

int main()
{
      // Telefonbuch definieren:
      typedef string Name;
      typedef string TelNr;
      map<Name,TelNr> telBuch;

      // In Telefonbuch indiziert eintragen:
      telBuch["Maier"] = "07531/40234";
      telBuch["Anton"] = "07731/23425";
      telBuch["Zimmermann"] = "07531/32545";

      // In Telefonbuch mit insert einfügen:
      pair<Name,TelNr> eintrag;
      eintrag.first = "Baier";
      eintrag.second = "07732/32547";
      telBuch.insert(eintrag);
```

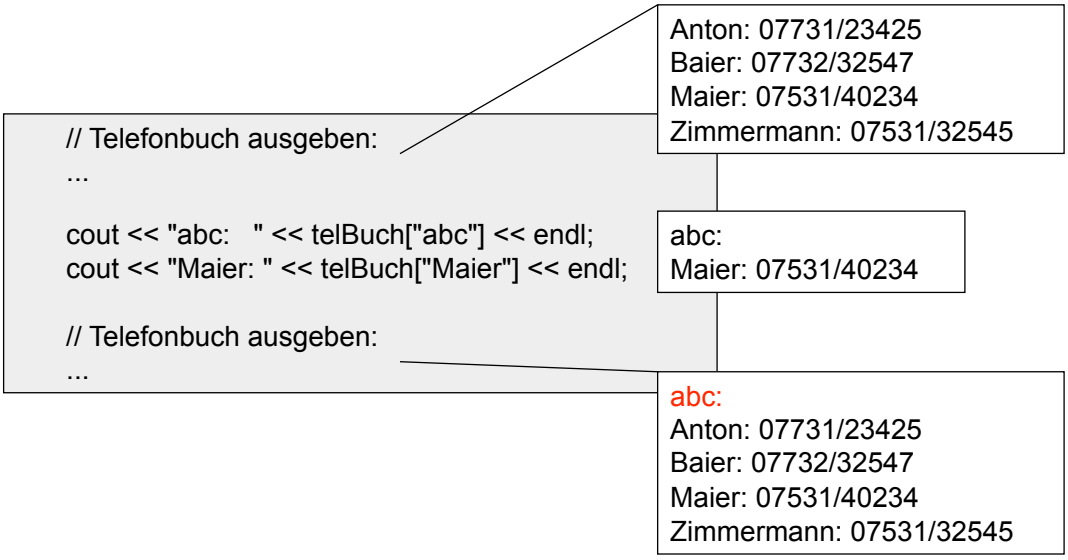
```
      // Telefonbuch ausgeben:
      map<Name,TelNr>::iterator it;
      it = telBuch.begin();
      while ( it != telBuch.end() )
      {
            cout << it->first << ": "
                 << it->second << endl;
            ++it;
      }
```

Ausgabe:

Anton: 07731/23425
Baier: 07732/32547
Maier: 07531/40234
Zimmermann: 07531/32545

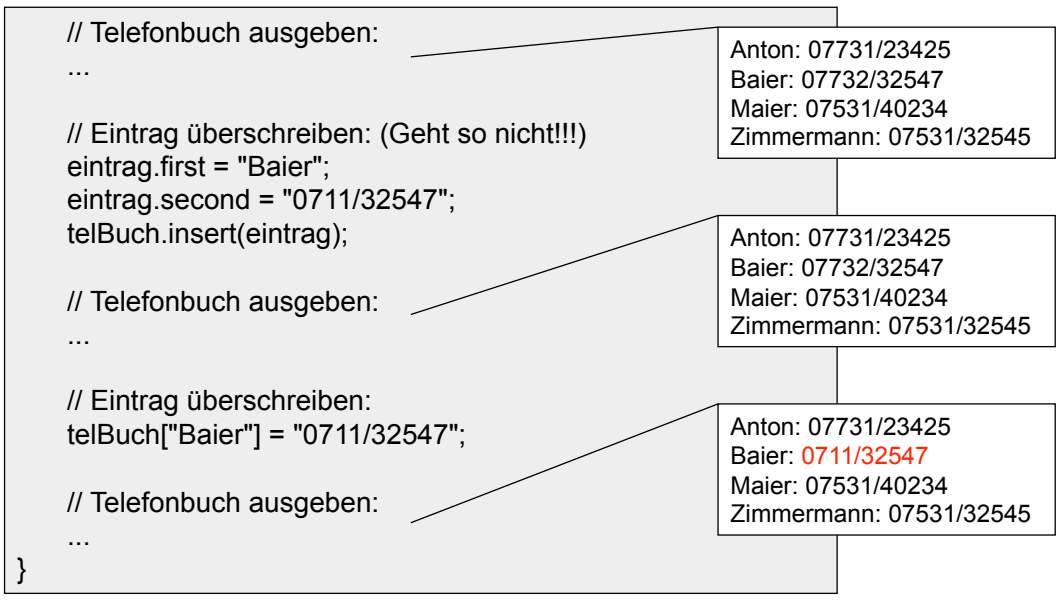
Beispiel mit Map: Telefonbuch (2)

Beachte:
Falls ein Element mit Schlüssel key nicht existiert,
dann wird mit dem Zugriff cont[key] ein neues Element angelegt.
Das geschieht auch dann, wenn cont[key] auf der rechten Seite
einer Zuweisung benutzt wird.



Beispiel mit Map: Telefonbuch (3)

Beachte:
Einfügen mit insert, nur falls Element mit Schlüssel nicht bereits existiert.
Wert eines Elements mittels indiziertem Zugriff überschreiben.



Beispiel mit Multimap: Wörterbuch (1)

