
STL

Hier wird die C++ Standard Template Library vorgestellt¹.

Die Bibliothek von Templates wird nicht vollständig beschrieben – es werden die im praktischen Umfeld häufig verwendeten Bestandteile mit kleinen Programmierbeispielen demonstriert.

Wir werden in der Vorlesung auch Beispiele aus <http://www.cplusplus.com/reference/stl/> ansehen und diskutieren.

Inhaltsverzeichnis

1. Einleitung.....	2
2. Grundlagen.....	4
Container von Strings.....	8
3. Iteratoren der STL.....	9
4. Sequenzen.....	20
5. Assoziative Container.....	46
6. STL Algorithmen.....	55

1 Die Ausarbeitung basiert auf dem Buch „Thinking in C++“ von Bruce Eckel.

1. Einleitung

Warum muss jeder Programmieren Listen oder Vektoren immer wieder programmieren?

Die STL wurde von Hewlett-Packard entwickelt, um standardisierte Templates für immer wiederkehrende Problemfelder verfügbar zu haben.

Die STL Stellt dazu Datenstrukturen und Algorithmen zu Verfügung, die leicht portierbar sind und heute in allen C++ Umgebungen integriert sind.

Die STL basiert auf der Trennung zwischen Daten und Funktionen:

- **Container**
verwalten eine Menge von Objekten des gleichen Typs.
- **Iterator**
sind im Prinzip Zeiger auf Objekte in einem Container, mit denen man über die Objekte wandern (iterieren) kann. Iteratoren stellen für alle Containertypen das gleiche Interface zur Verfügung.
- **Algorithmen**
bearbeiten die Objekte in Containern (sortieren, finden, löschen ...)

Die Verwendung von STL Templates macht den Programmierer unabhängig von der darunter liegenden Datenstruktur und den implementierten Algorithmen. Der Mehraufwand beschränkt sich auf das Einarbeiten in die STL.

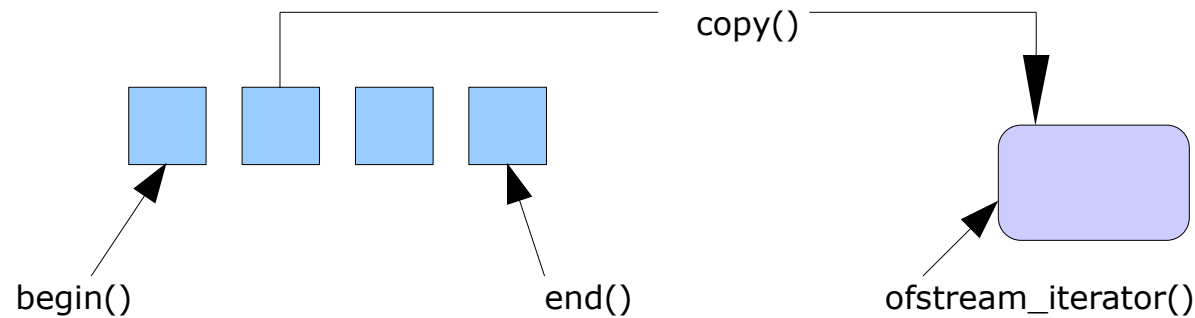
Alle drei Bestandteile sind im ersten Beispiel zusammen dargestellt: Ein Programm, in dem die Integer 0 bis 9 25 mal in eine Menge eingefügt werden und am Schluss ausgegeben werden.

```
$ cat Intset.cpp
// Simple use of STL set
#include <set>           // set...
#include <iostream>
#include <iterator>     // ostream_iterator
using namespace std;

int main() {
    set<int> intset;
    for(int i = 0; i < 25; i++)
        for(int j = 0; j < 10; j++)
            intset.insert(j);           // Try to insert multiple copies:
    copy(intset.begin(), intset.end(), // Print to output:
        ostream_iterator<int>(cout, "\n"));
}
$
```

Die Methode `insert()` fügt ein Element in die Menge (**Container**) `intset` ein, wenn es nicht schon vorher eingefügt wurde. Die Funktion `copy()` ist eine der STL Funktionen, die zu den STL **Algorithmen** gehört. `copy` verwendet drei Iteratoren: die `set`-Elementfunktionen `begin()` und `end()`, die jeweils einen **Iterator** zurück liefern und ein Iterator der Klasse `ostream` zur Ausgabe von Werten.

Welche Methoden kennt `<set>` noch – recherchieren Sie unter <http://www.cplusplus.com/reference/stl/> ?



`copy()` nimmt **jeden** Wert, zwischen `begin()` und `end()` und gibt ihn über den Iterator (3. Argument) mit Newline über `cout` aus. D.h. jeder Wert zwischen Anfang und Ende wird auf den 3. Iterator kopiert.

2. Grundlagen

Das Grundlegende Konzept der STL ist der **Container**. Alle Container der STL sind Speicherplatz für Objekte mit ähnlichen Zugriffsmechanismen, wobei jeder Container **selbst organisierend** ist, d.h. Speicher je nach Bedarf beschafft bzw. frei gibt. Unterschiede gibt es in der Art der Objekt-ablage, so sind z.B. Listen, Bäume und Schlangen realisiert.

Der **lesende Zugriff** kann bei einigen Containern **indiziert** erfolgen, der normale Weg ist aber die Verwendung von **Iteratoren**.

Zum schreibende Zugriff existieren Methoden, die vom Container abhängen. Im nächsten Beispiel wird ein Container verwendet, um eine Sequenz abzuspeichern.

```
$ cat Vector.cpp
#include <vector>
#include <iostream>
using namespace std;
int main() {
    typedef vector<int> vectorOfInt; // wg. Schreibvereinfachung
    vectorOfInt v;
    vectorOfInt::iterator it;

    for (int i=0; i<5; i++)           // Besetzen mit 0-4
        v.push_back(i);

    for (it=v.begin(); it != v.end(); ++it) // Ausgabe aller Elemente
        cout << *it << " ";           // Dereferenzieren
    cout << "\n";

    cout << "v[0]: " << v[0] << endl; // Zugriff ueber Index

    return 0;
}
$
```

Container beinhalten Objekte. In der OOP will man aber oft die Vorteile der späten Bindung (Polymorphismus) ausnutzen und erst zur Laufzeit entscheiden, auf welches Objekt bzw. welche Klasse zugegriffen werden soll. Dazu bieten es sich an, **Container von Zeigern** zu verwenden.

