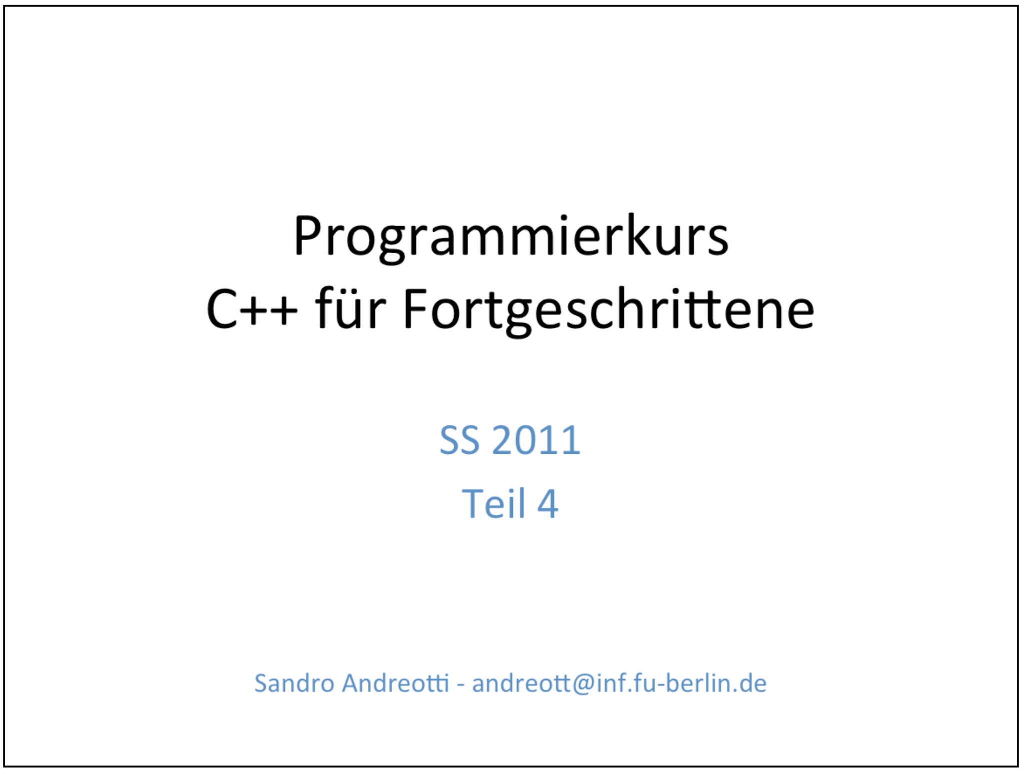
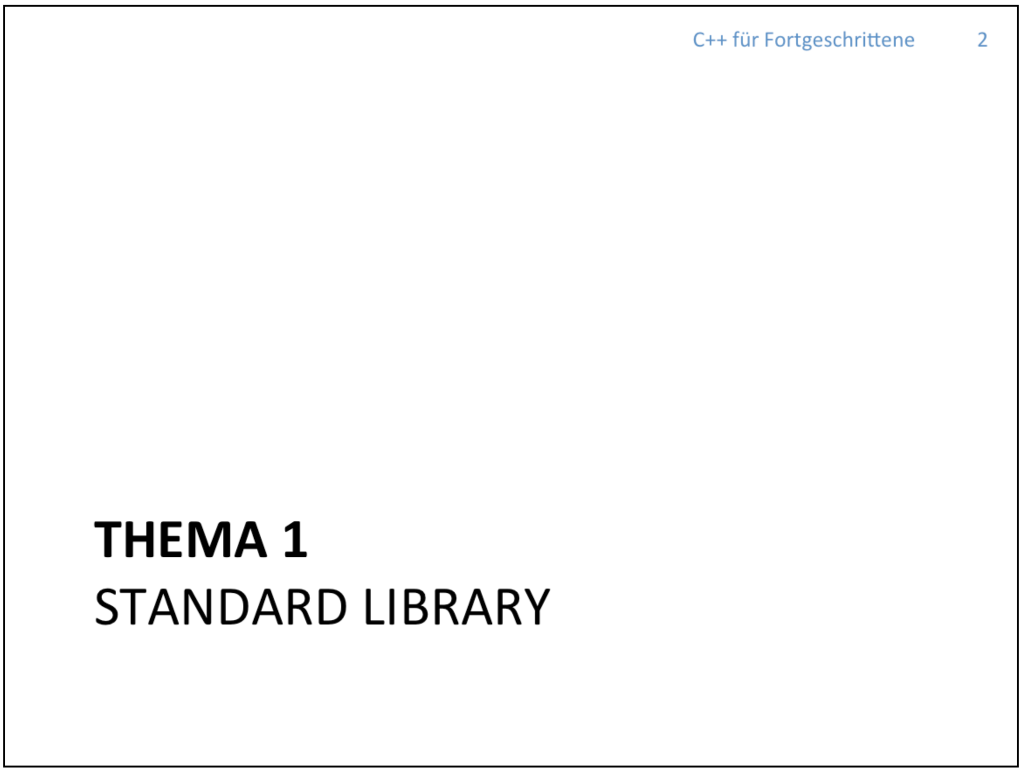


Programmierkurs
C++ für Fortgeschrittene

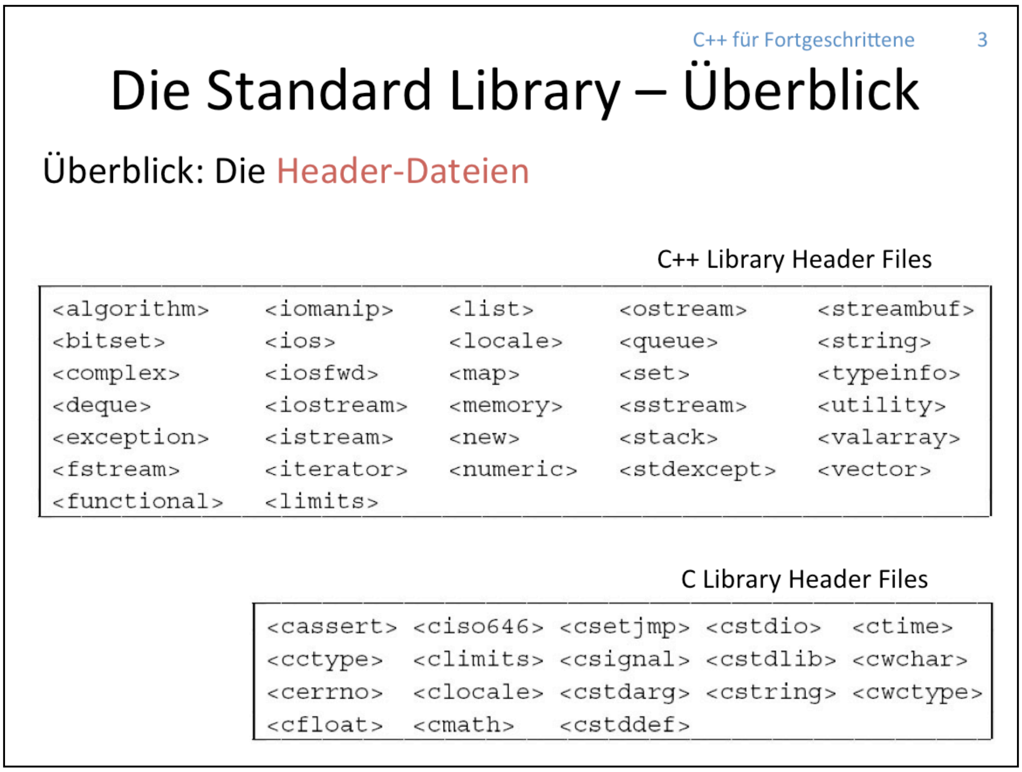
SS 2011

Teil 4

Sandro Andreotti - andreott@inf.fu-berlin.de



THEMA 1
STANDARD LIBRARY



Die Standard Library – Überblick

Überblick: Die Header-Dateien

C++ Library Header Files

```
<algorithm>   <iomanip>    <list>      <ostream>     <streambuf>
<bitset>      <ios>        <locale>    <queue>       <string>
<complex>     <iosfwd>     <map>       <set>         <typeinfo>
<deque>       <iostream>   <memory>    <sstream>     <utility>
<exception>   <istream>    <new>       <stack>       <valarray>
<fstream>     <iterator>   <numeric>   <stdexcept>   <vector>
<functional>  <limits>
```

C Library Header Files

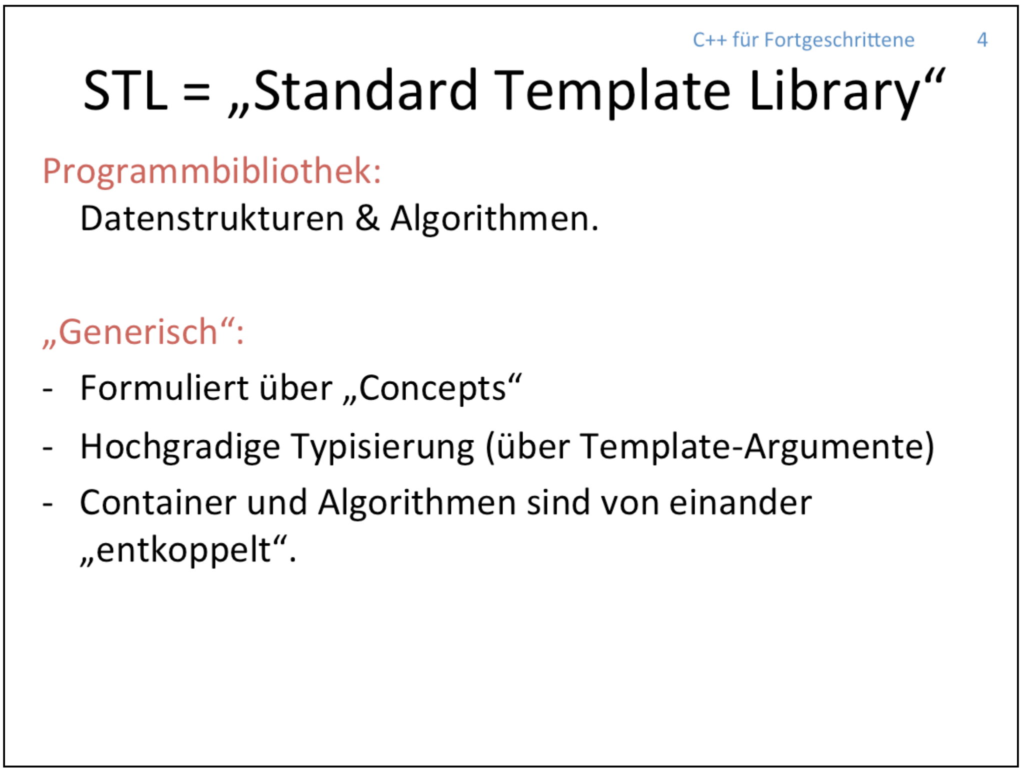
```
<cassert> <ciso646> <csetjmp> <cstdio>  <ctime>
<cctype>  <climits> <csignal> <cstdlib> <cwchar>
<cerrno>  <clocale> <cstdarg> <cstring> <cwctype>
<cfloat>  <cmath>   <cstddef>
```