```
#include <map>
#include <iostream>
#include <iterator>
#include <string>

int main()
{
    // Wörterbuch definieren:
    typedef string Deutsch;
    typedef string Englisch;
    multimap<Deutsch,Englisch> wBuch;

    // In Wörterbuch mit insert einfügen:
    pair<Deutsch,Englisch> eintrag;

    eintrag.first = "reden";      eintrag.second = "talk";     wBuch.insert(eintrag);
    eintrag.first = "reden";      eintrag.second = "speak";    wBuch.insert(eintrag);
    eintrag.first = "schreiben";  eintrag.second = "write";    wBuch.insert(eintrag);
    eintrag.first = "gehen";      eintrag.second = "go";       wBuch.insert(eintrag);
    eintrag.first = "gehen";      eintrag.second = "walk";     wBuch.insert(eintrag);
```


Beispiel mit Multimap: Wörterbuch (2)

```
    // Wörterbuch ausgeben:
    multimap<Deutsch,Englisch>::iterator it;
    for (it = wBuch.begin();  it != wBuch.end(); it++)
        cout << it->first << ": " << it->second << endl;

    // Alle Einträge zu "reden":
    string w = "reden";
    cout << w << ": " << wBuch.count(w) << " Eintraege\n";
    for (it = wBuch.lower_bound(w); it != wBuch.upper_bound(w); ++it)
        cout << it->second << endl;

    // Eintrag "reden" löschen:
    cout << w << ": " << endl;
    cout << wBuch.erase(w) << " Eintrage geloescht!\n";

    // Wörterbuch ausgeben:
    for (it = wBuch.begin();  it != wBuch.end(); it++)
        cout << it->first << ": " << it->second << endl;

    return 0;
}
```

```
gehen: go
gehen: walk
reden: talk
reden: speak
schreiben: write

reden: 2 Eintraege
talk
speak

reden:
2 Eintrage geloescht!