Das folgende Beispiel verwendet ebenfalls einen Vektor, diesmal einen **Vektor von Zeigern** auf Objekte der Klasse **Shape** (eine Vereinfachung unserer Klasse `GraphObj`).

```
$ cat Shape.cpp
// Simple shapes
#include <vector>
#include <iostream>
using namespace std;

class Shape {
public:
    virtual void draw() = 0;
    virtual ~Shape() {};
};

class Circle : public Shape {
public:
    void draw() { cout << "Circle::draw\n"; }
    ~Circle() { cout << "~Circle\n"; }
};

class Triangle : public Shape {
public:
    void draw() { cout << "Triangle::draw\n"; }
    ~Triangle() { cout << "~Triangle\n"; }
};

class Square : public Shape {
public:
    void draw() { cout << "Square::draw\n"; }
    ~Square() { cout << "~Square\n"; }
};
```

```
typedef std::vector<Shape*> Container;
typedef Container::iterator Iter;
```

Container ist Vektor von
Zeigern auf Shape

Iter ist Iterator
vector<Shape*>::iterator

```
int main() {
```

```
    Container shapes;
    shapes.push_back(new Circle);
    shapes.push_back(new Square);
    shapes.push_back(new Triangle);
```

shapes ist Container, der
unterschiedliche Objekte vom
Typ shape* verwaltet.

```
    for(Iter i = shapes.begin(); i != shapes.end(); i++)
        (*i)->draw();
```

das ist ein wenig kompliziert:
1) *i ist Inhalt der Inhalt einer Containerkomponente
2) davon die Methode draw() selektieren mit ->
3) (*i)-> Klammern wg. Rang erforderlich
man kann dafür auch einfach i->draw() schreiben, das ->
im Container überladen ist

```
    for(Iter j = shapes.begin(); j != shapes.end(); j++)
        delete *j;
```

```
}
$
```

```
$ Shape
Circle::draw
Square::draw
Triangle::draw
~Circle
~Square
~Triangle
```

Container von Strings

Im folgenden Beispiel werden keine Zeiger in einem Vektor zusammengefasst, sondern Objekte selbst, hier Strings.

Damit wäre ein rudimentärer Editor einfach realisierbar – hier soll eine Datei mit Zeilennummern ausgegeben werden:

```
$ cat main.cpp
// A vector of strings
#include <string>
#include <vector>
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char* argv[]) {
    vector<string> strings;
    string line;
    ifstream in(argv[1]);
    while(getline(in, line))           // read line
        strings.push_back(line);
    // Do something to the strings...
    int i = 1;
    vector<string>::iterator w;
    for(w = strings.begin(); w != strings.end(); w++) {
        cout << i++ << ": " << *w << endl;
    }
}
$
```

3. Iteratoren der STL

Wir haben bereits selbst Iteratoren programmiert. Die **Iteratoren** der **STL** haben alle **gemeinsame Eigenschaften**, die zunächst betrachtet werden.

Jeder Iterator ist eine Abstraktion, die es erlaubt **generischen** Code zu programmieren, d.h. man kann **unterschiedliche getypte Container** verwenden, **ohne** die interne **Organisation** zu kennen.

Jeder STL Container erzeugt Iteratoren, etwa durch:

```
ContainerType::iterator  
ContainerType::const_iterator
```

Jeder STL Container hat folgende Methoden:

```
begin()  
end()
```

die jeweils einen Iterator auf das erste Element des Containers bzw. das Element „nach dem letzten Element“ zurück liefern.

Jeder STL Iterator kann durch ++ (`operator++()`) zum nächsten Element bewegt werden und als Vergleichsoperatoren sind `!=` und `==` realisiert.

Durch Dereferenzierung kann auf das Element, auf das ein Iterator (`it`) zeigt zugegriffen werden; dabei sind zwei Formen möglich:

```
(*it).f();  
it->f();
```

Da dies für jeden STL Container gilt, kann man ein **Funktionstemplate** definieren, das mit jedem STL Container arbeiten kann:

```
$ cat apply.cpp
#include <iostream>
#include <vector>
using namespace std;

template<class Cont, class PtrMemFun>    // Functiontempate
void apply(Cont& c, PtrMemFun f) {
    typename Cont::iterator it;
    it = c.begin();
    while(it != c.end()) {
        ((*it).*f) ();                // execute function f
        it++;                          // next element
    }
}

class Z {
public:
    Z(int ii) : i(ii) {}    // Constructor
    void g() { i++;
        cout << i << " ";
    }

private:
    int i;
};
```

```
$ apply
1 2 3 4 5 6 7 8 9 10
$
```

```
int main() {  
    vector<Z> vz;           // Vector of Z-Objects  
    for(int i = 0; i < 10; i++)  
        vz.push_back(Z(i));  
    apply(vz, &Z::g);      // call member function g  
                                // for each element of vz  
    cout << endl;  
}
```

Achtung!
Adresse einer Funktion ?
vgl. PG1 && Beispiel nächste Folie !!!!

```
$ cat ZeigerAufFunktionen.cpp
#include <iostream>
using namespace std;

void f() {
    cout << "f" << endl;
}

class K {
private:
    int v;
public:
    K(int x) : v(x) {}
    void g() {
        cout << "g " << endl;
    }
};
```

```
int main() {
    int *pi;
    cout << "int" << endl;
    int x = 18;
    cout << "x:          " << x << endl; // value of x
    cout << "&x:          " << &x << endl; // address of x as hex
    cout << "(long) &x: " << (long) & x << endl; // address of x as long

    cout << endl << "fkt" << endl;
    void (*pf) ();
    pf = f;
    cout << "f():          "; f(); // value of f()
    cout << "&f:          " << &f << endl; // address of function f exist (bool 1)
    cout << "pf:          " << pf << endl;
    cout << "*pf:         " << *pf << endl;
    cout << "(long) pf:    " << (long) pf << endl;
    cout << "(long) *pf:  " << (long) *pf << endl;

    cout << "long (&f):  " << long (&f) << endl; // address of function f as long
    cout << "long (f):   " << long (f) << endl; // address of function f as long

    cout << endl << "class member" << endl;
    K k(7);
    cout << "k.g():      "; k.g(); // value of member fkt g of objekt g with 18 as argument
    cout << "&K::g:     " << &K::g << endl; // address of member funktion exists (bool 1)
    printf( "&K::g:  %p\n", &K::g);
}
```

→ Programm mehrfach laufen lassen: wieso ändern sich Adressen?

Iteratoren reversibler Container

Ein Container ist reversibel, wenn er Iteratoren zum Wandern vom Ende bis zum Anfang liefert.

Diese Container beinhalten Methoden `rbegin()` und `rend()`, die reverse-Iteratoren (`reverse_iterator`) zurück liefern.