32 C++ Library Headers und 18 C Library Headers.

Die C-Library Header entsprechen den üblichen Headern, wie sie schon längere Zeit bei C Standard sind und waren, jedoch mit einem zusätzlichen „c“ am Anfang, und ohne „.h“ am Ende. Statt „stdio.h“ heißt der Header für die C standard Library zur Ein- und Ausgabe also „cstdio“. Die Definitionen in diesen Dateien sind zum großen Teil in den Namespace **std** gewandert. Ansonsten gibt es hier und da kleinere Unterschiede zu den üblichen C-Library Dateien.

Literatur & Links:

- ISO-IEC- 14882 (1992): Programming languages - C++, Kapitel 17 – 27
- „C-Library Reference Guide“: http://www.acm.uiuc.edu/webmonkeys/book/c_guide/



STL = „Standard Template Library“

Programmbibliothek:

Datenstrukturen & Algorithmen.

„Generisch“:

- Formuliert über „Concepts“
- Hochgradige Typisierung (über Template-Argumente)
- Container und Algorithmen sind von einander „entkoppelt“.

"Generisches Programmieren" bedeutet: "Programmieren abstrakt von konkreten Datentypen": Datentypen wie **std::vector<T>** funktionieren für beliebige Typen **T** funktionieren, Algorithmen wie **std::sort** arbeiten auf einer Vielzahl von Iteratoren, usw.

Literatur:

- Meyers, „Effective STL“ (2001)
- Austern, „Generic Programming and the STL“ (1999)
- SGI STL: http://www.sgi.com/tech/stl/\
- Portierung der SGI-STL auf andere Platformen: STLport: http://www.stlport.org/

Concepts - Motivation

Aufgabe: Programmiere eine Ausgabefunktion...

```
void printSequence(char * start, char * end)
{
    while (start != end)
    {
        cout << *start;
        ++start;
    }
}
```

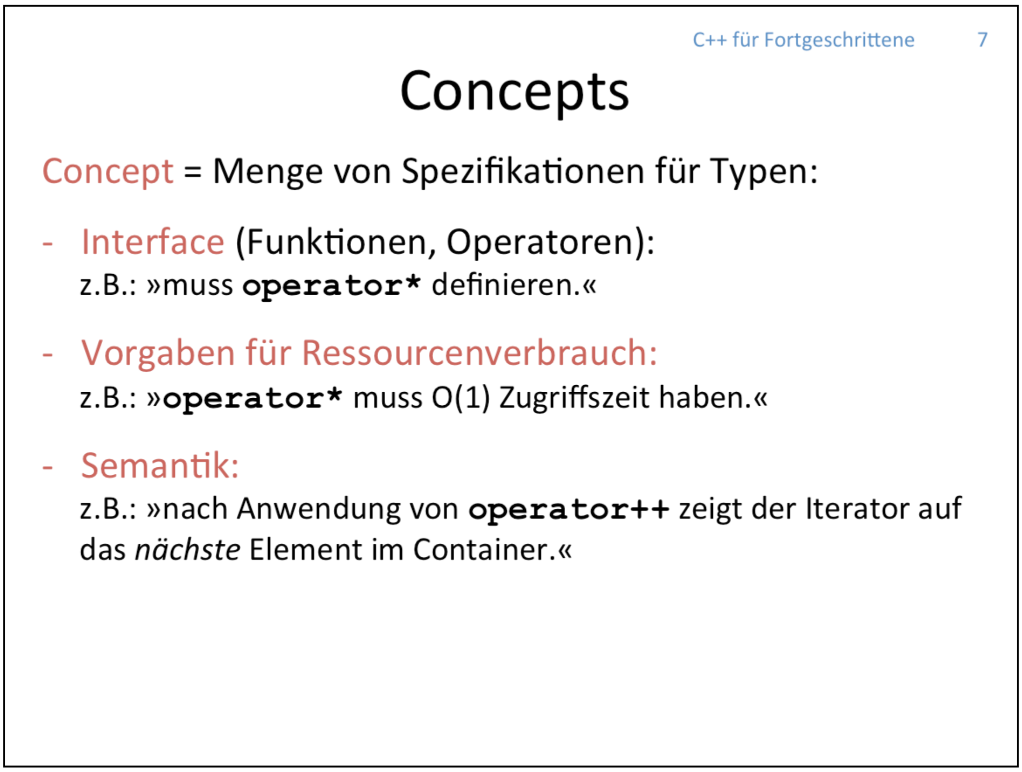
... aber bitte so, dass sie nicht nur für `char *` funktioniert!

Concepts - Motivation (II)

Lösung: Templates

```
template <typename T>
void printSequence(T start, T end)
{
    while (start != end)
    {
        cout << *start;
        ++start;
    }
}
```

Frage: Für welche Typen `T` funktioniert das?



Concepts

Concept = Menge von Spezifikationen für Typen:

- Interface (Funktionen, Operatoren):
  z.B.: »muss **`operator*`** definieren.«
- Vorgaben für Ressourcenverbrauch:
  z.B.: »**`operator*`** muss O(1) Zugriffszeit haben.«
- Semantik:
  z.B.: »nach Anwendung von **`operator++`** zeigt der Iterator auf das *nächste* Element im Container.«

Wird ein Typ verwendet, der die durch das Concept festgelegten Anforderungen nicht erfüllt, so wird in den meisten Fällen ein (mit hoher Wahrscheinlichkeit wenig verständlicher) Compilerfehler ausgelöst.

Concepts werden nicht im eigentlichen Sinne durch ein Programm festgeschrieben. Dafür gäbe es keine Konstrukte in C/C++. Sie werden vielmehr in Form einer Dokumentation festgelegt, an die sich dann der Programmierer halten muss, und zwar weitgehend ohne besondere Unterstützung durch den Compiler.

Anmerkung: Für unser Beispiel wäre das Konzept des Iterators angemessen: Nur Iteratoren dürfen bei **printSequence** als Typen **T** für die Argumente verwendet werden.

STL Varianten

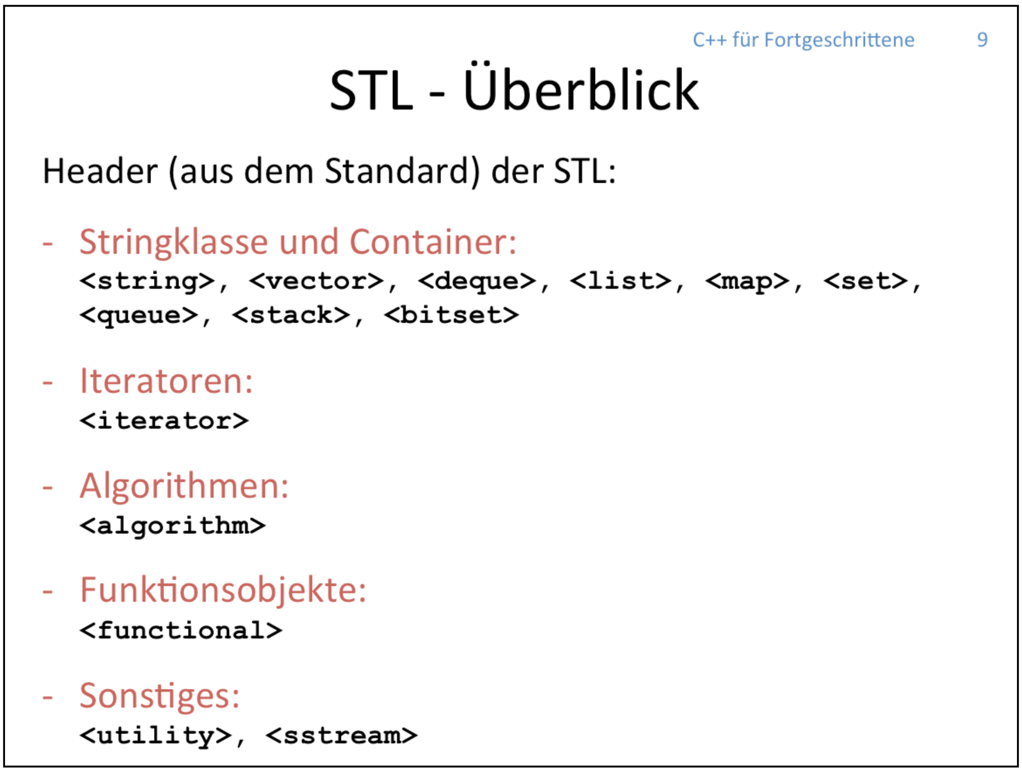
- Spezifikation lässt Implementierungsvarianten zu.

Bsp.: **`allocator`** der SGI-STL implementiert für kleine Speicherblöcke einen Pool Allocator.

- Gewisse nicht-standard Container & Algorithmen sind weit verbreitet.