gehen: go
gehen: walk
schreiben: write
```

Aufgaben

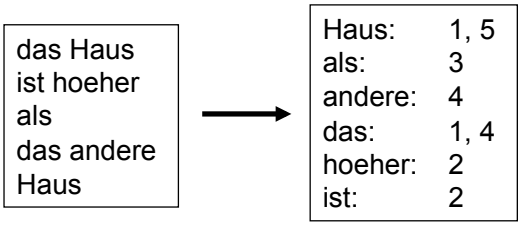
Aufgabe 8.3 Rechtschreibprüfung

Schreiben Sie ein Programm, das jedes Wort w einer Textdatei *input.txt* auf korrekte Rechtschreibung prüft, indem festgestellt wird, ob w in einem Wörterbuch *woerterBuch.txt* vorkommt.
Falsch geschriebene Wörter werden in die Datei *output.txt* abgespeichert.

Aufgabe 8.4 Indexerstellung

Schreiben Sie ein Programm, das für jedes Wort w einer Textdatei input.txt die Nummern der Zeilen ermittelt, in denen w vorkommt. Wörter und Zeilennummern sollen alphabetisch sortiert in eine Datei output.txt ausgegeben werden.

Beispiel:

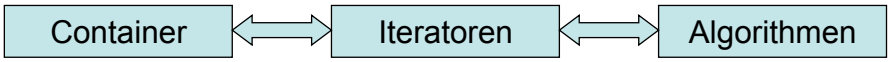


Teil 8:
Standard Template Library - STL

- Übersicht
- Konzepte
- Iteratoren
- Container
- Algorithmen

Algorithmen (1)

- Algorithmen sind keine Methoden sondern Funktionen, die über Iteratoren auf die Container zugreifen.



  Beispiel: void sort(iterator first, iterator last);

- Vorteil dabei ist, dass ein Algorithmus nur einmal implementiert werden muss, aber auf (fast) alle Container-Klassen und sogar C-Feldern anwendbar ist.

  Nachteil ist, dass für manche Container-Klassen die Algorithmen-Funktionen langsamer sind als die speziellen Container-Funktionen. (z.B. ist find-Methode für Set O(log n) während find-Algorithmus O(n) ist)

- Einteilung der Algorithmen:
  - Suchen
  - Sortieren
  - Modifizieren: Elemente verändern u. löschen, Reihenfolge ändern
  - Kopieren
  - Mengen-Operationen

Algorithmen (2)

Prädikate (predicate)

Funktionen, die einen boolschen Wert zurückliefern.
Die Anzahl der Parameter nennt man auch Stellenzahl.

Bei den Algorithmen werden sehr häufig Prädikate als Parameter übergeben.

Beispiel für 1-stelliges Prädikat:

```
bool isEven(int n)
{
    if (n%2 == 0)
        return true;
    else
        return false;
}
```

Beispiel für 2-stelliges Prädikat:

```
bool istJuenger(Person p, Person q)
{
    if (p.Alter < q.Alter)
        return true;
    else
        return false;
}
```

Such-Algorithmen (1)

it = find(first,last,value); it = find_if(first,last,pred);	Sucht im Bereich [first,last) nach dem ersten Vorkommen von value bzw. einem Element x, für das pred(x) zutrifft. Falls ein solches Element existiert, wird die Iterator-Pos. zurückgeliefert, sonst last.

it = adjacent_find(first,last); it = adjacent_find(first,last,pred2);	Sucht im Bereich [first,last) nach dem ersten Vorkommen von 2 aufeinander folgenden Elementen, die gleich sind bzw. auf die das 2-stellige Prädikate pred2 zutrifft.

it = min_element(first,last);
it = min_element(first,last,comp);
it = max_element(first,last);
it = max_element(first,last,comp);
	Sucht im Bereich [first,last) nach dem Minimum bzw. Maximum. Vergleichsoperator ist <.
	Es kann aber auch ein eigenes 2-stelliges Prädikat comp benutzt werden.

n = count(first,last,value); n = count_if(first,last,pred);	Bestimmt im Bereich [first,last) die Anzahl Vorkommen von value bzw. von Elementen, für die pred zutrifft.

Such-Algorithmen (2)

it = search(first1,last1, first2,last2); it = search(first1,last1,first2,last2,comp);	Sucht den Bereich [first1,last1) nach dem Muster [first2,last2) ab. Statt auf Gleichheit zu prüfen, kann auch das 2-stellige Prädikate comp benutzt werden.

pair = mismatch(first1,last1, first2); pair = mismatch(first1,last1,first2,comp);	Vergleicht den Bereich [first1,last1) mit dem Bereich ab first2 einschließlich und liefert ein Iterator-Paar für die erste ungleiche Stelle zurück. Statt auf Gleichheit zu prüfen, kann auch das 2-stellige Prädikate comp benutzt werden.

…	

Beispiel mit Such-Algorithmen (1)

```
#include <algorithm>
#include <list>
#include <iostream>
#include <fstream>

bool istAusIntervall(int n)
{
      if(1 <= n && n <= 10)
            return true;
      else
            return false;
}

int main()
{
      // Liste mit Zahlen aus
      // dat.txt füllen:
      list<int> v;
      ifstream fin("dat.txt");
      int x;
      while (fin >> x)
            v.push_back(x);
```

```
      list<int>::iterator it;

      // Suchen nach 12:
      it = find(v.begin(),v.end(),12);
      if (it != v.end())
            cout << *it << " gefunden \n";
      else
            cout << "nicht gefunden \n";

      // Anzahl Vorkommen von 12:
      cout << "Anzahl Vorkommen von 12: "
            << count(v.begin(),v.end(),12)  << endl;