Beispiele für solche Container sind `vector`, `deque` und `list` (wir werden sie noch behandeln).

```
$ cat Vektor.cpp
#include <vector>
#include <iostream>
using namespace std;
```

```
int main() {
    typedef vector<int> vectorOfInt; // wg. Schreibvereinfachung
    vectorOfInt v;
    vectorOfInt::iterator it;

    for (int i=0; i<5; i++) // Besetzen mit 0-4
        v.push_back(i);

    for (it=v.begin(); it != v.end(); ++it) // Ausgabe aller Elemente
        cout << *it << " "; // Dereferenzieren
    cout << "\n";

    vectorOfInt::reverse_iterator rit;
    for (rit=v.rbegin(); rit != v.rend(); ++rit) // Ausgabe aller Elemente
        cout << *rit << " "; // rueckwaerts
    cout << "\n";

    return 0;
}
$
```

```
$ Vector
0 1 2 3 4
4 3 2 1 0
$
```

Klassifikation der Iteratoren

Innerhalb der STL können folgende Kategorien von Iteratoren unterschieden werden:

- **Input: read-only, one pass**

Zur Zeit existiert in der STL nur zwei Vertreter dieser Kategorie: `istream_iterator` und `istreambuf_iterator`, um von einem Eingabestrom (`istream`) zu lesen.

Dabei wird bei der Dereferenzierung genau ein Element des Eingabestroms gelesen und konsumiert. Der Iterator bewegt sich nur in Richtung „Ende“ des Stroms.

- **Output: write-only, one pass**

Das Gegenstück zu der o.a. Kategorie ist `ostream_iterator` und `ostreambuf_iterator` zum Schreiben eines Ausgabestrom (`ostream`).

- **Forward: multiple read/write**

Diese Iteratoren sind Erweiterungen der beiden o.a. Typen; hier kann ein Element mehrfach gelesen bzw. geschrieben werden.

- **Bidirectional**

Dieser Iteratortyp ist ein Forward-Iterator, der über `operator--` den Zeiger auf ein Container-element dekrementieren kann.

- **Random-access**

Der Random-access Iterator umfasst die Funktionalität des Bidirektionalen Iterator, zudem kann über `operator[]` auf ein beliebige Element verwiesen werden.

Vordefinierte Iteratoren

Einige vordefinierte Iteratoren, wie `iterator` (erzeugt durch `begin()` und `end()`) und `reverse_iterator` (erzeugt durch `rbegin()` und `rend()`) haben wir schon gesehen.

Will man Elemente an **beliebiger Stelle** einfügen ohne zu überschreiben, so sind spezielle Iteratoren erforderlich.

Der `insert_iterator` erlaubt es, am Anfang, am Ende und irgendwo dazwischen **einzufügen**.

```
$ cat Inserter.cpp
// insert prime numbers into a list using insert_iterator
#include <iostream>
#include <list>
#include <iterator>
using namespace std;

int a[] = { 1, 3, 5, 7, 11, 13, 17, 19, 23 }; // prime numbers

template<class Cont>
void frontInsertion(Cont& ci) {
    copy(a, a + sizeof(a)/sizeof(int),
        front_inserter(ci)); // insert at front of ci
    copy(ci.begin(), ci.end(),
        ostream_iterator<int>(cout, " ")); // print out result
    cout << endl;
}
```

```

template<class Cont>
void backInsertion(Cont& ci) {
    copy(a, a + sizeof(a)/sizeof(int),
        back_inserter(ci));
    copy(ci.begin(), ci.end(),
        ostream_iterator<int>(cout, " "));
    cout << endl;
}

template<class Cont>
void midInsertion(Cont& ci) {
    typename Cont::iterator it = ci.begin();
    it++; it++; it++;
    copy(a, a + 2,
        inserter(ci, it)); // copy 2 elements before it
    copy(ci.begin(), ci.end(),
        ostream_iterator<int>(cout, " "));
    cout << endl;
}

int main() {
    list<int> li; // list of ints
    frontInsertion(li); // insert prime number at front
    li.clear(); // clear
    backInsertion(li);
    midInsertion(li);
}
$

```

```

$ Inserter
23 19 17 13 11 7 5 3 1
1 3 5 7 11 13 17 19 23
1 3 5 1 3 7 11 13 17 19 23
$

```

Dies ist nur ein Ausschnitt aus den vordefinierten Iteratoren, weitere sind z.B. `istream_iterator` und `istreambuf_iterator` zur Behandlung von Eingabeströmen.

Sehen Sie sich die Beispiele für Iteratoren unter <http://www.cplusplus.com/reference/stl/> an.

Stellen Sie Fragen dazu.

Nun werden einzelne Container behandelt.

4. Sequenzen

Die STL unterscheidet 2 Arten von Containern:

- Sequenzen, z.B. `vector`, `list`, `deque`, `stack`, `queue`, und `priority_queue`
- Assoziationen, z.B. `set`, `multiset`, `map` und `multimap`

Sequenzen haben die **Eigenschaft**, dass es eine **Ordnung** auf der Menge der Elemente gibt; jedes Element wird am Anfang, am Ende oder an eine feste Position eingefügt. Zu jedem Zeitpunkt hat also jedes Objekt innerhalb des Containers eine definierte Position.

Obwohl alle Sequenzen-Arten diese Eigenschaft haben, existieren **Unterschiede** in der **Effizienz der Operationen** auf ihnen. `vector` und `list` sind beides Sequenzen; sie haben die **selben Schnittstellen** bzgl. des externen Verhaltens. Der Zugriff auf ein beliebiges Element eines `vector` geht in **konstanter Zeit** (wg. indiziertem Zugriff), bei einer **Liste** kann es **lange dauern**, wenn das gesuchte Element weit am Ende steht, da man sequentiell vom Anfang an durchlaufen muss. Andererseits geschieht das **Einfügen** in eine **Liste schnell**, bei einem **Vektor** müssen u.U. **viele Elemente verschoben** werden.