Bsp.:

- **`hash_map`**
- **`hash_multimap`**
- **`hash_set`**
- **`rope`**

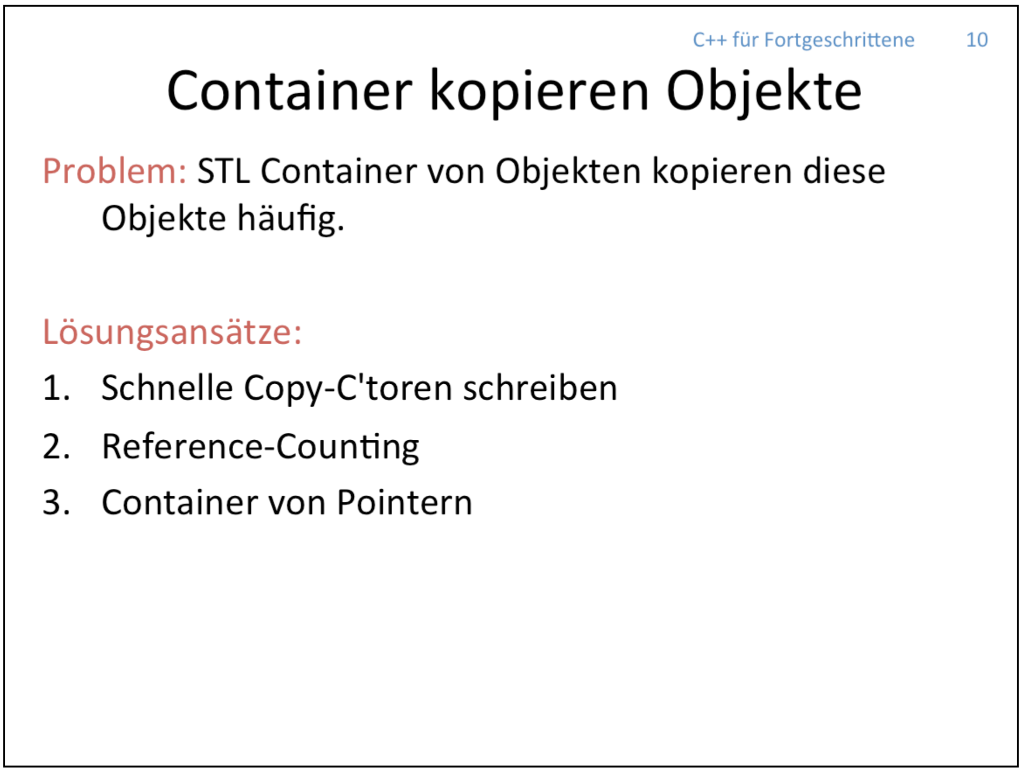


STL - Überblick

Header (aus dem Standard) der STL:

- Stringklasse und Container:
  `<string>, <vector>, <deque>, <list>, <map>, <set>, <queue>, <stack>, <bitset>`
- Iteratoren:
  `<iterator>`
- Algorithmen:
  `<algorithm>`
- Funktionsobjekte:
  `<functional>`
- Sonstiges:
  `<utility>, <sstream>`

In <sstream> steht u.A. Code für die Integration der STL mit den IOStreams, insbesondere **basic_istringstream**, **basic_ostringstream**, **basic_stringstream**, **basic_stringbuf**. Aus diesem Grund ist dieser Header hier mit aufgeführt.



Container kopieren Objekte

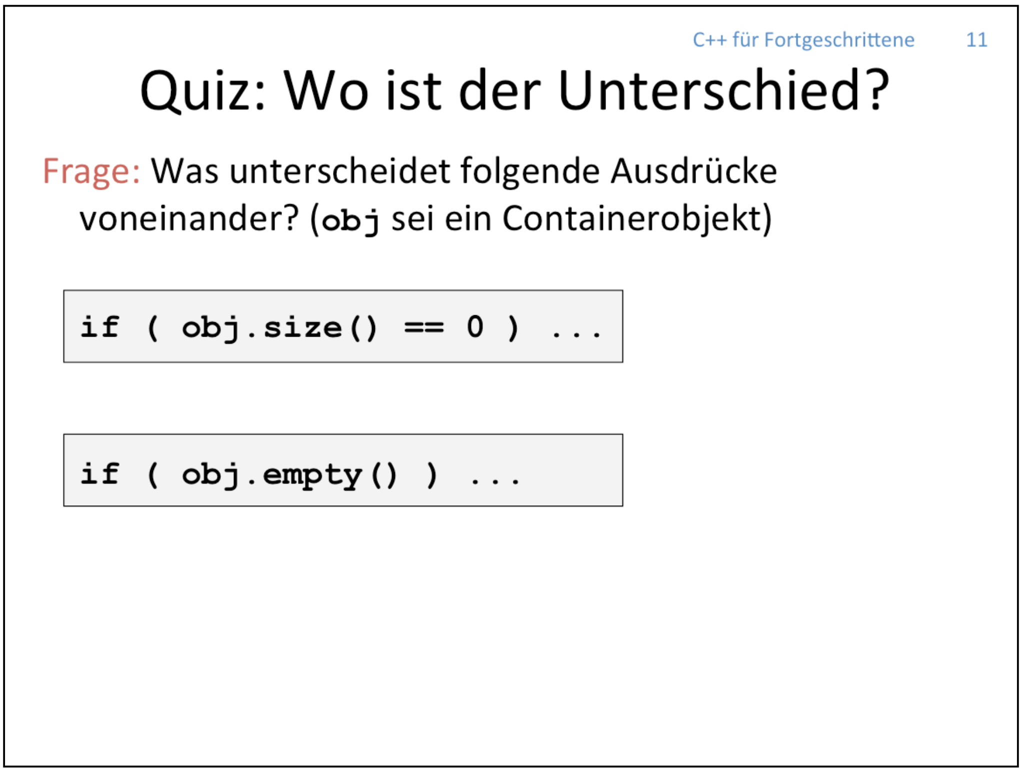
Problem: STL Container von Objekten kopieren diese Objekte häufig.

Lösungsansätze:

1. Schnelle Copy-C'toren schreiben
2. Reference-Counting
3. Container von Pointern

zu 2. Reference-Counting: Copy C‘tor und Copy Assignment Operator legen tatsächlich keine echte Kopien des Objekts an, sondern erhöhen im Wesentlichen nur einen Zähler. Der Zähler gibt an, wie viele (angebliche) Objektkopien sich die Daten des Objekts teilen. Erst wenn ein Objekt verändert werden soll, wird eine echte Kopie angelegt und diese geändert.

zu 3.Container von Pointern: Das Problem ist, dass nun der Benutzer selbst darauf achten muss, dass er den Speicherbereich, in dem sich die tatsächlichen Objekte befinden, vor der Zerstörung des Containers wieder frei gibt.



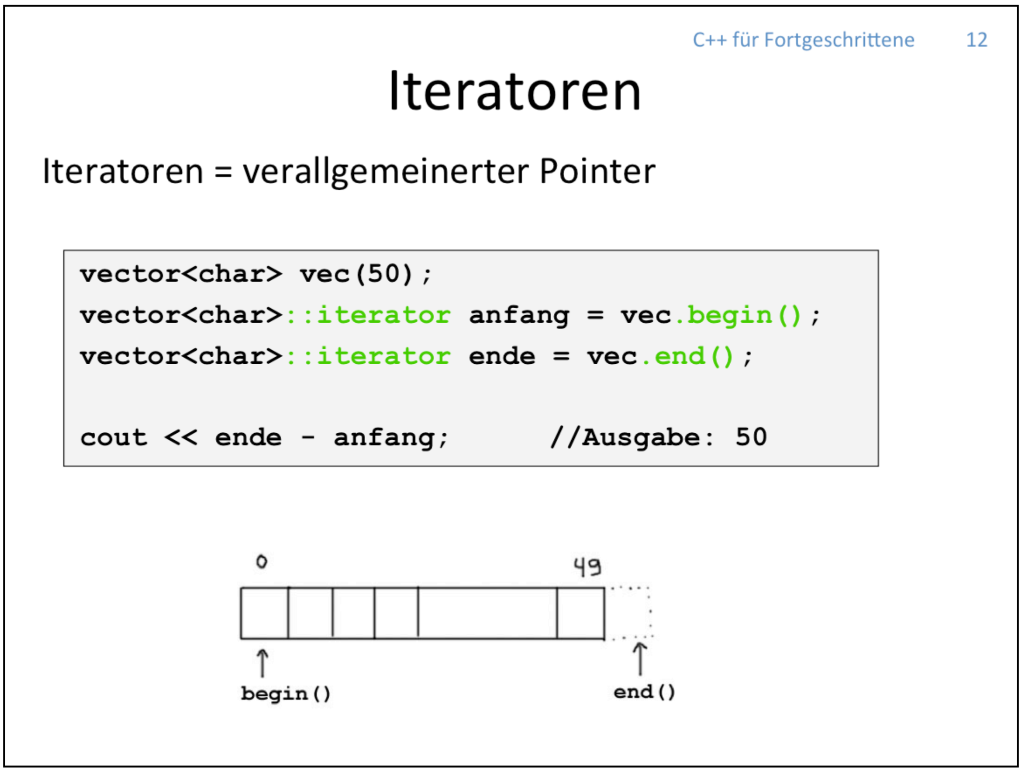
Quiz: Wo ist der Unterschied?

Frage: Was unterscheidet folgende Ausdrücke voneinander? (`obj` sei ein Containerobjekt)

```
if ( obj.size() == 0 ) ...
```

```
if ( obj.empty() ) ...
```

Antwort: Die Semantik ist die gleiche, und bei den meisten Containern gibt es tatsächlich keinen Unterschied. Bei manchen Containern jedoch (insbesondere bei Listen) ist die zweite Methode deutlich schneller: Eine Liste muss eventuell (je nach Implementierung) alle Listenelemente durchlaufen, um festzustellen, wie lang sie ist, ob sie jedoch leer ist oder nicht kann sie in jedem Fall sofort feststellen.



