Praxisorientierte Einführung in C++
Lektion: "STL-Container"

Christof Elbrechter

Neuroinformatics Group, CITEC

June 26, 2014

Table of Contents

Containertypen

- STL bietet eine Vielzahl von Containern als Template-Klassen
  - `std::vector<T, Alloc>`
  - `std::list<T, Alloc>`
  - `std::slist<T, Alloc>`
  - `std::set<Key, Compare, Alloc>`
  - `std::map<Key, Data, Compare, Alloc>`
  - `std::deque<T, Alloc>`
  - ...
- ... und Operationen auf diesen Containern

Containertypen

- Wir konzentrieren uns erst einmal auf:
  - `std::vector<T, Alloc>`
  - `std::list<T, Alloc>`
  - `std::map<Key, Data, Compare, Alloc>`
  - `std::set<Key, Compare, Alloc>`

std::vector

- Wie alle STL-Typen: komplett `inline` deklariert
- Deklaration im Header `<vector>`
- Array-Typ für Elemente vom Typ `T`
- Erlaubt effizienten, O(1), Random-Access-Zugriff auf einzelne Elemente
- Größenänderungen und Einfüge-Operationen (relativ) ineffizient
- Templateparameter `Alloc` interessiert uns erstmal nicht: Standardwert ausreichend[1]

[1] Zum Thema *Allocatoren*: Später mehr

vector

Konstruktoren (vector)

```
// Leerer vector (Groesse 0)
vector();

// Vector der Groesse n
explicit vector(size_type n, const T&t=T());

// Copy-Construktor
vector(const vector&);

// Iteratorbasierter Konstruktor
template<class Iterator>
vector(Iterator begin, Iterator end);
```

Beispiel

```
#include <vector>
#include <string>

int main(int n, char **ppc) {
    // Leerer vector von int
    std::vector<int> v1;

    // String-vector der Groesse 10,
    // Initialisiert mit ""
    std::vector<std::string> v2(10);

    // Initialisiert mit "!"
    std::vector<std::string> v3(10, "!");

    // Iterator-basierte Konstruktion (Iteratoren: gleich)
    std::vector<std::string> v4(ppc+1,ppc+n);
}
```

Elementzugriff

- Zugriff auf Elemente mit Index-Operator

```
T &operator[](size_type)
const T &operator[](size_type) const
```

- size_type ist typedef innerhalb der std::vector Definition
- Verwendung: std::vector<std::string>::size_type i;
- Ist meistens einfach unsigned int oder ähnliches
- Wir benutzen ab jetzt einfach unsigned int damit der Code besser auf Folien passt

Beispiel Elementzugriff

```
std::vector<std::string> v(10);
v[0] = "Hallo"; v[1] = "Welt";
for (unsigned int i = 0; i < 10; ++i)  std::cout << v[i] << std::endl;;
```

Elementzugriff

- Der Index-Operator macht per Definition keinen Index-Check
  $\Rightarrow$ daher ist dieser effizient
- Falls Index-Check erwünscht ist: Methode `T &vector<T>::at(size_type)` (auch `const`) verwenden

Beispiel: Elementzugriff mit at()

```
std::vector<std::string> v(10);
v.at(0) = "Hallo"; v.at(1) = "Welt";
for (unsigned int i = 0; i < 12; ++i) std::cout << v.at(i) << std::endl;
```

- `at()` hat den Vorteil, dass Index gecheckt wird
- Wenn zu gross (oder zu klein – was nur möglich ist, falls `size_type` nicht `unsigned` ist), wird `std::out_of_range` Exception geworfen
- **Nachteil:** Durch Check ist Zugriff sehr viel langsamer als `operator[]`
- $\Rightarrow$ Nur verwenden falls es absolut notwendig ist!

Weitere Features von std::vector

- Größe ist zur Laufzeit veränderbar
- Funktionen z.B.:
  - `resize(size_type newSize, const T &init);`
  - `push_back(const T&val);`
  - `insert(...)`
- Großer Vorteil gegenüber konstanten oder dynamisch verwalteten Arrays
- Aber: Ineffizient im Vergleich zu Listen (evtl. müssen Elemente kopiert werden)
- Bedeutet auch: `T` muss kopierbar und zuweisbar sein

Aber Achtung!