Basisoperationen

Das folgende Programm zeigt die Basisoperationen, die für alle Sequenzen verfügbar sind.

```
$ cat BasicSequenceOperations.cpp
// The operations available for all the basic sequence Containers.
#include <iostream>
#include <vector>
#include <deque>
```

```
#include <list>
using namespace std;

template<typename Container>
void print(Container& c, char* s = "") {
    cout << s << ":" << endl;
    if(c.empty()) {
        cout << "(empty)" << endl;
        return;
    }
    typename Container::iterator it;
    for(it = c.begin(); it != c.end(); it++)
        cout << *it << " ";
    cout << endl;
    cout << "size() " << c.size()
        << " max_size() " << c.max_size()
        << " front() " << c.front()
        << " back() " << c.back() << endl;
}
```

```
template<typename ContainerOfInt>
void basicOps(char* s) {
    cout << "----- " << s << " -----" << endl;
    typedef ContainerOfInt Ci;
    Ci c;
    print(c, "c after default constructor");
    Ci c2(10, 1); // 10 elements, values all 1
    print(c2, "c2 after constructor(10,1)");
    int ia[] = { 1, 3, 5, 7, 9 };
    const int iasz = sizeof(ia)/sizeof(*ia);
    // Initialize with begin & end iterators:
    Ci c3(ia, ia + iasz);
    print(c3, "c3 after constructor(iter,iter)");
    Ci c4(c2); // Copy-constructor
    print(c4, "c4 after copy-constructor(c2)");
    c = c2; // Assignment operator
    print(c, "c after operator=c2");
    c.assign(10, 2); // 10 elements, values all 2
    print(c, "c after assign(10, 2)");
    // Assign with begin & end iterators:
    c.assign(ia, ia + iasz);
    print(c, "c after assign(iter, iter)");
    cout << "c using reverse iterators:" << endl;
    typename Ci::reverse_iterator rit = c.rbegin();
    while(rit != c.rend())
        cout << *rit++ << " ";
    cout << endl;
    c.resize(4);
}
```

```
print(c, "c after resize(4)");
c.push_back(47);
print(c, "c after push_back(47)");
c.pop_back();
print(c, "c after pop_back()");
typename Ci::iterator it = c.begin();
it++; it++;
c.insert(it, 74);
print(c, "c after insert(it, 74)");
it = c.begin();
it++;
c.insert(it, 3, 96);
print(c, "c after insert(it, 3, 96)");
it = c.begin();
it++;
c.insert(it, c3.begin(), c3.end());
print(c, "c after insert(it, c3.begin(), c3.end())");
it = c.begin();
it++;
c.erase(it);
print(c, "c after erase(it)");
typename Ci::iterator it2 = it = c.begin();
it++;
it2++; it2++; it2++; it2++; it2++;
c.erase(it, it2);
print(c, "c after erase(it, it2)");
c.swap(c2);
print(c, "c after swap(c2)");
```

```

c.clear();
print(c, "c after clear()");
}

int main() {
    basicOps<vector<int> >("vector");
    basicOps<deque<int> >("deque");
    basicOps<list<int> >("list");
}
$

```

Hier die Ausgabe für den ersten Aufruf
`basicOps<vector<int> >("vector");`

```

c after default constructor:
(empty)
c2 after constructor(10,1):
1 1 1 1 1 1 1 1 1 1
size() 10 max_size() 1073741823 front() 1 back() 1
c3 after constructor(iter,iter):
1 3 5 7 9
size() 5 max_size() 1073741823 front() 1 back() 9
c4 after copy-constructor(c2):
1 1 1 1 1 1 1 1 1 1
size() 10 max_size() 1073741823 front() 1 back() 1
c after operator=c2:
1 1 1 1 1 1 1 1 1 1
size() 10 max_size() 1073741823 front() 1 back() 1
c after assign(10, 2):
2 2 2 2 2 2 2 2 2 2
size() 10 max_size() 1073741823 front() 2 back() 2
c after assign(iter, iter):
1 3 5 7 9
size() 5 max_size() 1073741823 front() 1 back() 9
c using reverse iterators:
9 7 5 3 1
c after resize(4):
1 3 5 7
size() 4 max_size() 1073741823 front() 1 back() 7
c after push_back(47):
1 3 5 7 47
size() 5 max_size() 1073741823 front() 1 back() 47
c after pop_back():
1 3 5 7

```

```
size() 4 max_size() 1073741823 front() 1 back() 7
c after insert(it, 74):
1 3 74 5 7
size() 5 max_size() 1073741823 front() 1 back() 7
c after insert(it, 3, 96):
1 96 96 96 3 74 5 7
size() 8 max_size() 1073741823 front() 1 back() 7
c after insert(it, c3.begin(), c3.end()):
1 1 3 5 7 9 96 96 96 3 74 5 7
size() 13 max_size() 1073741823 front() 1 back() 7
c after erase(it):
1 3 5 7 9 96 96 96 3 74 5 7
size() 12 max_size() 1073741823 front() 1 back() 7
c after erase(it, it2):
1 96 96 96 3 74 5 7
size() 8 max_size() 1073741823 front() 1 back() 7
c after swap(c2):
1 1 1 1 1 1 1 1 1
size() 10 max_size() 1073741823 front() 1 back() 1
c after clear():
(empty)
```

Die Ausgabe für die anderen Sequenz-Arten ist die gleiche.

Nun werden die Unterschiede einzelner Sequenzarten betrachtet.

vector

Ein `vector` verwaltet seine Objekte in einem durchgehenden Array von Objekten. Die Folge ist:

- **schneller Zugriff** auf die Elemente mittels Index
- Einfügen nicht am Ende ist **aufwendig**
- bei Erreichen der **Arraygrenze** muss eine neues **größeres Array angelegt** werden, die **Elemente** hinein **kopiert** werden und der **alte Platz gelöscht** werden. Das kann bei komplexen Objekte extrem teuer sein.

D.h. Vektoren sollten nur verwendet werden, wenn stets am Ende eingefügt bzw. gelöscht wird und wenn die Größe nicht extrem schwankt und anfänglich komplett allokiert werden kann.

Ein **Problem** bei Vektoren ist, dass die **Iteratoren** darin wegen der effizienten Zugriffsmechanik als **einfache Zeiger** realisiert sind. Dies kann zu fehlerhaftem Verhalten führen, wenn ein Vektor vergrößert wird:

```
$ cat VectorCoreDump.cpp
// How to break a program using a vector
#include <vector>
#include <iostream>
#include <iterator>
using namespace std;
```

```

int main() {
    vector<int> vi(10, 0);
    ostream_iterator<int> out(cout, " ");
    copy(vi.begin(), vi.end(), out);

    vector<int>::iterator i = vi.begin();
    cout << "\n i: " << *i << endl;

    *i = 47;
    copy(vi.begin(), vi.end(), out);

    // Force it to move memory (could also just add enough objects):
    vi.resize(vi.capacity() + 1);

    // Now i points to wrong memory:
    cout << "\n i: " << *i << endl;
    cout << "vi.begin(): " << *vi.begin();

    *i = 48; // Access violation
}
$

```

Sehen Sie sich die Beispiele für vektor unter <http://www.cplusplus.com/reference/stl/> an. Stellen Sie Fragen dazu.

```

$ VectorCoreDump
0 0 0 0 0 0 0 0 0 0
  i: 0
47 0 0 0 0 0 0 0 0 0
  i: 134523768
$ vi.begin(): 47
$

```

deque

Eine deque („double-ended-queue“, ausgesprochen „deck“) ist die grundlegende Sequenz, bei der am **Anfang** (`push_front()`, `pop_front()`) **und Ende** eingefügt und gelöscht werden kann. Der **Zugriff** ist fast so **effizient** wie bei `vector`, aber die Organisation der Elemente ist anders: es werden **mehrere Blöcke** von **zusammenhängendem Speicher** verwendet.

Dadurch muss **kein Kopieren und Löschen** von Objekten bei neuem Speicherbedarf (wie bei `vector`) durchgeführt werden.

Somit ist deque immer anstelle von vector verwendbar.