Iteratoren

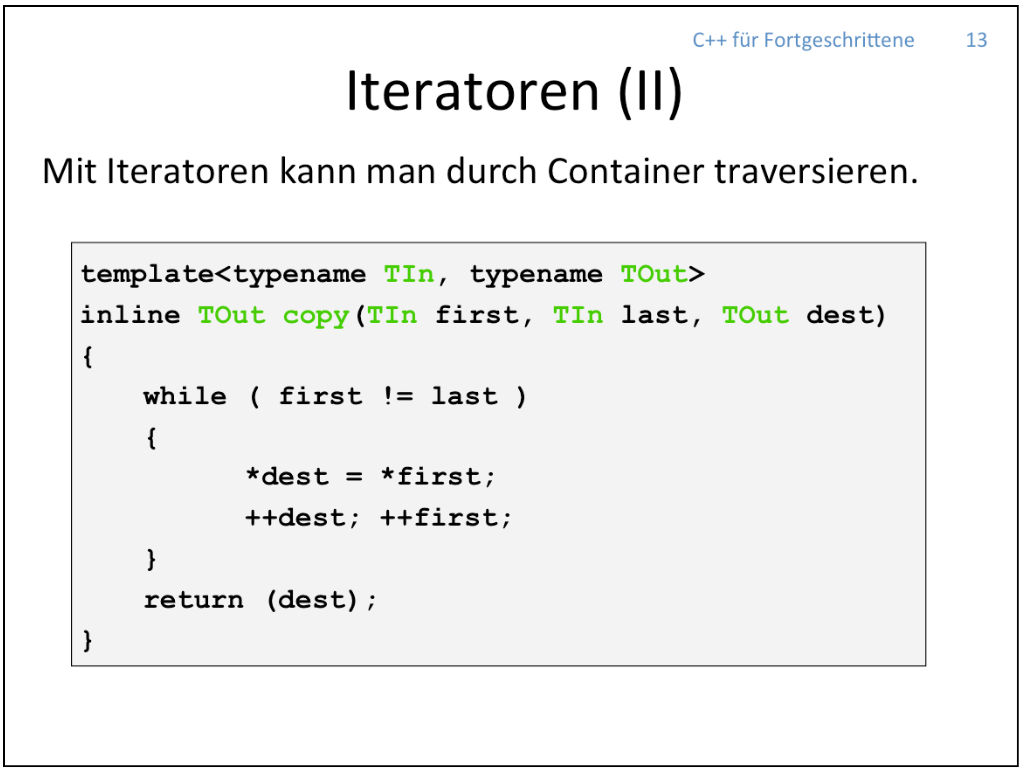
Iteratoren = verallgemeinerter Pointer

```
vector<char> vec(50);
vector<char>::iterator anfang = vec.begin();
vector<char>::iterator ende = vec.end();

cout << ende - anfang;      //Ausgabe: 50
```

0
49
begin()
end()

Man kann tatsächlich auch einfache Pointer als Iteratoren verwenden, doch die meisten Iteratoren verlangen nur einen Teil der Operatoren, die mit Pointern möglich sind. Iteratoren von Listen beispielsweise sind lediglich „bidirektional“, d.h. sie implementieren zwar ++ und --, aber nicht + und -.

Anmerkung: Man beachte, dass **vec.end()** nicht etwa auf das letzte Element in **vec**, sondern auf die Position *hinter* das letzte Element zeigt. Andernfalls wäre es nicht möglich, aus der Differenz der Iteratoren die Länge der Sequenz zu ermitteln. Dies ist insbesondere zu beachten, wenn der **reverse_iterator** – Adaptor verwendet wird.

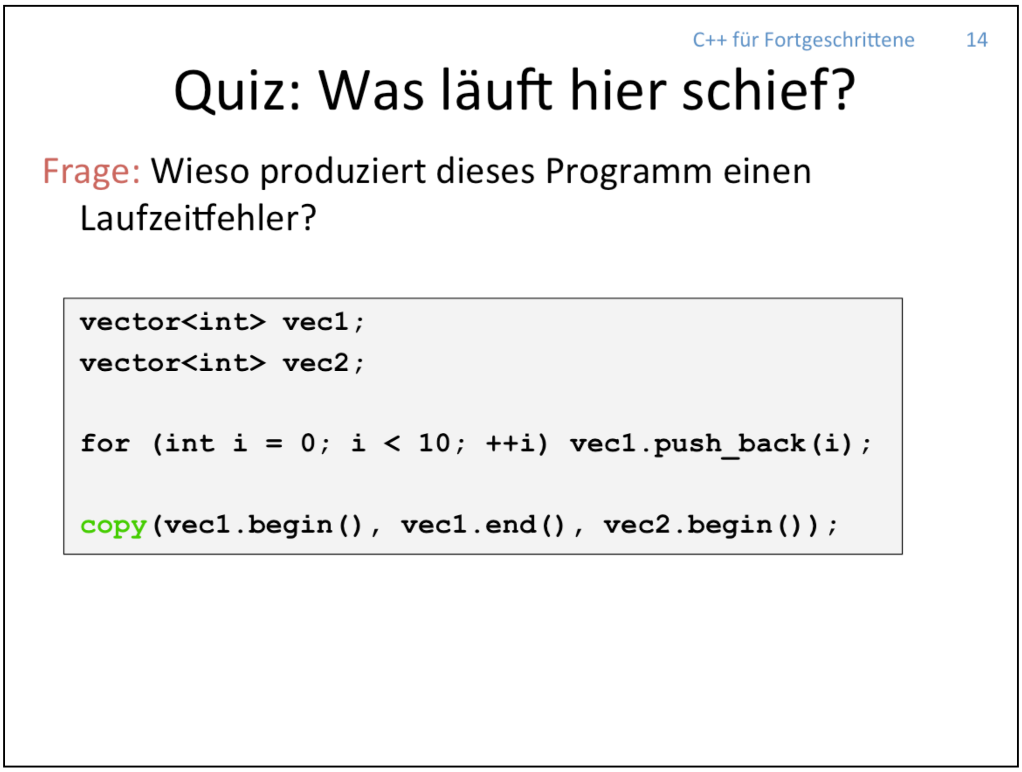


Iteratoren (II)

Mit Iteratoren kann man durch Container traversieren.

```
template<typename TIn, typename TOut>
inline TOut copy(TIn first, TIn last, TOut dest)
{
    while ( first != last )
    {
          *dest = *first;
          ++dest; ++first;
    }
    return (dest);
}
```

Anmerkung: Dies tatsächlich der **copy**-Algorithmus, wie er in der STL definiert wird (bzw. wie man ihn implementieren könnte).



Quiz: Was läuft hier schief?

Frage: Wieso produziert dieses Programm einen Laufzeitfehler?

```
vector<int> vec1;
vector<int> vec2;

for (int i = 0; i < 10; ++i) vec1.push_back(i);

copy(vec1.begin(), vec1.end(), vec2.begin());
```

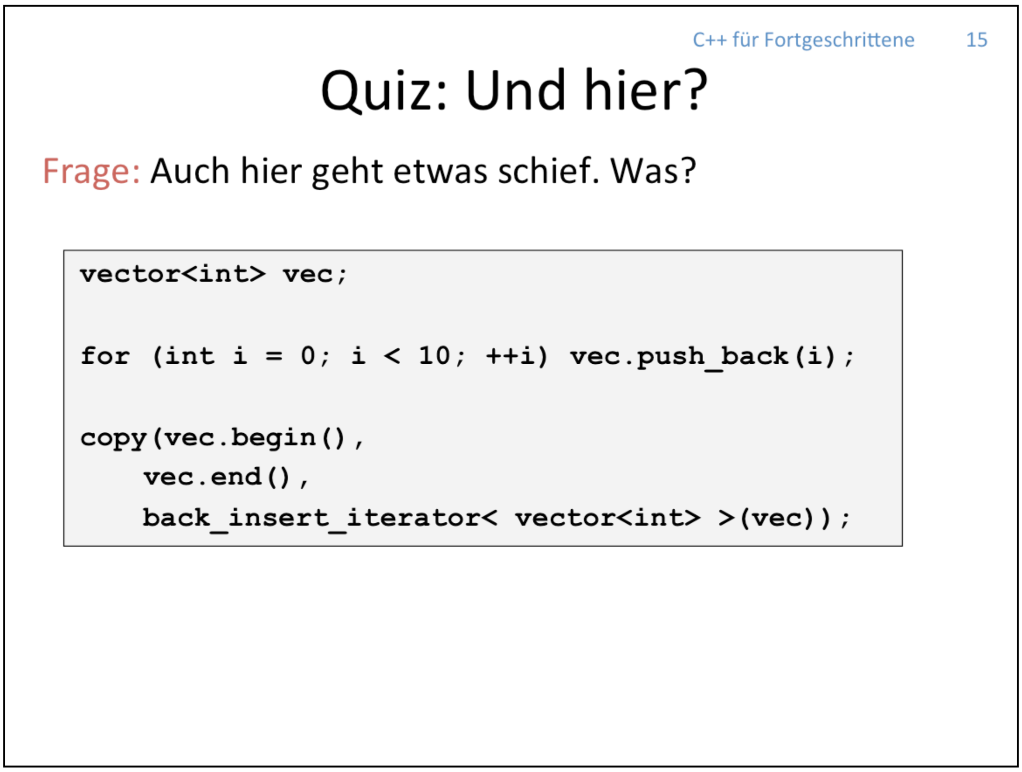
Antwort: vec2 ist nicht groß genug, um alle Elemente von **vec1** aufzunehmen. **copy** prüft dies nicht nach.