In C++ ist bzgl. Komplexität von Algorithmen oft der konstante Faktor entscheidend

Größenänderung

std::vector - Größe verändern

```
// Leerer Vector
std::vector<std::string> v;

// Groesse aendern auf 10
v.resize(10, "ein String");

// Groesse aendern auf 5
v.resize(5, "ein Bling");

// Da nur Elemente verworfen wurden,
// steht immer noch "ein String" in den
// restlichen Elementen von v
std::cout << v[4] << std::endl;
```

push_back

Hinzufügen von Elementen

```
int main(int argc, char *argv[]) {
    std::vector<std::string> v;
    for (unsigned int i = 0; i < argc; ++i){
        v.push_back(argv[i]);
    }
}
```

- Da Größe im letzten Beispiel bekannt: `resize()` und Zuweisung wäre effizienter

size()

- Die aktuelle Größe einer vector-Instanz kann mittels size() abgefragt werden

Beispiel

```
int main(int argc, char *argv[]) {
    std::vector<std::string> > v;
    for (unsigned int i = 0; i < argc; ++i) {
        v.push_back(argv[i]);
    }
    for (unsigned int i = 0; i < v.size(); ++i) {
        std::cout << v[i] << std::endl;
    }
}
```

Interne Repräsentation (Qualitativ)

- Intern verwaltet der `std::vector` immer einen `begin_data`-Pointer
- Dazu kommt ein Pointer: `end_contained_data`, welcher hinter das z.Z. letzte enthaltene Element zeigt
- .. und ein Pointer: `end_allocated_data`, welcher hinter den z.Z. allozierten Speicherbereich zeigt

Interne Repräsentation (Qualitativ)

- Intern verwaltet der `std::vector` immer einen `begin_data`-Pointer
- Dazu kommt ein Pointer: `end_contained_data`, welcher hinter das z.Z. letzte enthaltene Element zeigt
- .. und ein Pointer: `end_allocated_data`, welcher hinter den z.Z. allozierten Speicherbereich zeigt
- D.h. es gibt immer zwei wichtige Größenangaben:
  - `vector::size()` = `end_contained_data-begin_data`
  - `vector::capacity()` = `end_allocated_data-begin_data`
- `resize` passt beides an
- Wenn nur die *Kapazität* des Vectors verändert werden soll, muss `reserve(size_t)` aufgerufen werden

Standardverhalten des Vectors

- Beim Einfügen (mittels `insert`) oder Anfügen (mittels `push_back`) kann der Fall auftreten, dass `capacity()` zu gering ist
- In diesem Fall wird die aktuelle Kapazität des Vectors verdoppelt

```
template<class T, class Alloc>
void vector<T,Alloc>::push_back(const T&t){
     if(end_contained_data == end_allocated_data){
         reserve_and_copy_old_data(2*size());
     }
     *end_contained_data++ = t;
}
```

- Anmerkung: Es gibt eine Version von `vector::insert`, in der mehrere Elemente auf einmal eingefügt werden können
- In diesem Fall könnte ein mehrfaches Verdoppeln der Größe vermieden werden

reserve

- Wird mehr Speicher alloziert, so muss:
  - neuer Speicher angelegt werden
  - der Inhalt des alten Speichers in den Neuen kopiert werden
  - der alte Speicher freigegeben werden
- Dieses kann mit einem einfachen Versuch gezeigt werden

vector-Verhalten bei push_back

Test zum Protokollieren des Verhaltens:

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

#define log(x) std::cout << #x
struct A{
  A(){ log(N); }                    // New
  A(const A &){ log(C); }           // Copy
  A&operator=(const A&){ log(A);} // Assignment
  ~A() { log(D); }                  // Destructor
};

int main(int n, char **ppc){
  std::vector<A> v;
  const A a;
  std::cout << std::endl;
  for(int i=0;i<10;++i){
    std::cout << " " << i << ":";
    v.push_back(a);
    std::cout << std::endl;
  }
  std::cout << std::endl;
}
```

Ergebnis des Versuchs:

```
N
 0:C
 1:CCD
 2:CCCDD
 3:C
 4:CCCCCDDDD
 5:C
 6:C
 7:C
 8:CCCCCCCCCDDDDDDDDD
 9:C

DDDDDDDDDDD
```

Fazit

- Falls möglich: Vector gleich mit richtiger Größe erstellen
- Ansonsten: falls möglich zu einem späteren Zeitpunkt Größe mit `resize` anpassen und Index-Operator für Zugriff verwenden
- Wenn auch das nicht möglich ist: Finale Größe des Vectors *konservativ* abschätzen und mit `reserve` die Anzal der Reallozierungen von Speicher verringern
- **Aber Achtung:** Profile first – Optimize later!

Weitere Anmerkungen zum std::vector

- `std::vector` garantiert, dass Elemente kontinuierlich im Speicher liegen, wie bei normalem Array
- Zeiger auf erstes Element: `&v[0]`
- Alternativ: `&v.front()` (letztes Element: `back()`)
- Nicht im Standard: `data()`
- Zeigerarithmetik funktioniert wie gewohnt

`std::vector` - Zeigerarithmetik

```
std::vector<int> v(100,0);
int *begin = &v[0];
int *end = &v.front() + 100;
for (int *i = begin; i != end; ++i) {
   std::cout << *i << std::end;
}
```

Weitere Anmerkungen zum std::vector

- Instanzen von `std::vector<T>` können auch miteinander verglichen werden
- 2 Vectoren sind gleich, falls:
  - 1.) Ihre Länge gleich ist
  - 2.) Jedes einzelne Element gleich ist
- Alle STL Container werden tief (elementweise) kopiert
- Es gibt auch noch `operator<` & Co
  - Lexikographischer Vergleich (Vergleich: elementweise)
- `assign(iterator start, iterator end)` weist dem `vector` einen neuen Inhalt zu (alter Inhalt geht verloren)
- `assign(size_type num, const T &val)` füllt den `vector` mit `num` Kopien von `val` (alter Inhalt geht verloren)

Container die polymorphe Typen enthalten

- Was ist mit Zeigern, Referenzen und Polymorphismus?
- Referenzen können nicht in `std::vector` abgelegt werden
- Zeiger dagegen sind kopier- und zuweisbar, daher können sie in `std::vector` abgelegt werden
- Verantwortung für Speichermanagement beim Benutzer
- Lösung: Smartpointer
- Polymorphismus: Zeiger oder Smartpointer

std::list

- Header `<list>`
- Implementiert doppelt verkettete Liste
- Effizientes Einfügen von Elementen
  - ...am Anfang
  - ...am Ende
  - ...in der Mitte
- Aber: Kein effizienter Random-Access $\Rightarrow$ Daher wird kein Index-Operator angeboten!

Konstruktoren std::list

```
// Leere Liste (Groesse 0)
list();

// Liste der Groesse n gefuellt mit Kopien von t
list(size_type n, const T& t=T());

// Copy-Konstruktor
list(const list&);

// Iterator-basierter Konstruktor
template<class Iterator>
list(Iterator begin, Iterator end);
```

Eigenschaften: std::list

- Bietet Methoden, um Elemente in O(1) vorne oder hinten anzuhängen

```
void push_front(const T&)
void push_back(const T&)
```

- .. und zu entfernen

```
void pop_front()
void pop_back()
```

Beispiel

```
std::list<int> l;

l.push_back(42); l.push_back(23); l.push_front(42);