Das folgende Programm zeigt die Performance-Unterschiede.

```
$ cat StringDeque.cpp
// Converted from StringVector.cpp
#include <string>
#include <deque>
#include <vector>
#include <fstream>
#include <iostream>
#include <iterator>
#include <sstream>
#include <ctime>
using namespace std;
```

```
int main(int argc, char* argv[]) {
    ifstream in(argv[1]);
    vector<string> vstrings;
    deque<string> dstrings;
    string line;
    // Time reading into vector:
    clock_t ticks = clock();
    while(getline(in, line))
        vstrings.push_back(line);
    ticks = clock() - ticks;
    cout << "Read into vector: " << ticks << endl;
    // Repeat for deque:
    ifstream in2(argv[1]);
    ticks = clock();
    while(getline(in2, line))
        dstrings.push_back(line);
    ticks = clock() - ticks;
    cout << "Read into deque: " << ticks << endl;
    // Now compare indexing:
    ticks = clock();
    for(int i = 0; i < vstrings.size(); i++) {
        ostringstream ss;
        ss << i;
        vstrings[i] = ss.str() + ": " + vstrings[i]; // add line numbers
    }
    ticks = clock() - ticks;
    cout << "Indexing vector: " << ticks << endl;
    ticks = clock();
```

```

for(int j = 0; j < dstrings.size(); j++) {
    ostringstream ss;
    ss << j;
    dstrings[j] = ss.str() + ": " + dstrings[j];
}
ticks = clock() - ticks;
cout << "Indexing deque: " << ticks << endl;
// Compare iteration
ofstream tmp1("tmp1.tmp"), tmp2("tmp2.tmp");
ticks = clock();
copy(vstrings.begin(), vstrings.end(),
    ostream_iterator<string>(tmp1, "\n"));
ticks = clock() - ticks;
cout << "Iterating vector: " << ticks << endl;
ticks = clock();
copy(dstrings.begin(), dstrings.end(),
    ostream_iterator<string>(tmp2, "\n"));
ticks = clock() - ticks;
cout << "Iterating deque: " << ticks << endl;
}
$

```

Die Ausgabe des Programm bei einer großen Datei (4,4 MB) zeigt, dass es kaum Unterschiede gibt:

```

$ ls -l song.mp3
-rw---- as users      4433361 Mär  6 16:40 song.mp3
$ StringDeque song.mp3
Read into vector: 1490000
Read into deque:  1470000
Indexing vector:   210000
Indexing deque:   230000
Iterating vector:  40000
Iterating deque:  40000
$

```

Der **indizierte Zugriff** auf Elemente ist mit `vector` und `deque` durch `operator[]` und `at()` möglich. Dabei wird eine **Feldgrenzenprüfung nur mit `at()`** durchgeführt. Was die unterschiedlichen Zugriffsmethoden kosten, zeigt folgendes Programm:

```
$ cat IndexingVsAt.cpp
// Comparing "at()" to operator[]
#include <vector>
#include <deque>
#include <iostream>
#include <ctime>
using namespace std;

int main(int argc, char* argv[]) {
    long count = 1000;
    int sz = 1000;
    if(argc >= 2) count = atoi(argv[1]);
    if(argc >= 3) sz = atoi(argv[2]);
    vector<int> vi(sz);
    clock_t ticks = clock();
    for(int i1 = 0; i1 < count; i1++)
        for(int j = 0; j < sz; j++)
            vi[j];
    cout << "vector[]" << clock() - ticks << endl;
    ticks = clock();
    for(int i2 = 0; i2 < count; i2++)
        for(int j = 0; j < sz; j++)
            vi.at(j);
    cout << "vector::at()" << clock()-ticks <<endl;
```

```

deque<int> di(sz);
ticks = clock();
for(int i3 = 0; i3 < count; i3++)
    for(int j = 0; j < sz; j++)
        di[j];
cout << "deque[]" << clock() - ticks << endl;
ticks = clock();
for(int i4 = 0; i4 < count; i4++)
    for(int j = 0; j < sz; j++)
        di.at(j);
cout << "deque::at()" << clock()-ticks <<endl;
// Demonstrate at() when you go out of bounds:
di.at(vi.size() + 1);
}
$

```

Der kontrollierte Zugriff mit `at()` kostet zwar Zeit, aber die Sicherheit sollte hier höher bewertet werden.

```

$ IndexingVsAt 1000
vector[]      50000
vector::at() 140000
deque[]       290000
deque::at()   390000
Abgebrochen
$

```

Sehen Sie sich die Beispiele für deque unter <http://www.cplusplus.com/reference/stl/> an.

Stellen Sie Fragen dazu.

list

`list` ist als **doppelt verkettete Liste** realisiert und eignet sich daher gut, wenn an einer beliebigen Stelle einzufügen oder zu löschen ist. Der Zugriff auf ein beliebiges Element erfolgt langsam, daher ist `operator[]` **nicht** realisiert.

Die Objekte in einem `list`-Container werden **nie physisch bewegt** – lediglich Zeiger werden „umgebogen“. Daher eignen sie sich gut, wenn speicherintensive Objekte abgelegt werden müssen.

Algorithmen für Listen sind z.B.:

- `sort()` - zum Sortieren
- `merge()` - zum Verbinden von zwei sortierten Listen
- `unique()` entfernt Duplikate in einer sortierten Liste
- `remove()`

```
$ cat list.cpp
// testing list functions
#include <list>
#include <iterator>
#include <algorithm>
#include <iostream>
using namespace std;

int a[] = { 1, 3, 1, 4, 1, 5, 1, 6, 1 };
const int asz = sizeof a / sizeof *a;
int b[] = { 1, 3, 4, 9};
const int bsz = sizeof b / sizeof *b;
```

```
int main() {
    // For output:
    ostream_iterator<int> out(cout, " ");
    list<int> li(a, a + asz); // list out off an array
    list<int> li2(b, b + bsz); // list out off an array
    li.unique(); // Oops! No duplicates removed:
    copy(li.begin(), li.end(), out);
    cout << endl;
    li.sort(); // Must sort it first:
    copy(li.begin(), li.end(), out);
    cout << endl;
    li.unique(); // Now unique() will have an effect:
    copy(li.begin(), li.end(), out);
    cout << endl;
    li.merge(li2); // merge
    copy(li.begin(), li.end(), out);
    cout << endl;
    list<int>::iterator it = li2.begin();
    it++;
    li2.remove(*it); // remove second element
}
$
```

Sehen Sie sich die Beispiele für `list` unter <http://www.cplusplus.com/reference/stl/> an. Stellen Sie Fragen dazu.