Lösung: Statt den einfachen **vec2.begin()**-Iterator als Ziel für das Kopieren zu verwenden, könnte man einen Iterator vom Typ **back_insert_iterator< vector<int> >** benutzen:

copy(vec1.begin(), vec1.end(), back_insert_iterator< vector<int> >(vec2));

Noch besser wäre es allerdings, vor dem Aufruf von **copy** die Größe von **vec2** anzupassen:

vec2.resize(vec1.size());



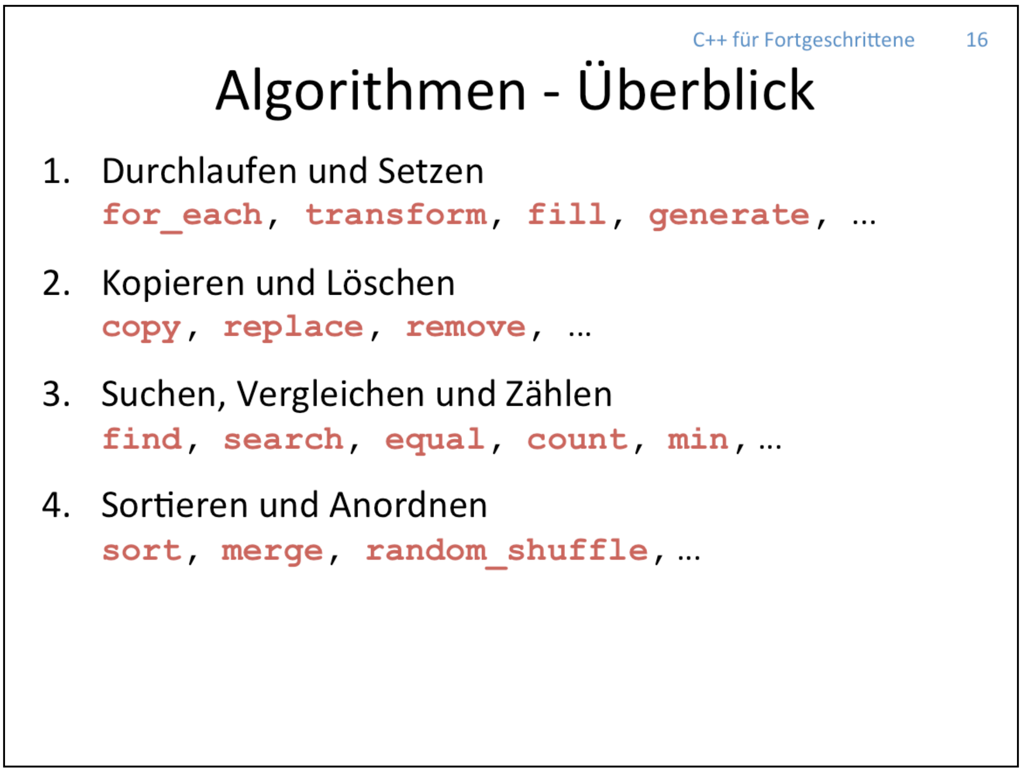
Quiz: Und hier?

Frage: Auch hier geht etwas schief. Was?

```
vector<int> vec;

for (int i = 0; i < 10; ++i) vec.push_back(i);

copy(vec.begin(),
    vec.end(),
    back_insert_iterator< vector<int> >(vec));
```

Antwort: Über den **back_insert_iterator** werden mit **copy** an **vec** neue Element angehängt. Sobald die Kapazität von **vec** nicht mehr ausreicht, muss ein neuer Speicherbereich allokiert, die Elemente von **vec** in den neuen Speicherbereich kopiert, und anschließend der alte Speicher wieder freigegeben werden. Damit werden aber alle alten Iteratoren ungültig: **vec.begin()** und **vec.end()** zeigen nun auf ganz andere Speicherbereiche als zuvor. Somit werden anschließend von **copy** Daten aus dem veralteten Speicherbereich kopiert, d.h. **copy** greift auf einen Speicherbereich zu, der nicht mehr allokiert ist.

Lösung: (Hier wie überall zu empfehlen) Container erst auf die richtige Größe bringen und dann füllen.



Algorithmen - Überblick

1. Durchlaufen und Setzen
   `for_each`, `transform`, `fill`, `generate`, …
2. Kopieren und Löschen
   `copy`, `replace`, `remove`, …
3. Suchen, Vergleichen und Zählen
   `find`, `search`, `equal`, `count`, `min`, …
4. Sortieren und Anordnen
   `sort`, `merge`, `random_shuffle`, …

Praktisch alle Algorithmen arbeiten auf einer oder mehreren Ranges, d.h. Gebieten, die mit Anfangs- und Enditerator angegeben werden.

for_each durchläuft eine Range, und wendet auf jedes Element eine Funktion an.

transform tut das gleiche wie **for_each** und speichert das Ergebnis der Funktion in eine Zielrange.

fill füllt die Range mit einem festen Wert.

generate füllt die Range mit dem Ergebnis einer Funktion.

copy kopiert eine Range in eine Zielrange.

replace ersetzt alle Zeichen einer bestimmten Art innerhalb der Range durch ein anderes Zeichen.

remove löscht alle Zeichen einer bestimmten Art aus der Range. Vorsicht: Die Größe des Containers wird dadurch natürlich nicht herabgesetzt. (Siehe unten)

find sucht ein Zeichen in einer Range.

search sucht eine Range in einer Range.

equal vergleicht zwei Ranges.

count zählt die Zahl der Vorkommen eines Zeichens in einer Range.

min ermittelt das minimum innerhalb der Range.

sort sortiert eine Range.

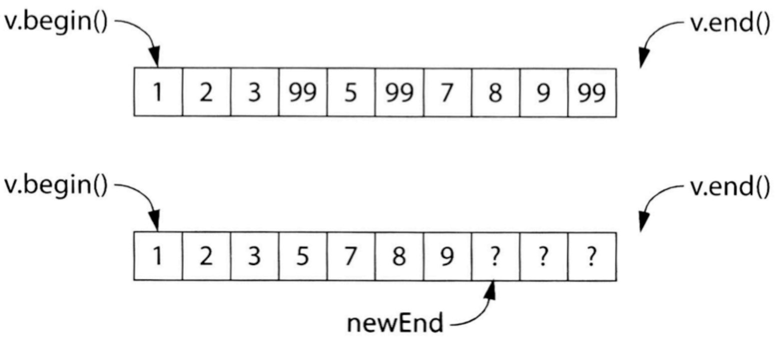
merge verschmilzt zwei sortierte Ranges und schreibt sie in eine andere Range

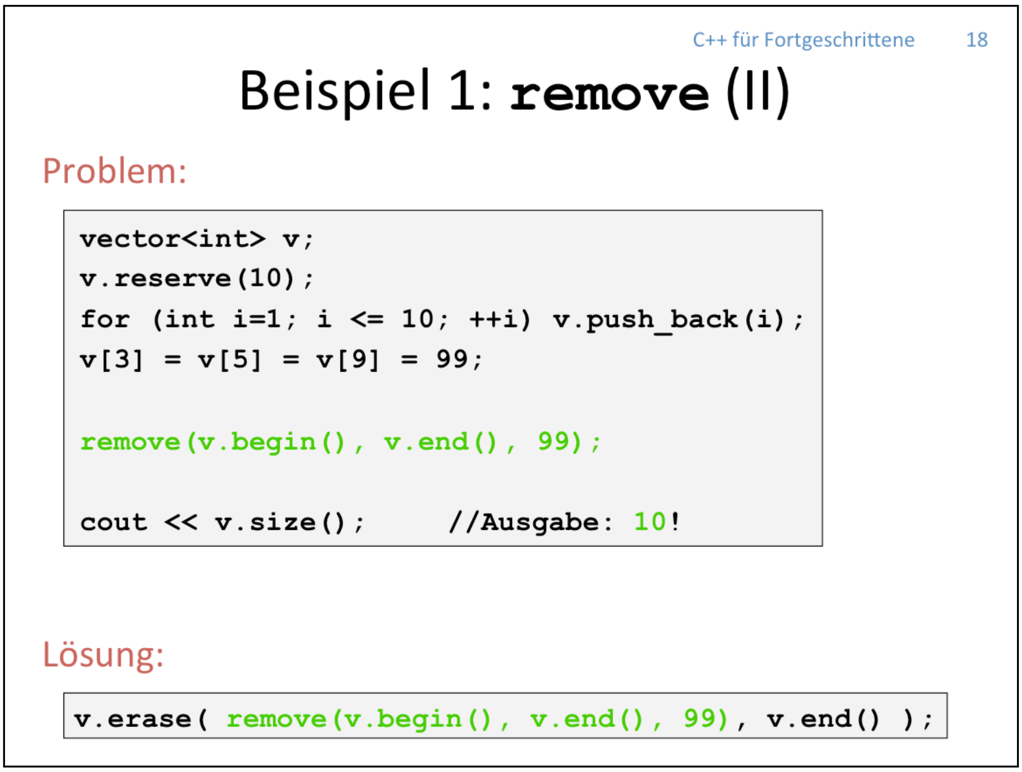
random_shuffle durchmischt die Elemente einer Range.