l.pop_front(); l.pop_back(); l.pop_front();
// l.size() ist wieder 0
```

Weitere Merkmale von std::list

- `std::list` verhält sich zum großen Teil wie `std::vector`
- Allerdings: `std::list` hat nicht `reserve` und `capacity` (klar!)
- Aber: `front()` und `back()` gibt's auch
- Auch `size()` und sogar `resize(..)` wird unterstützt.

Weitere Merkmale von std::list

- `std::list` verhält sich zum großen Teil wie `std::vector`
- Allerdings: `std::list` hat nicht `reserve` und `capacity` (klar!)
- Aber: `front()` und `back()` gibt's auch
- Auch `size()` und sogar `resize(..)` wird unterstützt.
- **Große Frage:** Wie kommt man an die Elemente in der Mitte?

Weitere Merkmale von std::list

- `std::list` verhält sich zum großen Teil wie `std::vector`
- Allerdings: `std::list` hat nicht `reserve` und `capacity` (klar!)
- Aber: `front()` und `back()` gibt's auch
- Auch `size()` und sogar `resize(..)` wird unterstützt.
- **Große Frage:** Wie kommt man an die Elemente in der Mitte?
- **Einfache Antwort:** Mithilfe von sog. *Iteratoren*

Exkurs: Iteratoren

- Iteratoren stellen ein grundsätzliches Konzept der STL dar
- Grundsätzlich kann eine Menge von Daten immer irgendwie linear durchlaufen werden
- D.h.: Ein Container beschreibt eine *Range* von Datenelementen zwischen `begin()` und `end()`
- Einfachstes Beispiel: C++-Array
  - z.B.:`int array[10];`
  - `begin()` ist der Zeiger auf das erste Element (also `array` selbst)
  - `end()` ist der Zeiger hinter das letzte Element
  - Schritt zwischen den Elementen ist `sizeof(int)`

Wichtig

STL-Ranges: entsprechen immer halboffenen Intervallen `[begin(),end()[` bzw. `[begin(),end())`

STL: Iteratoren

- Alle STL-Datentypen bieten Iterator-basierten Zugriff mittels Methoden `begin()` und `end()`
- Dadurch: Entwicklung von generischen Templates möglich

Was ist ein Iterator?

Ein Iterator ist etwas, das sich wie ein normaler Pointer verhält: Er benötigt mindestens:

- einen Inkrement Operator `operator++(int)`
- einen De-Referenz Operator `operator*`
- den Pointer-Zugriffs-Operator `operator->`

STL: Iterator Hierarchie

- Iteratoren unterscheiden sich in zwei Merkmalen
- Wie kann auf die Daten zugegriffen werden?
  - nur lesend $\Rightarrow$ Es handelt sich um einen sog. *Input*-Iterator
  - lesend und schreibend $\Rightarrow$ Es handelt sich um einen sog. *Output*-Iterator

STL: Iterator Hierarchie

- Iteratoren unterscheiden sich in zwei Merkmalen
- Wie kann auf die Daten zugegriffen werden?
  - nur lesend ⇒ Es handelt sich um einen sog. *Input*-Iterator
  - lesend und schreibend ⇒ Es handelt sich um einen sog. *Output*-Iterator
- Wie flexibel kann der Iterator bewegt werden?
  - nur immer um einen Schritt vorwärts ⇒ *Forward*-Iterator (`slist`)
  - vorwärts und rückwärts (++ und --) ⇒ *Bidirectional*-Iterator (`list`, `map`, `set`, `multimap` und `multiset`)
  - Beliebig (z.B.: `*(it+10)`) ⇒ *Random-access*-Iterator (`vector`, `array`, `deque` und `string`)

Verschiedene Iteratoren

- Jeder STL-Datentyp bringt seine eigenen Iteratoren mit sich
- Typen immer als Klassen-interne `typedef`s:
  - `iterator`: Normaler Iterator
  - `const_iterator`: Normal, aber nur Lesezugriff (*Input-Iterator*)
  - `reverse_iterator`: Rückwärts-laufender Iterator (++ macht -- und andersrum)
  - `const_reverse_iterator`: Wie oben, aber als *Input*-Iterator

Wichtig!

- `const iterator` und `const_iterator` sind i.d.R. unterschiedliche Typen
- `const iterator` darf nicht verschoben werden
- beim `const_iterator` sind die dahinterliegenden Daten `const`

Eigene Iterator-Implementationen:

- Für eigene Datentypen können auch eigene Iterator-Typen implementiert werden

Dabei sollte:

- .. darauf geachtet werden, dass alle Operationen *billig* sind
- .. `std::iterator` (im Header `<iterator>`) beerbt werden
  Hier werden wichtige Typedefs vorgenommen

Beispiel

```
class MyIterator : public std::iterator<std::forward_iterator_tag, T> { ... };
```

- .. Alle notwendigen Methoden und Operatoren für den entsprechenden Iterator-Typen implementiert werden[2]

[2] siehe Internet!

Anmerkungen

- Nochmal: Iteratoren sind grundliegendes Konzept der STL
- Die STL bietet eine Vielzahl von Algorithmen (wie z.B. `sort` und `for_each`) für das Iterator-Konzept an
- Das Konzept ist sehr mächtig und es ermöglicht viele sowohl elegante als auch effiziente Lösungen

Beispiel: Ausgabe der Kommandozeilenargumente