      // Minimum ausgeben:
      cout << *min_element(v.begin(),v.end()) << endl;

      // Zahl aus [1,10] suchen:
      it = find_if(v.begin(),v.end(),istAusIntervall);
      if (it != v.end())
            cout << *it << " ist aus Intervall \n";
      else
            cout << "Keine Zahl aus Intervall gefunden \n";
```

istAusIntervall ist ein 1-stelliges Prädikat

Beispiel mit Such-Algorithmen (2)

```
      // Fortsetzung von vorhergehender Seite:


      // Alle Vorkommen von 12 suchen:
      it = v.begin();
      while (( it = find(it,v.end(),12) ) != v.end() )
            cout << *it++ << " gefunden \n";


      // Muster suchen:
      int a[ ] = {1,2,3};
      list<int> muster(a,a+3);
      it = search(v.begin(),v.end(),muster.begin(),muster.end());
      if (it != v.end())
            cout << "Muster gefunden \n";
      else
            cout << "Muster nicht gefunden \n";

}
```

Aufgaben zu Such-Algorithmen

Aufgabe 8.5 Primzahlen bestimmen

Schreiben Sie einen Algorithmus, der aus einer Menge von Zahlen (Set-Container) alle Primzahlen bestimmt.

Definieren Sie ein geeignetes 1-stelliges Prädikat und benutzen Sie find_if.

Aufgabe 8.6 Sequenzen bestimmen

Schreiben Sie einen Algorithmus, der in einer Liste von Zahlen die erste Sequenz mit mehreren gleichen aufeinander folgenden Elementen bestimmt. Gesucht ist die Anfangs- und Endposition der Sequenz.

Benutzen Sie dazu adjacent_find.

Sortier-Algorithmen

sort(first,last); sort(first,last,comp);	Sortiert den Bereich [first,last). Vergleichsoperator ist <. Es kann aber auch ein eigenes 2-stelliges Prädikat comp benutzt werden. Aufwand ist O(n log n).

stable_sort(first,last); stable_sort(first,last,comp);	Stabiles Sortierverfahren. Reihenfolge gleicher Elemente bleibt erhalten. Aufwand ist O(n log n).

…	

Modifizierende Algorithmen

- Reihenfolge der Elemente ändern:

reverse(first,last);	Kehrt die Reihenfolge der Elemente in [first,last) um.
random_shuffle(first,last);	Ordnet die Reihenfolge zufällig um. Nur bei Containers mit wahlfreiem Iteratorzugriff
next_permutation(first,last);	Erzeugt die nächste Permutation (Umordnung)
…	

- Elemente ändern:

replace(first,last,x,r);	Ersetzt in [first,last) alle Elemente mit Wert x durch r.
replace_if(first,last,pred,r);	Ersetzt alle Elemente, für die pred(x) gilt, durch r.
for_each(first,last,func);	Wendet auf alle Elemente die 1-stellige Funktion func an. Sollen die Container-Elemente geändert werden, dann ist für f ein Referenz-Parameter vorzusehen (siehe Beispiel)
…	

- Elemente löschen:

remove(first,last,x);	Löscht alle Elemente aus [first,last) mit Wert x.
remove_if(first,last,pred);	Löscht alle Elemente, für die pred(x) gilt.
unique(first,last);	Folge von gleichen Elementen werden zu einem Element reduziert.
…	

Beispiel mit modifizierenden Algorithmen

```
#include <algorithm>
#include <list>
#include <iostream>

void f(int& x) {x *= x;}

bool isEven(int x) {return x%2==0;}

int main()
{
    // Liste füllen:
    const int size = 7;
    int a[size] = {1,3,3,4,5,6,4};
    list<int> v(a,a+size);

    // f auf jedes Element anwenden:
    for_each(v.begin(),v.end(),f);

    // jedes gerade Element löschen:
    remove_if(v.begin(),v.end(), isEven);

    // Gleiche Elemente zu einem
    // reduzieren:
    unique(v.begin(),v.end());