Beispiel 1: **`remove`**

remove durchläuft eine Range, entfernt alle Elemente einer bestimmten Art daraus und gibt einen Iterator auf hinter das neue Ende zurück.

```
remove(v.begin(), v.end(), 99);
```





Beispiel 1: `remove` (II)

Problem:

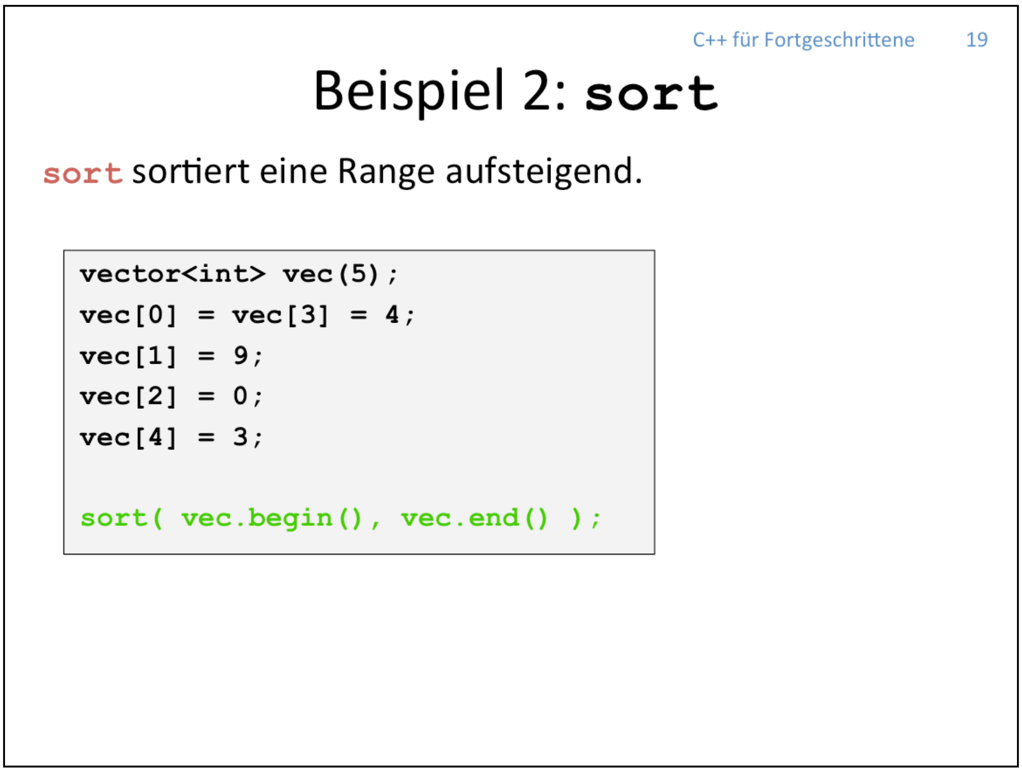
```
vector<int> v;
v.reserve(10);
for (int i=1; i <= 10; ++i) v.push_back(i);
v[3] = v[5] = v[9] = 99;

remove(v.begin(), v.end(), 99);

cout << v.size();     //Ausgabe: 10!
```

Lösung:

```
v.erase( remove(v.begin(), v.end(), 99), v.end() );
```

remove selbst kann die Größe von v nicht verändern: Der Algorithmus kennt nur die Iteratoren, nicht den Container selbst.

Lösung: Der Rückgabewert von **remove** muss dafür verwendet werden, das Ende von v neu zu setzen. Dies geschieht z.B. wie oben gezeigt mit Hilfe der Member-Funktion **erase**.

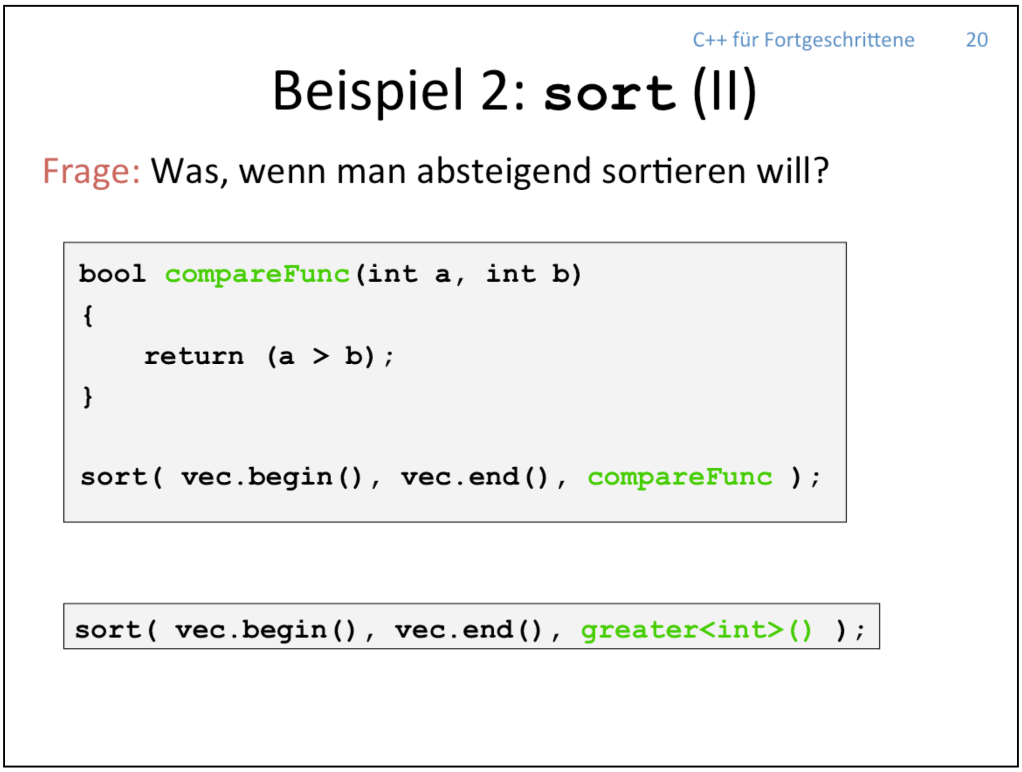


Beispiel 2: `sort`

`sort` sortiert eine Range aufsteigend.

```
vector<int> vec(5);
vec[0] = vec[3] = 4;
vec[1] = 9;
vec[2] = 0;
vec[4] = 3;

sort( vec.begin(), vec.end() );
```

In **vec** steht nun: 0, 3, 3, 4, 9



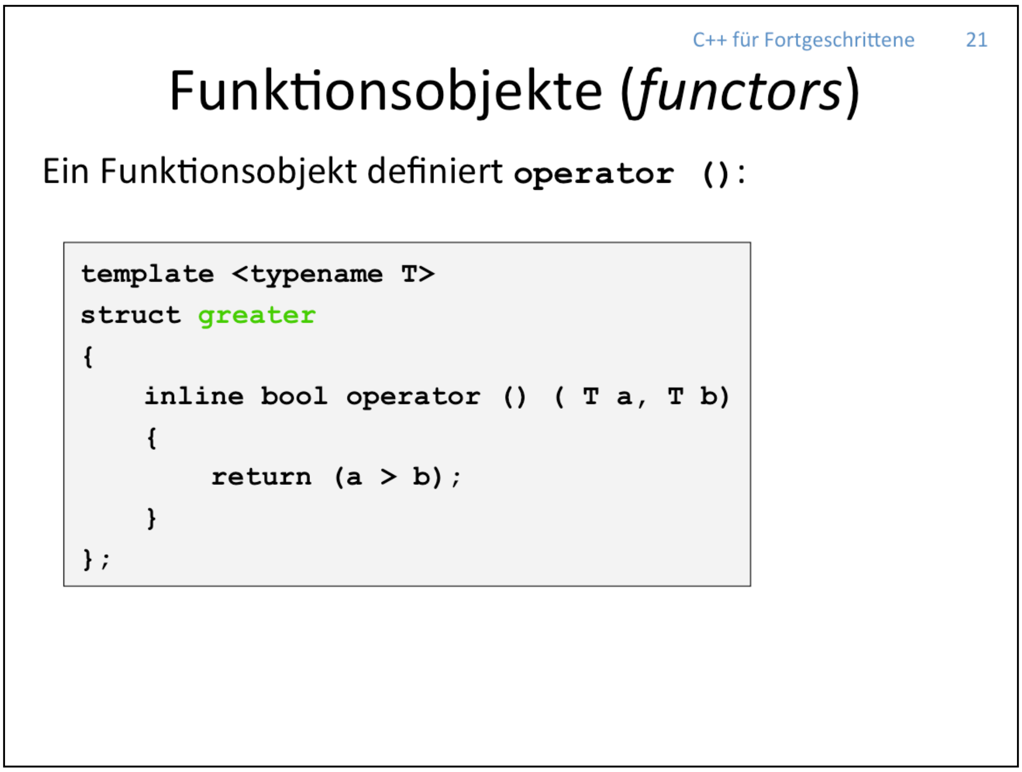
Beispiel 2: `sort` (II)

Frage: Was, wenn man absteigend sortieren will?

```
bool compareFunc(int a, int b)
{
    return (a > b);
}

sort( vec.begin(), vec.end(), compareFunc );
```

```
sort( vec.begin(), vec.end(), greater<int>() );
```

Die Funktionalklasse **greater** ist in <functional> definiert.

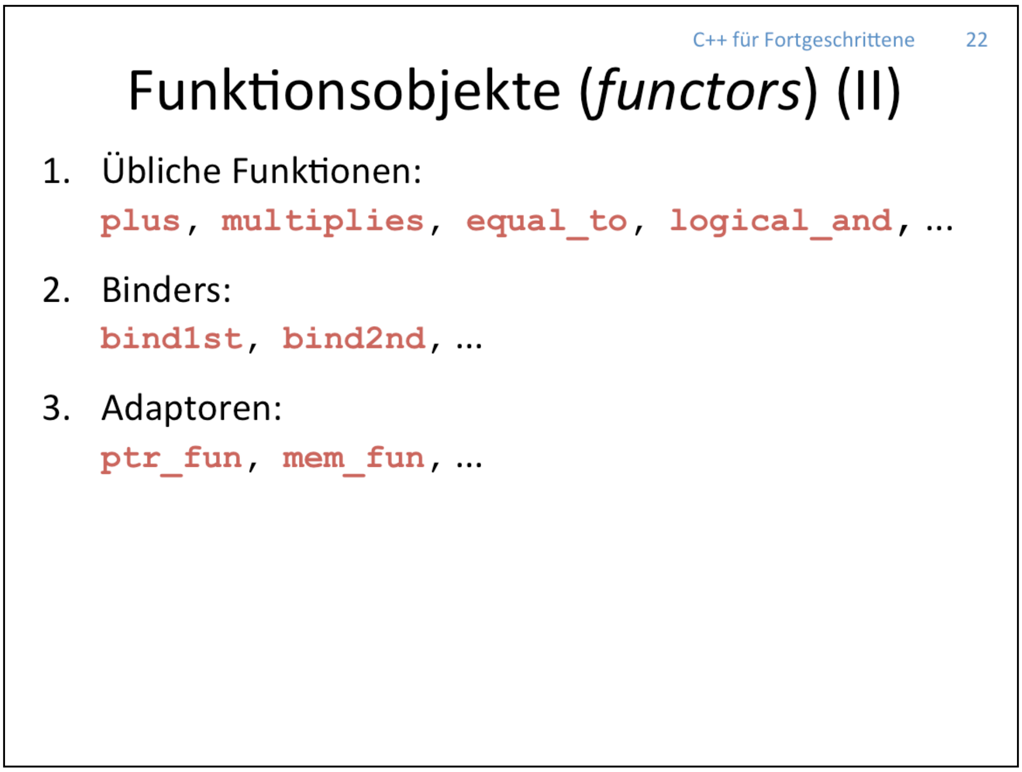


Funktionsobjekte (*functors*)

Ein Funktionsobjekt definiert `operator ()`:

```
template <typename T>
struct greater
{
    inline bool operator () ( T a, T b)
    {
        return (a > b);
    }
};
```

Funktionsobjekte (auch "Funktionale" genannt) haben den Vorteil, dass sie gewöhnlich (anders als Funktionen, die man über einen Funktionspointer übergibt) vom Compiler ge-*inlined* werden können. Darum würde die Sortierung im Beispiel mit **greater<int>** schneller gehen als mit **compareFunc()** (und zwar selbst wenn **compareFunc inline** deklariert worden wäre).