set

Eine Menge kann ein Element nur einmal enthalten. Das folgende Programm verwendet set, um Lottozahlen zu generieren:

```
$ cat lotto.cpp
#include <iostream>
#include <set>
#include <cstdlib>           // fur exit()
using namespace std;
const int ZAHLEN = 45;
const int NUMTIP = 6;
unsigned int zz() {         // Zufallszahl erzeugen
    return 1 + random() % ZAHLEN;
}
int main() {
    typedef set <unsigned int> IntSet;
    IntSet tip;
    srandom( time( NULL));  // initialisiere den Zufallszahlengenerator
    while ( tip.size() < NUMTIP) {
        tip.insert(zz()); // Zufallszahl in das Set einfügen
    }
    // Set ausgeben
    for (IntSet:: iterator i = tip. begin(); i != tip. end(); ++ i) {
        cout << *i << endl;
    }
    return 0;
}
$
```

stack

Ein `stack` (sowie auch `queue` und `priority_queue`) sind als **Adapter** implementiert, d.h. man kann neben dem Elementtyp den Containertyp angeben. Dabei kann man angeben, dass der Stack mittels **vector**, **list** oder **deque** erzeugt werden soll.

```
$ cat stack.cpp
// Demonstrates the STL stack
#include <iostream>
#include <fstream>
#include <stack>
#include <list>
#include <vector>
#include <string>
using namespace std;
```

```
typedef stack<string> Stack1;           // Default: deque<string>:  
typedef stack<string, vector<string> > Stack2; // Use a vector<string>:  
typedef stack<string, list<string> > Stack3; // Use a list<string>:  
  
int main(int argc, char* argv[]) {  
    ifstream in(argv[1]);  
    Stack1 textlines; // Try the different versions  
    // Read file and store lines in the stack:  
    string line;  
    while(getline(in, line))  
        textlines.push(line + "\n");  
    // Print lines from the stack and pop them:  
    while(!textlines.empty()) {  
        cout << textlines.top();  
        textlines.pop();  
    }  
}
```

Sehen Sie sich die Beispiele für `stack` unter <http://www.cplusplus.com/reference/stl/> an.

Stellen Sie Fragen dazu.

queue

Eine `queue` ist ein eingeschränkter `deque`: man kann nur an einem Ende einfügen und am anderen Ende entnehmen.

Das letzte Beispiel mit `queue`:

```
$ cat queue.cpp
// Demonstrates the STL queue
#include <iostream>
#include <fstream>
#include <queue>
#include <vector>
#include <string>
using namespace std;
typedef queue<string> Queue;
int main(int argc, char* argv[]) {
    ifstream in(argv[1]);
    Queue textlines;
    // Read file and store lines in the queue:
    string line;
    while(getline(in, line))
        textlines.push(line + "\n");
    // Print lines from the queue and pop them:
    while(!textlines.empty()) {
        cout << textlines.front();
        textlines.pop();
    }
}
```

priority_queue

Wenn ein Objekt in eine `priority_queue` mittels `push()` eingefügt wird, so wird es gemäß einer Prioritätsfunktion (bzw. ein Funktionsobjekt) in die Schlange einsortiert.

Durch `top()` wird dabei immer auf das Element mit der **höchsten Priorität** verwiesen; dementsprechend nimmt `pop()` dieses Element aus der Schlange.

Zunächst betrachten wir eine `queue`, in die 10 Zufalls-Elemente eingefügt werden:

```
$ cat queue2.cpp
#include <iostream>
#include <queue>
#include <cstdlib>
#include <ctime>
using namespace std;
```

```
int main() {  
    queue<int> qi;  
    srand(time(0)); // Seed random number generator  
    for(int i = 0; i < 10; i++)  
        qi.push(rand() % 25);  
    while(!qi.empty()) {  
        cout << qi.front() << ' ';  
        qi.pop();  
    }  
    cout << endl;  
}  
$
```

```
$ queue2  
3 1 20 2 24 11 7 21 5 21  
$
```

Ändern wir das Programm, so dass eine **priority_queue** verwendet wird, entsteht eine **sortierte** Ausgabe.

```
$ cat priorityQueue.cpp
#include <iostream>
#include <queue>
#include <cstdlib>
#include <ctime>
using namespace std;

int main() {
    priority_queue<int> pqi;
    srand(time(0)); // Seed random number generator
    for(int i = 0; i < 10; i++)
        pqi.push(rand() % 25);
    while(!pq.empty()) {
        cout << pqi.top() << ' ';
        pqi.pop();
    }
    cout << endl;
}
$
```

```
$ priorityQueue
22 17 16 16 16 12 12 11 5 3
$
```

Hier wurde das Default-Template `less` zur Vergabe der Priorität verwendet, d.h. wenn der Inhalt der Schlange „**3 5 == top**“ ist, das nächste Element zum Einfügen die **11** ist, wird **less(5,11)** true liefern und die 11 als Element mit höchster Priorität das neue Top-Element .

Damit haben die Zahlen mit dem großen Werten die höhere Priorität.

Soll die **höchste Priorität** an Zahlen mit **kleinem Wert** vergeben werden, braucht man eine andere Funktion. Wir programmieren ein Funktionsobjekt Reverse, das zwei Integer vergleicht:

```
$#include <iostream>
#include <queue>
#include <cstdlib>
#include <ctime>
using namespace std;
```

```
struct Reverse {  
    bool operator() (int x, int y) {  
        return y < x;  
    }  
};
```

Konstruktor benutzt ein
Vergleichsobjekt.

```
int main() {  
    priority_queue<int, vector<int>, Reverse> pqi;  
    srand(time(0));  
    for(int i = 0; i < 10; i++)  
        pqi.push(rand() % 25);  
    while(!pq.empty()) {  
        cout << pqi.top() << ' ';  
        pqi.pop();  
    }  
    cout << endl;  
}
```

```
$ priorityQueue2  
5 6 7 14 16 16 16 20 21 24  
$
```

Alternativ könnte auch das Funktionstemplate `greater` der STL verwendet werden:

```
priority_queue<int, vector<int>, greater<int> > pqi;
```

Prioritäts-Schlangen können nicht nur für einfach Objekte (wie o.a. int), sondern für komplexeres Klassen verwendet werden. Dann ist die Vergleichsoperator selbst zu programmieren. Dies ist am Beispiel einer ToDo Liste gezeigt, bei der eine Aufgabe (item) eine zweistufige Priorität haben kann:

```
$ cat ToDo.cpp
// A more complex use of priority_queue
#include <iostream>
#include <queue>
#include <string>
using namespace std;

class ToDoItem {
    char primary;
    int secondary;
    string item;
public:
    ToDoItem(string td, char pri = 'A', int sec = 1)
        : item(td), primary(pri), secondary(sec) {}
    friend bool operator<(const ToDoItem& x, const ToDoItem& y) {
        if(x.primary > y.primary)
            return true;
        if(x.primary == y.primary)
            if(x.secondary > y.secondary)
                return true;
        return false;
    }

    friend ostream&
```

```

operator<<(ostream& os, const ToDoItem& td) {
    return os << td.primary << td.secondary
        << ": " << td.item;
}
};

int main() {
    priority_queue<ToDoItem> toDoList;
    toDoList.push(ToDoItem("Prog II Uebung", 'C', 4));
    toDoList.push(ToDoItem("Kino mit JL", 'A', 1));
    toDoList.push(ToDoItem("Essen mit EC", 'B', 3));
    toDoList.push(ToDoItem("Abwaschen bei AS", 'C', 3));
    toDoList.push(ToDoItem("Urlaub mit XY", 'A', 2));
    toDoList.push(ToDoItem("Muell raus bringen", 'B', 1));
    while(!toDoList.empty()) {
        cout << toDoList.top() << endl;
        toDoList.pop();
    }
}
$

```

```

$ ToDo
A1: Kino mit JL
A2: Urlaub mit XY
B1: Muell raus bringen
B3: Essen mit EC
C3: Abwaschen bei AS
C4: Prog II Uebung
$

```

Sehen Sie sich die Beispiele für queue und priority_queue unter <http://www.cplusplus.com/reference/stl/> an. Stellen Sie Fragen dazu.