    // Reihenfolge umkehren:
    reverse(u.begin(),u.end());
}
```

v = 1,3,3,4,5,6,4

v = 1,9,9,16,25,36,16

v = 1,9,9,25

v = 1,9,25

v = 25,9,1

Kopier-Algorithmen (1)

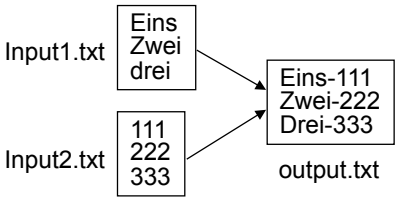
- Die Kopier-Algorithmen kopieren einen Container-Bereich (Quelle) in einen anderen Container-Bereich (Ziel). Dabei können Modifikationen vorgenommen werden.
- Für den Zielbereich muss Speicherplatz vorhanden sein. Gegebenenfalls Einfüge-Iterator verwenden.
- Die meisten modifizierenden Algorithmen gibt es auch als Kopiervarianten.

copy(first1,last1,first2);	Kopiert den Bereich [first1,last1) in den Bereich ab first2.
replace_copy(first1,last1,first2,x,r);	Beim Kopieren wird x durch r ersetzt.
replace_copy_if(first1,last1,first2,pred,r);	Beim Kopieren werden Elemente, für die pred(x) gilt, durch r ersetzt.
remove_copy(first1,last1,first2,x);	Beim Kopieren werden alle Elemente mit Wert x gelöscht.
remove_copy_if(first1,last1,first2,pred);	Beim Kopieren werden alle Elemente, für die pred(x) gilt, gelöscht.
unique_copy(first1,last1,first2);	Beim Kopieren werden Folge von gleichen Elementen zu einem Element reduziert.

Kopier-Algorithmen (2)

transform(first1,last1,first2,fun);	Kopiert den Bereich [first1,last1) in den Bereich ab first2 und ersetzt dabei jedes Element x durch fun(x).
transform(first1,last1,first2,first3,fun2);	Läuft synchron über alle Elemente x aus dem Bereich [first1,last1) und alle Elemente y aus dem Bereich ab first2 und schreibt fun2(x,y) in den Bereich ab der Position first3.

```
#include <algorithm>
#include <string>
#include <iostream>
#include <fstream>

string f(string x, string y) {
      return x + "-" + y;
}
```



Beispiel

```
int main() {
      // Eingabeströme:
      ifstream fin1("input1.txt");
      ifstream fin2("input2.txt");
      istream_iterator<string> inIt1(fin1), inIt2(fin2), end;

      // Ausgabestrom fout:
      ofstream fout("output.txt");
      ostream_iterator<string> outIt(fout,"\n");

      transform(inIt1,end,inIt2,outIt,f);
}
```

Mengen-Operationen (1)

b = includes(first1,last1,first2,last2);	Prüft, ob Bereich [first1,last1) in Bereich [first2,last2) enthalten ist
set_union(first1,last1, first2,last2, res);	Vereinigung
set_intersection(first1,last1, first2,last2, res);	Schnitt
set_difference(first1,last1, first2,last2, res);	Differenz; Bereich1 \ Bereich2
set_symmetric_difference(first1,last1, first2,last2, res);	Symmetrische Differenz: (A \ B) ∪ (B \ A)

Mengen-Operationen (2)

- Beachte: Die Mengenoperationen dürfen nur auf sortierte Container (d.h. sortierte sequentielle oder assoziative Container) angewandt werden.
- Beachte: Für die Ergebnismenge muss Speicherplatz vorhanden sein. Gegebenenfalls Einfüge-Iterator verwenden.
  (Siehe auch Kopier-Algorithmen.)
- Multisets: Die Semantik der Mengenoperationen ist in natürlicher Weise auf Multisets (Elemente dürfen mehrfach vorkommen) erweitert worden:
  - Vereinigung:
    Anzahl von x in A ∪ B
    = Maximum von Anz. von x in A und Anzahl von x in B
  - Schnitt:
    Anzahl von x in A ∩ B
    = Minimum von Anz. von x in A und Anzahl von x in B Die Anzahl
  - …
- Map, Multimap: analog zu Set bzw. Multiset, wobei Elemente Key-Value-Paare sind

Beispiel für Mengen-Operationen (1)

```
#include <algorithm>
#include <iostream>

int main()
{
      const int size1 = 5;
      const int size2 = 3;
      int x1[size1] = {2,3,4,5,6};
      int x2[size2] = {1,2,3};

      ostream_iterator<int> oit(cout,", ");

      set_union(x1,x1+size1, x2,x2+size2, oit);          cout << endl;
      set_intersection(x1,x1+size1,x2,x2+size2, oit);    cout << endl;
      set_difference(x1,x1+size1,x2,x2+size2,oit);       cout << endl;
      set_symmetric_difference(x1,x1+size1,x2,x2+size2,oit);
      cout << endl;
}
```