Funktionsobjekte (*functors*) (II)

1. Übliche Funktionen:
   `plus`, `multiplies`, `equal_to`, `logical_and`, …
2. Binders:
   `bind1st`, `bind2nd`, …
3. Adaptoren:
   `ptr_fun`, `mem_fun`, …

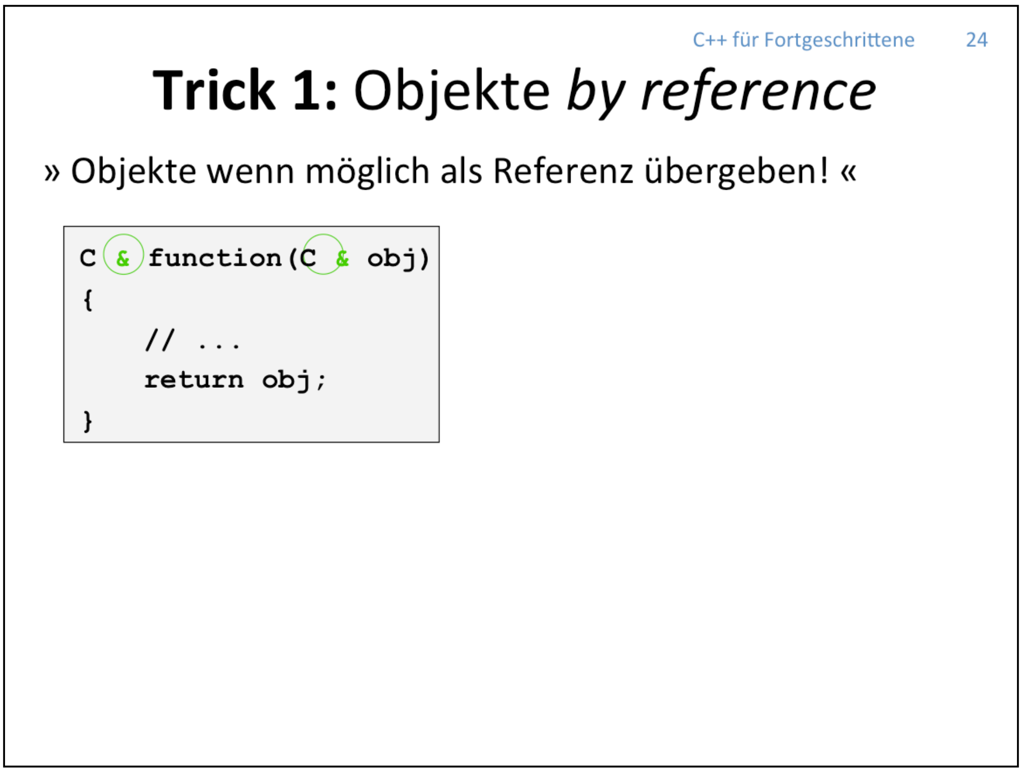
bind1st nimmt ein Funktionsobjekt mit 2 Argumenten und einen Wert x und liefert ein Funktionsobjekt mit 1 Argument, wobei das erste Argument der ursprünglichen Funktion mit x belegt wurde. Entsprechendes gilt für **bind2nd**.

Mit **ptr_fun** wird ein Pointer auf eine Funktion in ein Funktionsobjekt umgewandelt. Auf diese Weise wird es möglich, Funktionen mit Adaptoren für Funktionsobjekte zu verwenden.

mem_fun ermöglicht die Verwendung von Memberfunktionen über Funktionsobjekte.

THEMA 2

TIPS & TRICKS



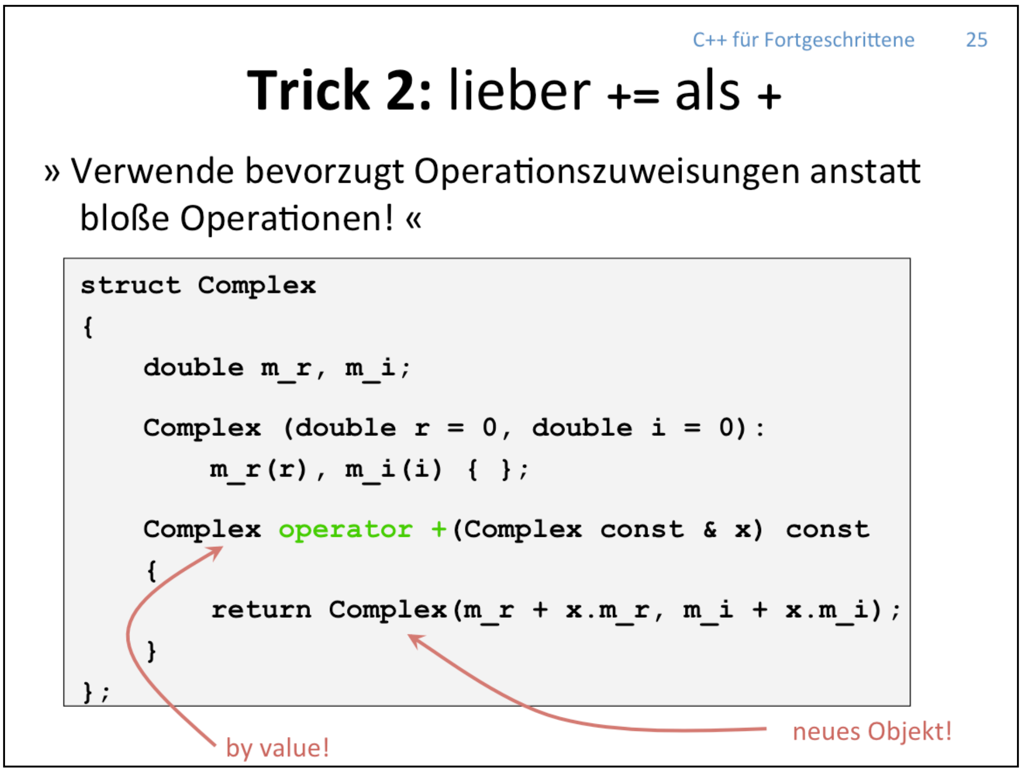
Trick 1: Objekte *by reference*

» Objekte wenn möglich als Referenz übergeben! «

```
C & function(C & obj)
{
    // ...
    return obj;
}
```

Auch Pointer sind möglich. Objekte per Value zu übergeben ist fast immer eine schlechte Idee, da dann auf temporären Kopien gearbeitet wird (und das Anlegen dauert Zeit).

Vorsicht: Wenn Objekte per Referenz zurückgegeben, dann darf es sich dabei nicht um ein Stackobjekt der Funktion handeln, da dieses Objekt ja nach Ende der Funktion zerstört wird. Besser ist es darum, wenn möglich gänzlich auf Objekte als Rückgabewerte zu verzichten. Geben Sie statt dessen die Resultate der Funktion über die Argumente zurück.



Trick 2: lieber += als +

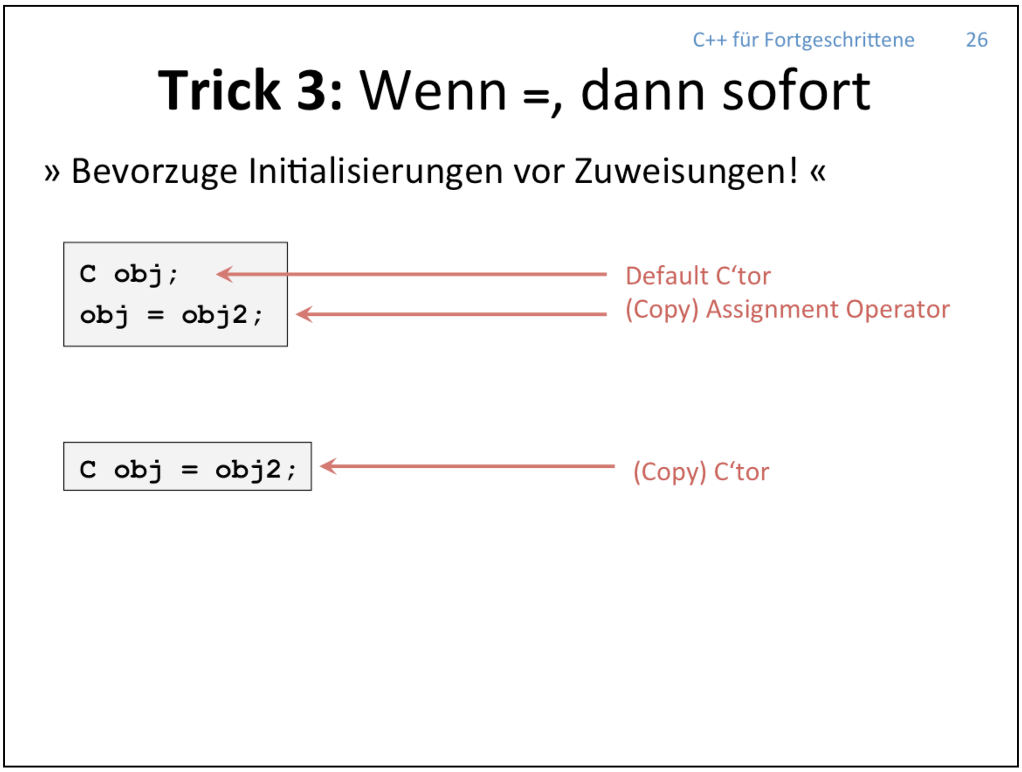
» Verwende bevorzugt Operationszuweisungen anstatt bloße Operationen! «

```
struct Complex
{
    double m_r, m_i;

    Complex (double r = 0, double i = 0):
        m_r(r), m_i(i) { };

    Complex operator +(Complex const & x) const
    {
        return Complex(m_r + x.m_r, m_i + x.m_i);
    }
};
```

by value!

neues Objekt!