```
#include <iostream>
#include <iterator>
#include <algorithm>
int main(int n, char **args){
    std::copy(args+1,args+n,
              std::ostream_iterator<std::string>(std::cout,"\n"));
}
```

- Kann man ein Iterator-basierten Zugang anbieten, so kann man auch all diese Features direkt nutzen!

Zurück zur std::list

- `std::list` stellt Iteratortypen zur Verfügung: `std::list<T>::iterator` und `std::list<T>::const_iterator`
- Außerdem Methoden, um Iteratoren zu erstellen, die auf den Anfang/das Ende zeigen: `std::list<T>::iterator begin()` und `std::list<T>::iterator end()`

Anwendungsbeispiel:

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

int main() {
   std::list<int> l;
   l.push_back(1);
   l.push_back(2);

   for(std::list<int>::iterator it = l.begin();
      it != l.end(); ++it){
         std::cout << (*it) << std::endl;
   }
}
```

Weitere Iterator-Eigenschaften

std::list - Zugriff über (Zeiger-)Operatoren

```
int main() {
   std::list<Point2D> l;
   l.push_back(Point2D(0,0));
   l.push_back(Point2D(0,1));
   for(std::list<Point2D>::iterator it = l.begin();
      it != l.end(); ++it) {
      std::cout << (*it).x << " " << it->y << std::endl;
   }
}
```

Weitere Iterator-Eigenschaften

std::list - Zugriff über (Zeiger-)Operatoren

```
int main() {
   std::list<Point2D> l;
   l.push_back(Point2D(0,0));
   l.push_back(Point2D(0,1));
   for(std::list<Point2D>::iterator it = l.begin();
      it != l.end(); ++it) {
      std::cout << (*it).x << " " << it->y << std::endl;
   }
}
```

Reihenfolge umkehren

```
#include <list>
#include <iostream>
#include <iterator>
int main(){
    int is[] = {1,2,3,4,5};
    std::list<int> l(is,is+5),l2(5);
    std::copy(l.rbegin(),l.rend(),l2.begin());
    std::copy(l2.begin(),l2.end(),
              std::ostream_iterator<int>(std::cout," "));
}
```

Sortieren einer std::list

- Header <algorithm> enthält generische Funktionstemplates
- u.A.: `std::sort(begin,end)` zum Sortieren einer *Range*
- `std::sort` benötigt allerdings random-access-Iterator
- `std::list` liefert aber nur bidirectional-Iterator

Sortieren einer Liste

```
#include <list>
#include <vector>
#include <iostream>
#include <iterator>
#include <algorithm>
int main(){
    int is[] = {5,2,3,1,4};
    std::list<int> l(is,is+5);
    std::vector<int> v(l.begin(),l.end());
    std::sort(v.begin(),v.end());
    std::copy(v.begin(),v.end(),l.begin());
    std::copy(l.begin(),l.end(),
              std::ostream_iterator<int>(std::cout," "));
}
```

- Viel einfacher: Methode `list<T>::sort()`

Random-Access-Iteratoren (std::vector)

- `std::vector<T>::iterator` hat zusätzlich (da Random-Access-Zugriff effizient implementierbar)
  - `operator+(size_type)`
  - `operator+=(size_type)`
  - `operator-(size_type)`
  - `operator-(const std::vector<T,Alloc>::iterator&)`
  - usw.

Verwendungsbeispiel:

```
std::vector<Point2D> v(10, Point2D(0,0));

std::cout << (v.begin() + 5)->x << std::endl;
std::cout << (*(v.begin() + 5)).x << std::endl;
std::cout << (v.end() - 3)->x << std::endl;
```

Anmerkungen zu std::list-Iteratoren

- Bei `std::list`: Distanz nicht effizient implementierbar
- Im schlimmsten Fall muss zur Berechnung die gesamte Liste durchsucht werden
- Es gibt aber `std::distance` in `<iterator>`
- Keine `+`, `-`, etc..
- Nur `++`/`--`, `begin()`/`end()`
- Daher auch kein `operator[]`

C++-11 ...