```
1, 2, 3, 4, 5, 6,
2, 3,
4, 5, 6,
1, 4, 5, 6,
```


Beispiel für Mengen-Operationen (2)

```
#include <algorithm>
#include <set>
#include <iostream>

int main() {
      int x1[6] = {3,2,3,5,3,4};
      int x2[6] = {6,3,2,3,1,6};

      multiset<int> m1(x1,x1+6);
      multiset<int> m2(x2,x2+6);
      // m1 und m2 ausgeben ...
      multiset<int> m;
      insert_iterator< multiset<int> >   insIt(m, m.begin() );

      set_union(m1.begin(),m1.end(),m2.begin(),m2.end(), insIt );
      // m ausgeben und m.clear(); ...
      set_intersection(m1.begin(),m1.end(), m2.begin(),m2.end(),insIt);
      // m ausgeben und m.clear(); ...
      set_difference(m1.begin(),m1.end(), m2.begin(),m2.end(), insIt);
      // m ausgeben und m.clear(); ...
      set_symmetric_difference(m1.begin(),m1.end(), m2.begin(),m2.end(), insIt);
      // m ausgeben und m.clear(); ...
}
```

```
m1:                    2, 3, 3, 3, 4, 5,
m2:                    1, 2, 3, 3, 6, 6,

Union:                 1, 2, 3, 3, 3, 4, 5, 6, 6,
Intersection:          2, 3, 3,
Difference (m1\m2):    3, 4, 5,
Symmetric Diff.:       1, 3, 4, 5, 6, 6,
```

Aufgabe zu Mengen-Operationen

Aufgabe 8.7

Was leisten die Funktionen f und g?

```
#include <set>
#include <algorithm>

bool istPrim(int n)
{
      // prüft, ob n eine Primzahl ist
}

void f(int n, multiset<int>& s)
{
      int f = 2;

      s.clear();
      while (n > 1) {
            if (istPrim(f) && n%f == 0) {
                  s.insert(f);
                  n = n/f;
            }
            else
                  f++;
      }
}
```

```
int g(int n, int m)
{
      multiset<int> s1, s2, s;
      insert_iterator< multiset<int> > ins(s,s.begin());

      f(n,s1);
      f(m,s2),
      set_union(s1.begin(),s1.end(),s2.begin(),s2.end(),ins);

      int p = 1;
      for (multiset<int>::iterator it = s.begin(); it != s.end(); it++)
            p *= *it;

      return p;
}
```


Literaturverzeichnis (1)

- Hartmut Helmke u. Rolf Isernhagen, *Softwaretechnik in C und C++: Modulare, objektorientierte und generische Programmierung*, Hanser-Verlag, 2001.
- N. Josuttis, *Die C++-Standarbibliothek*, Addison-Wesley, 1996.
- Ulrich Breymann, Designing Components with the C++-STL: A New Approach to Programming, Addison-Wesley, 1998
- Timothy Budd, Data Structures in C++ using the STL, Addision-Wesley, 1998.
- ISO/IEC-14882: International Standard for the C++ Programming Language, 1998.

Literaturverzeichnis (2)

Vollständige und präzise Informationen lassen sich am besten im ISO-Standard nachschlagen. Z.B. remove:

25.2.7 Remove **[lib.alg.remove]**

```
template<class ForwardIterator, class T>
  ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                         const T& value);

template<class ForwardIterator, class Predicate>
  ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                            Predicate pred);
```

1 **Requires:** Type `T` is `EqualityComparable` (20.1.1).

2 **Effects:** Eliminates all the elements referred to by iterator `i` in the range `[first, last)` for which the following corresponding conditions hold: `*i == value, pred(*i) != false`.

3 **Returns:** The end of the resulting range.

4 **Notes:** Stable: the relative order of the elements that are not removed is the same as their relative order in the original range.

5 **Complexity:** Exactly `last - first` applications of the corresponding predicate.

```
template<class InputIterator, class OutputIterator, class T>
  OutputIterator
    remove_copy(InputIterator first, InputIterator last,
                OutputIterator result, const T& value);

template<class InputIterator, class OutputIterator, class Predicate>
  OutputIterator
    remove_copy_if(InputIterator first, InputIterator last,
                   OutputIterator result, Predicate pred);
```

551