5. Assoziative Container

Die Templates `set`, `map`, `multiset` und `multimap` werden assoziative Container genannt, da sie einen Wert mit einem Schlüssel assoziieren.

Die am meisten verwendete **Operation** für assoziative Container ist das **Einfügen eines Schlüssel-Werte Paares** und das **Suchen eines Wertes**, der zu einem Schlüssel passt.

map

Im folgenden Beispiel wird eine `map` verwendet, um zu einem Integer, die zugehörige Wurzel abzuspeichern.

```
$ cat map.cpp
// Basic operations with maps using (n, sqrt(n))
#include <iostream>
#include <map>
using namespace std;

int main() {
    map<int, double> dm;

    cout << "\n-----\n";
    for(int i = 0; i < 3; i++)
        dm[i]; // Automatically makes pairs (init with 0)
    for(int j = 0; j < dm.size(); j++)
        cout << "dm[" << j <<"] = " << dm[j] << endl;
```

```
$ map
dm[0] = 0
dm[1] = 0
dm[2] = 0
```

```

cout << "\n-----\n";
dm[6] = sqrt(double(6)); // now we have 7 pairs
for(int j = 0; j < dm.size(); j++)
    cout << "dm[" << j <<"] = " << dm[j] << endl;

```

```

dm[0] = 0
dm[1] = 0
dm[2] = 0
dm[3] = 0
dm[4] = 0
dm[5] = 0
dm[6] = 2.44949

```

```

cout << "\n-----\n";
dm.insert(make_pair(8, sqrt(double(8)))); // make_pair explained later!
for(int j = 0; j < dm.size(); j++)
    cout << "dm[" << j <<"] = " << dm[j] << endl;

```

```

dm[0] = 0
dm[1] = 0
dm[2] = 0
dm[3] = 0
dm[4] = 0
dm[5] = 0
dm[6] = 2.44949
dm[7] = 0
dm[8] = 2.82843

```

Anzahl von Einträgen mit Schlüssel 6, (kann nur Wert 0 oder 1 annehmen, bei multimap auch größere Werte)

```

cout << "\n-----\n";
cout << "\n dm.count(6) = " << dm.count(6) << endl;
cout << "\n-----\n";

```

```
dm.count(6) = 1
```

```
map<int, double>::iterator it = dm.find(6);
if(it != dm.end())
    cout << "value:" << (*it).second << " found in dm at location 6" << endl;

for(it = dm.begin(); it != dm.end(); it++)
    cout << (*it).first << ":" << (*it).second << ", ";
cout << "\n-----\n";
}
```

```
$ value:2.44949 found in dm at location 6
0:0, 1:0, 2:0, 3:0, 4:0, 5:0, 6:2.44949, 7:0, 8:2.82843,
```

Im o.a. Beispiel haben wir gesehen, dass eine `map` eine Menge von Paaren ist, wobei `operator[]` dafür sorgt, dass bei großen Schlüsseln immer automatisch Paare „dazwischen“ erzeugt werden. Dies kann bei großen Objekten sehr **speicherintensiv** sein!.

Der Typ von `map`, also `map::value_type` ist definiert als:

```
typedef pair<const Key, T> value_type;
```

wobei `pair` in `<utility>` definiert ist als:

```
template <class T1, class T2>
struct pair {
    T1 first;
    T2 second;
    pair();
    pair(const T1& x, const T2& y)
        : first(x), second(y) {}
    // Templated copy-constructor:
    template<class U, class V>
        pair(const pair<U, V> &p);
};
```

Im letzten Beispiel haben wir `map` in einer **Array-analogen** Weise verwendet. Eine Array ist eine Assoziation von `int` nach Komponententyp. `map`'s sind aber mehr, man kann Assoziationen von Objekten nach beliebigen anderen Objekten vollziehen. Im nachfolgenden Beispiel wird mittels `map` eine **Abbildung von `string` nach `int`** machen, um die Telefonnummern abzuspeichern und zu suchen.

```
$ cat phonebook.cpp
// phone book with maps
#include <iostream>
#include <map>
using namespace std;

int main() {
    typedef map<string, int> phoneBook;
    phoneBook b;
    b.insert(make_pair("jenni", 123456));           // insert entry into phonebook
    b.insert(make_pair("bret", 654321));
    b["as"] = 186475;

    phoneBook::iterator it;                        // print phonebook
    for(it = b.begin(); it != b.end(); it++)
        cout << (*it).first << ": " << (*it).second << endl;

    string name;
    cout << "Name: ";
    cin >> name;
    if((it = b.find(name)) != b.end())             // search entry
        cout << name << " has number " << (*it).second << endl;
    else
        cout << name << "' number not found!"<< endl;
}
$
```

Sehen Sie sich die Beispiele für map unter <http://www.cplusplus.com/reference/stl/> an. Stellen Sie Fragen dazu.

multimap

Eine `multimap` erlaubt **mehrere Einträge** zu einen Schlüssel. Dadurch ist beim Versuch, ein Element zu finden, ein wenig mehr Aufwand zu betreiben, wie im Fall der `map`.

```
$ cat multimap.cpp
// phone book with maps
#include <iostream>
#include <iterator>
#include <map>
using namespace std;

int main() {
    typedef multimap<string, int> phoneBook;
    typedef phoneBook::iterator phoneBookIter;

    phoneBook b;
    b.insert(make_pair("jenni", 123456));
    b.insert(make_pair("jenni", 18));
    b.insert(make_pair("bret", 654321));

    phoneBookIter it;
    for(it = b.begin(); it != b.end(); it++)
        cout << (*it).first << ": " << (*it).second << endl;
}
```

```
string name;
cout << "Name: ";
cin >> name;
/* if((it = b.find(name)) != b.end()) // first occurrences
    cout << name << " has number " << (*it).second << endl;
else
    cout << name << "' number not found!"<< endl;
*/

pair<phoneBookIter,phoneBookIter> range; // pair of phoneBooks iterators
range = b.equal_range(name); // pair of iterators to first
//and last matching phonebook entry

for(it = range.first; it != range.second; it++)
    cout << (*it).first << ": " << (*it).second << endl;
}
$
```

Sehen Sie sich die Beispiele für `multi_map` unter <http://www.cplusplus.com/reference/stl/> an.