- In C++-11 gibt es sog. range-based for loops

Beispiel

```
std::vector<float> data(10);
for(float &f : data){   // range based for
   f = 10;
}
```

- das geht für alles, was begin() und end() implementiert, und für Fixed-size Arrays

Performance Vergleich std::list vs. std::vector

Grundlage:

```
#include <algorithm>
#include <vector>
#include <list>
template<class T, class Container>
void test(){
  static Container v;
  v.clear();
  for(int i=0;i<1000000;++i){
    v.push_back(T());
  }
}
int main(){
  test<int,std::vector<int> >();
  test<int,std::list<int> >();
}
```

Performance Vergleich `std::list` vs. `std::vector`

Grundlage:

```
#include <algorithm>
#include <vector>
#include <list>
template<class T, class Container>
void test(){
  static Container v;
  v.clear();
  for(int i=0;i<1000000;++i){
    v.push_back(T());
  }
}
int main(){
  test<int,std::vector<int> >();
  test<int,std::list<int> >();
}
```

- Ergebnis:
  - `std::vector`: 3.7ms (-O4 -march=native), 16.5ms (-O0)
  - `std::list`: 228ms (-O4 -march=native), 268ms (-O0)

Erklärung

- Pointer-Handling in der `std::list` ist sehr aufwendig!
- Wann lohnt sich die `std::list` überhaupt?
- 2.Versuch: *Nehme komplexere Datentypen*

```
struct A {
   int x[100];
};
struct B {
   int x[2000];
};
```

- Ergebnis mit `T = A`:
  - `std::vector`: 320ms (-O4 -march=native), 324 (-O0)
  - `std::list`: 831ms (-O4 -march=native), 990 (-O0)
- Ergebnis mit `T = B`:[3]
  - `std::vector`: 6.7s (-O4 -march=native)
  - `std::list`: 6.4s (-O4 -march=native)

[3] Extrapoliert von Container-Endgröße 100000

Fazit

- Bessere Komplexität der `std::list` bringt erst bei relativ großen Einzelelementen Vorteile
- Für Standarddatentypen ist der `std::vector` fast 2 Größenordnungen schneller
- **Aber:** Kopie der oben verwendeten Klassen (`A` und `B`) ist sehr simpel (built-in `memcpy`)
- Bei Komplexeren Copy-Konstruktoren schneidet `std::vector` vermutlich schlechter ab
- Und zeitweise belegt der `std::vector` 2x so viel Speicher wie nötig

Anmerkung `slist`

`slist`

- Die einfach verkettete Liste `std::slist` wird nicht mehr richtig gepflegt
- Header: `<ext/slist>` ist nicht mehr im Standard
- `slist` ist im Namensraum `__gnu_cxx`
- Nebenbei: `slist` war noch deutlich langsamer als `std::list`

Assoziative Container: std::map

- Assoziatives Mapping:
- `std::map<Key,T, Compare, Alloc>`
- Im Header `<map>`
- Leider keine `std::hashmap` in der STL

Beispiel: Verwendung einer `std::map`

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

int main(int argc, char *argv[]) {
   std::map<std::string, int> m;

   m["Flo"] = 1000;
   m["Christof"] = 7;
}
```

Anmerkungen

- Bei Aufruf des Index-Operators (`T &operator[](const Key &)`) wird ein leerer Eintrag erzeugt falls notwendig
- Grund: es muss immer eine gültige Referenz zurückgegeben werden
- Auswirkung: Es existiert keine `const`-Version des Index Operators

Beispiel

```
std::map<std::string, std::string> leisureTime;
// noch nicht existierende Eintraege
// werden on-demand erzeugt (mit dem Allocator)
std::cout << "Marvin hat "
          << leisureTime["Marvin"]
          << " Zeit"  << std::endl;

// Ausgabe: "Marvin hat  Zeit"
```

std::map - Iteratoren - std::pair

- Iteratoren zeigen auf std::pair<Key,Value>
- std::pair definiert im Header <utility> (wird automatisch von <map> mit eingebunden)
- std::pair hat Elemente first und second

Beispiel std::pair

```
#include <utility>
#include <iostream>

int main() {
    std::pair<int, float> p(10, 0.1);

    std::cout << "first: " << p.first
              << " second: " << p.second << std::endl;
}
```

std::map - Iteratoren - std::pair

- Iteratoren zeigen auf std::pair<Key,Value>
- $\Rightarrow$ Iteration ist etwas komplizierter

std::map - Iteratoren

```
#include <map>
#include <iostream>

int main(){
  // maps koennen auch sequentiell durchlaufen werden
  typedef std::map<std::string,std::string> smap;
  smap leisureTime;
  leisureTime["me"]="alot of";
  leisureTime["flo"]="almost no";

  for(smap::iterator it = leisureTime.begin();
      it != leisureTime.end(); ++it){
    std::cout  << it->first << " has "
               << it->second
               << " leisure time" << std::endl;
  }
}
```

std::map contains(Key?)

- es gibt keine Methode namens `bool std::map<T,Key>::contains(const Key &)`
- dafür gibt es die Methode `iterator std::map<T,Key>::find(const Key &key)`
  - gibt typspezifischen Iterator zurück
  - .. oder `std::map<T,Key>::end()` falls key nicht enthalten ist

Ein komplexeres Beispiel

Beispiel

```
#include <map>
#include <iostream>
#include <stdexcept>

typedef std::runtime_error err;

struct MySMap : public std::map<std::string,std::string>{
  bool contains(const std::string &key) const{
    return find(key) != end();
  }
  std::string &operator[](const std::string &key) throw (err){
    iterator it = find(key);
    if(it == end()) throw err("MySMap::operator[] invalid key: " + key);
    return it->second;
  }
  const std::string &operator[](const std::string &key) const throw (err){
    return const_cast<MySMap&>(*this)[key];
  }
  void add(const std::string &key, const std::string &value){
    std::map<std::string,std::string>::operator[](key)=value;
  }
};
```

Fortsetzung

Beispiel (Fortsetzung)

```
int main(){
  MySMap m;
  m.add("baum","tree");
  m.add("auto","car");

  std::cout << m["baum"] << std::endl;
  std::cout << m["auto"] << std::endl;

  m["hund"] = "chicken";
}
```

Ausgabe

tree

car

terminate called after throwing an instance of 'std::runtime_error'

what(): MySMap::operator[] invalid key: hund

Weitere Anmerkungen zu std::map

- Key muss sortierbar sein (totale Ordnung)
- Dazu muss eine *Kleiner*-Operation für key definiert sein

Negativ-Beispiel

```
struct Point{
   Point(uchar x=0,uchar y=0):x(x),y(y){}
   uchar x,y;
};

typedef std::map<Point,std::string> citymap;

int main(){
   citymap cm;
   cm[Point(2,4)] = "Bielefeld";      // Fehler! kein
                                      // Operator '<' fuer
                                      // Point definiert
}
```

- Es muss ein `operator<` für `Point` definiert werden oder ...
- .. ein anderes `std::map`-Template verwendet werden

Lösungsstrategien

1.Lösung: operator< in die Klasse Point

```
#include <map>
#include <string>

typedef unsigned char uchar;

struct Point{
   Point(uchar x=0,uchar y=0):x(x),y(y){}
   uchar x,y;
   bool operator<(const Point &p) const{
      return x+256*y < p.x+256*p.y;
   }
};

typedef std::map<Point,std::string> citymap;

int main(){
   citymap cm;
   cm[Point(2,4)] = "Bielefeld";    // kein Fehler!
}
```

- Problem: Was ist wenn die Klasse nicht geändert werden kann?

Lösungsstrategien

2.Lösung: operator< außerhalb der Klasse Point

```
#include <map>
#include <string>

typedef unsigned char uchar;

struct Point{
   Point(uchar x=0,uchar y=0):x(x),y(y){}
   uchar x,y;
};

bool operator<(const Point &a, const Point &b){
  return a.x+256*a.y < b.x+256*b.y;
}

typedef std::map<Point,std::string> citymap;

int main(){
   citymap cm;
   cm[Point(2,4)] = "Bielefeld";    // kein Fehler!
}
```

- Problem: Was wenn der Operator schon anders definiert wurde?

Lösungsstrategien

- Dritter `std::map`-Template-Parameter: `Compare`
- `Compare` ist *per default* `std::less<Key>` was wiederum den `operator<` aufruft
- Durch Ersetzen von `std::less<Key>` durch anderen Typen kann das Verhalten angepasst werden
- Damit auch möglich: Maps mit gleichen Keys aber unterschiedlichen Sortierungen

Lösungsstrategien

3.Lösung: Anderes `std::map`-Template verwenden

```
#include <map>
#include <string>

typedef unsigned char uchar;

struct Point{
   Point(uchar x=0,uchar y=0):x(x),y(y){}
   uchar x,y;
};

struct PointCmp{ // Vergleichsfunktor
  bool operator()(const Point &a, const Point &b) const{
    return a.x+256*a.y < b.x+256*b.y;
  }
};

typedef std::map<Point,std::string,PointCmp> citymap;

int main(){
   citymap cm;
   cm[Point(2,4)] = "Bielefeld";    // kein Fehler!
}
```

multimap, set und deque

- Bei der `std::multimap` können Keys auch mehrfach vorhanden sein
- Damit ändert sich das Interface
- Der []-operator macht keinen Sinn mehr, da hier ja nicht angegeben werden kann, auf welchen Schlüssel man sich bezieht, falls dieser mehrfach enthalten ist
- Übrigens: Ebenfalls im Header <map> definiert!