Stellen Sie Fragen dazu.

multiset

Sollen mehrere Elemente in einer Menge enthalten sein, die den selben Wert repräsentieren, so kann ein `multiset` verwendet werden.

```
$ cat multiset.cpp
#include <iostream>
#include <set>
using namespace std;
```

```
int main() {
    typedef multiset <unsigned int> IntMultiSet;
    IntMultiSet s;
    s.insert(1);
    s.insert(2);
    s.insert(1);

    // Set ausgeben
    for (IntMultiSet:: iterator i = s. begin(); i != s. end(); ++ i) {
        cout << *i << endl;
    }
    return 0;
}
$
```

**Sehen Sie sich die Beispiele für `multi_set` unter <http://www.cplusplus.com/reference/stl/> an.
Stellen Sie Fragen dazu.**

6.STL Algorithmen

STL Algorithmen sind Funktionstemplates, die mit den STL Containern arbeiten. Die Idee ist es, sich beim Programmieren auf das Was und nicht auf das Wie zu konzentrieren.

„The STL algorithms provide a *vocabulary* with which to describe solutions. That is, once you become familiar with the algorithms you'll have a new set of words with which to discuss what you're doing, and these words are at a higher level than what you've had before. You don't have to say "this loop moves through and assigns from here to there ... oh, I see, it's copying!" Instead, you say **copy()**. This is the kind of thing we've been doing in computer programming from the beginning – creating more dense ways to express *what* we're doing and spending less time saying *how* we're doing it. „

[Bruce Eckel, 2000]

Funktionsobjekte

Ein Funktionsobjekt hat einen überladenen `operator()`. Daher kann ein Funktionstemplate nicht erkennen, ob es sich um einen Zeiger auf eine Funktion oder ein Objekt mit überladenem `operator()` handelt.

Beispiel:

```
$ cat functionObject.cpp
// Simple function objects (functor)
#include <iostream>
using namespace std;
```

```

template<class UnaryFunc, class T> // template function
void callFunc(T& x, UnaryFunc f) {
    f(x);
}

void g(int& x) { // function
    x = 47;
}

class UFunc { // functor: objekt with overloaded operator()
public:
    void operator() (int& x) {
        x = 48;
    }
};

int main() {
    int y = 0;
    callFunc(y, g); // template function used with pointer to function g
    cout << y << endl;
    y = 0;
    callFunc(y, UFunc()); // template function used with functor
    cout << y << endl;
}
$

```

Aus **f** und **x** wird **f(x)** gemacht, ob f ein Funktor oder ein Zeiger auf eine Funktion ist, ist egal.

Diese Eigenschaft macht man sich bei STL-Algorithmen zu nutze.

Die STL Algorithmen sind sehr mächtig, aber erfordern einen relativ hohen Einarbeitungsaufwand. Dies würde den Rahmen der Vorlesung sprengen.

Einige Beispiele sollen die Verwendung demonstrieren. Eine ausführlicher Beschreibung ist in [Bruce Eckel, Thinking in C++] zu finden.

Die Bearbeitung der Elemente eines Containers erfolgt mit vorhandenen Standard-Algorithmen. Es gibt Algorithmen zum Suchen, Sortieren, Löschen, Ändern von Elementen.

Man unterscheidet zwischen

- **globalen Funktionen:**
Der Algorithmus musste nur einmal für alle Container implementiert werden.
- **Elementfunktionen:**
Einige Funktionen sind auch als Elementfunktion der Klasse vorhanden.

Die vorhandenen Algorithmen werden im Rahmen des folgenden Teils (Beispiele) verdeutlicht.

find

```
$ cat vector.cpp
#include <vector>
#include <algorithm> // wg. find()
main() {
    typedef vector<int> vectorOfInt;
    vectorOfInt v;
    vectorOfInt::iterator it;

    // Besetzen mit 0-9
    for (int i=0; i<9; i++)
        v.push_back(i);

    // Ausgabe aller Elemente
    for (it=v.begin(); it != v.end(); ++it)
        cout << *it << " ";
    cout << "\n";
    // Suche nach Element
    int suche;
    cout << "Bitte Integer eingeben: ";
    cin >> suche;
    it = find(v.begin(), v.end(), suche);
    if (it != v.end())
        cout << "gefunden\n";
    else
        cout << "nicht gefunden\n";
}
```

sort

Sortieren erfolgt über die globale Funktion `sort()`. Die Sortierreihenfolge kann mit einem *Funktionsobjekt* (`greater<typ>()`) bestimmt werden.

```
#include <string>
#include <algorithm>          // fuer sort
#include <functional>        // fuer greater
using namespace std;
int main()
{
    typedef vector<string> VectorOfStrings;
    VectorOfStrings v;
    v.push_back("Bo");
    v.push_back("Eva");
    v.push_back("Adam");

    sort(v.begin(), v.end(), less<string>());
    for(unsigned int i=0; i < v.size(); ++i) {
        cout << v[i] << endl;
    }
    cout << endl;

    sort(v.begin(), v.end(), greater<string>());
    for(unsigned int i=0; i < v.size(); ++i) {
        cout << v[i] << endl;
    }
}
$
```

```
$ sort  
Adam  
Bo  
Eva  
  
Eva  
Bo  
Adam  
$
```

copy

Kopieren erfolgt über die globale Funktion `copy()`.

Der Zielbereich muss vorher groß genug definiert werden.

Die InsertIteratoren

- `back_inserter(container)`
- `front_inserter(container)` und
- `inserter(container, pos)`

erweitern den Zielbereich automatisch.

Beispiel (**ohne** Insert Iterator)

```
$ cat copy1.cpp
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;
int main() {
    typedef vector<int> VectorOfInts;
    VectorOfInts tip1, tip2;

    tip1.push_back(45);
    tip1.push_back(1);
    tip1.push_back(17);
    tip1.push_back(27);
    tip2.resize(tip1.size());           // Vorher genügend Platz
                                        // für die neuen Elemente schaffen.

    copy( tip1.begin(),                 // Quelle start
          tip1.end(),                 // Quelle ende
          tip2.begin());              // Ziel start

    for(unsigned int i=0; i < tip2.size(); ++i) {
        cout << tip2[i] << endl;
    }
}
$
```

Beispiel (**mit** Insert Iterator)

```
$ cat copy2.cpp
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;
int main() {
    typedef vector<int> VectorOfInts;
    VectorOfInts tip1, tip2;

    tip1.push_back(45);
    tip1.push_back(1);
    tip1.push_back(17);
    tip1.push_back(27);

    copy(tip1.begin(),           // Quelle start
         tip1.end(),           // Quelle ende
         back_inserter(tip2)); // Ziel

    for(unsigned int i=0; i < tip2.size(); ++i) {
        cout << tip2[i] << endl;
    }
}
$
```