```
// hier koennen Keys doppelt vorkommen
typedef std::multimap<int,const char*> agemap;

agemap am;

am.insert(std::pair<int,const char*>(60,"Al Gore"));
am.insert(std::make_pair(61,"Alice Cooper"));
am.insert(std::make_pair(60,"Niki Lauda"));
am.insert(std::make_pair(61,"Billy Crystal"));
am.insert(std::make_pair(61,"OJ Simpson"));
am.insert(std::make_pair(61,"Meat Loaf"));
am.insert(std::make_pair(61,"Sam Neill"));
```

Index-Access ⇒ Sub-*Ranges*

upper_bound und lower_bound

```
#include <map>
#include <iostream>
typedef std::multimap<int,const char*> agemap;

int main(){
  agemap am;

  am.insert(std::pair<int,const char*>(60,"Al Gore"));
  am.insert(std::make_pair(61,"Alice Cooper"));
  am.insert(std::make_pair(60,"Niki Lauda"));
  am.insert(std::make_pair(61,"Sam Neill"));
  am.insert(std::make_pair(64,"Franz Beckenbauer"));

  // alle ueber 60 und unter 62

  for(agemap::iterator it = am.upper_bound(60);
      it != am.lower_bound(62); ++it){
    std::cout  << it->second << " is JUST "
               << it->first << " years old"
               << std::endl;
  }
}
```

std::set und multiset

- std::set ist eine stets sortierte Menge an Elementen
- Dies entspricht einer std::map ohne Value
  - Neu-Deutsch: *Der Key ist quasi der Value*
- Analog gibt es auch noch std::multiset (in Anlehnung an std::multimap)

std::deque

- std::deque: Double ended queue
- Entspricht std::vector welche an beiden Seiten ein *offenes Ende* hat
- Konstanter Zugriff auf Elemente
- Geringer Overhead im Vergleich zum std::vector
- push_front analog zum push_back beim std::vector

Container-Adapter

- Container-Adapter setzen auf anderen Containern auf
  - Adapter: Front-End = Interface
  - anderer Container: Back-End = Internes handling
- STL Adapter:
  - `stack<T,Container=deque<T>>` (LIFO = last-in, first-out)
  - `priority_queue<T,Container=deque<T>,Compare=less<T>>` (Sortierte Queue)
  - `queue<T, Container=vector<T>>` Normale Queue (FIFO = first-in, first-out)
- Bieten simpleres Interface: `push()`, `pop()`, `top()`, `size()`, `empty()` und `clear()`

Inserter (Adapter mit Iterator-Verhalten)

Beispiel

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

int main(){
  int is[] = {1,2,3,4,5};
  std::deque<int> d;

  std::copy(is,is+5,std::back_inserter(d));
  std::copy(is,is+5,std::front_inserter(d));
  std::copy(is,is+5,std::inserter(d,d.begin()+4));

  std::copy(d.begin(),d.end(),std::ostream_iterator<int>(std::cout," "));

  //  Ausgabe: 5 4 3 2 1 2 3 4 5 1 1 2 3 4 5
}
```

- Beachte: z.B. `front_inserter` funktionieren nicht für `std::vector`

Allocatoren

- Alle STL-Container können mit speziellen Alloc-Template-Parameter definiert werden
- Dieser definiert, wie
  - Speicher für N neue Objekte alloziert wird
  - Speicher für N Objekte freigegeben wird
  - Ein Objekt an Speicheraddresse X konstruiert wird
  - Ein Objekt an Speicheraddresse X zerstört wird
  - Die maximale Anzahl an konstruierbaren Objekten
- **Anmerkung:** Das Definieren von Allocatoren ist *sehr sehr sehr* speziell (hier nur der Vollständigkeit halber)
- *Beispiel: `examples/alloc.cpp` im emacs da hier zu wenig Platz!*