Bei der Definition eines **operator +** bleibt einem nichts anderes übrig, als ein neues Objekt by-value zurückzugeben. Beim **operator +=** wäre dies nicht notwendig.



Trick 3: Wenn =, dann sofort

» Bevorzuge Initialisierungen vor Zuweisungen! «

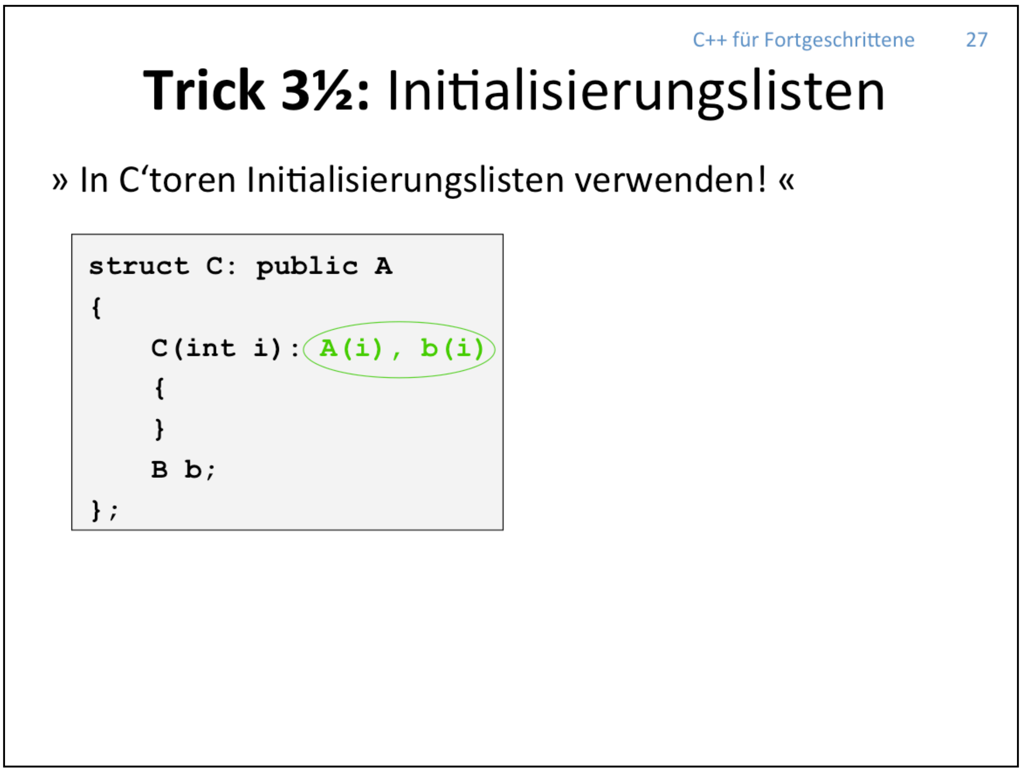
```
C obj;
obj = obj2;
```

Default C‘tor
(Copy) Assignment Operator

```
C obj = obj2;
```

(Copy) C‘tor

Grund: Man spart auf u.U. einen Funktionsaufruf, der Code ist kleiner und Übersichtlicher. Außerdem werden Initialisierungen, die im Default C‘tor vorgenommen werden, umsonst getätigt.

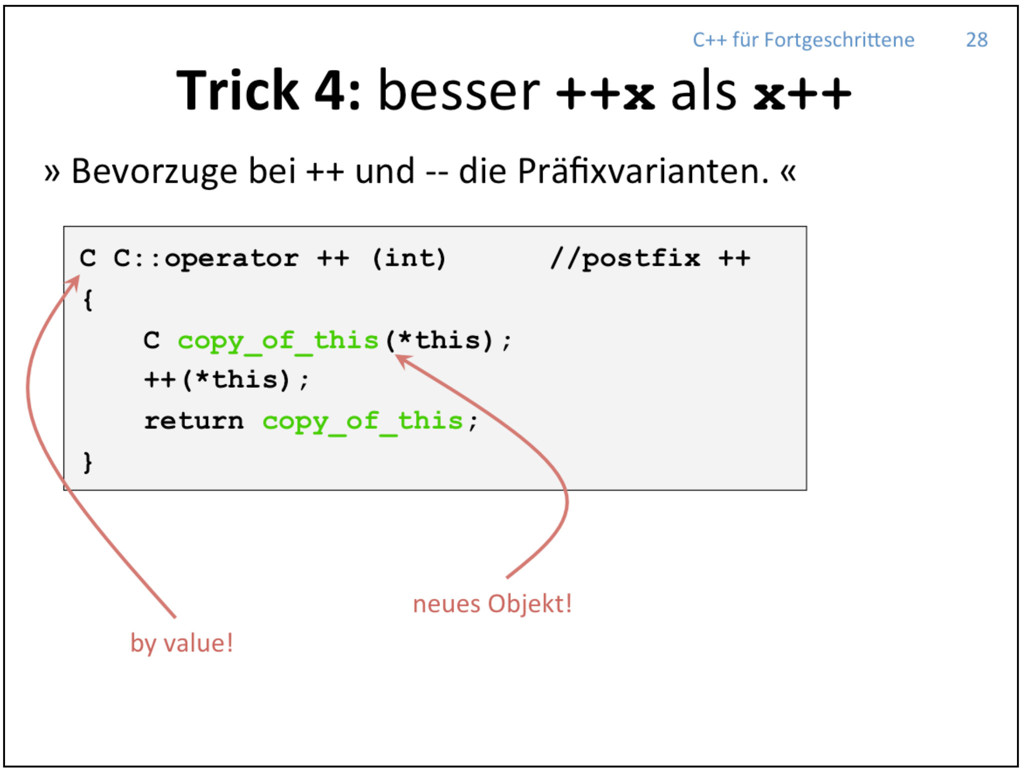


Trick 3½: Initialisierungslisten

» In C‘toren Initialisierungslisten verwenden! «

```
struct C: public A
{
    C(int i): A(i), b(i)
    {
    }
    B b;
};
```

Ohne Initialisierungslisten werden die Basisobjekte und die Member-Objekte zunächst mit Default C‘toren erzeugt. Dabei werden u.U. die Member dieser Objekte mit Werten initialisiert, die man gleich danach wieder überschreiben müsste. Bei Initialisierungslisten werden hingegen gleich die passenden C‘toren aufgerufen.



Trick 4: besser **++x** als **x++**

» Bevorzuge bei ++ und -- die Präfixvarianten. «

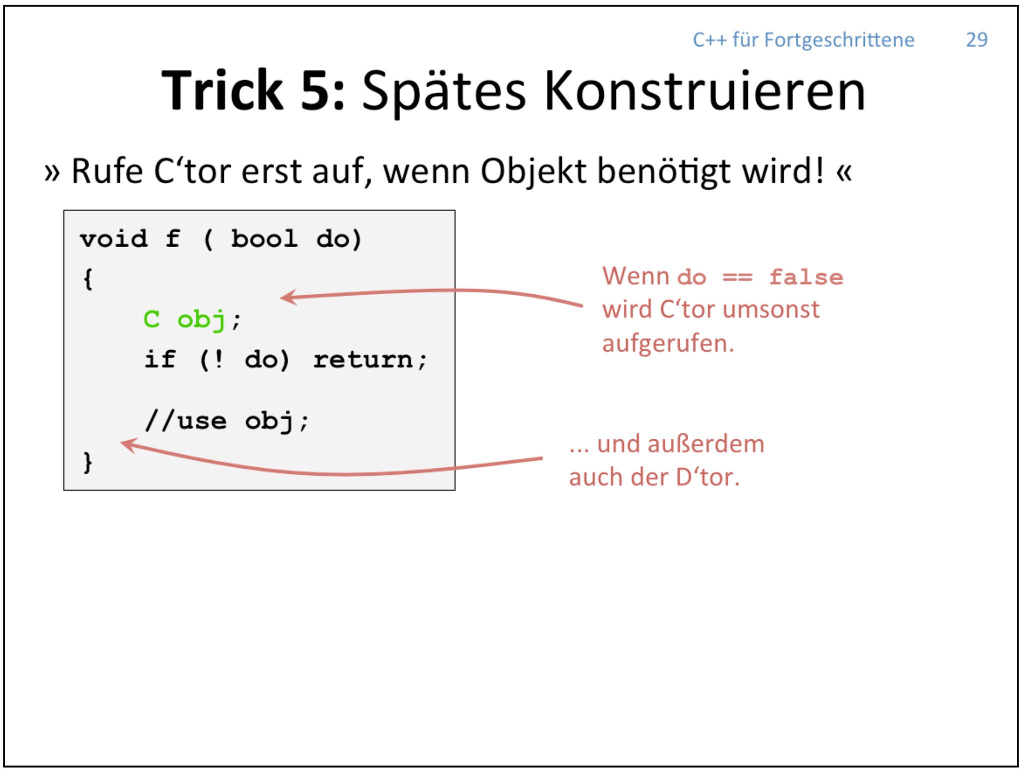
```
C C::operator ++ (int)      //postfix ++
{
    C copy_of_this(*this);
    ++(*this);
    return copy_of_this;
}
```

neues Objekt!

by value!

Durch das Argument „**int**“ wird angezeigt, dass es sich hier um die Definition eines Postfix-Increments handelt.

Bei **++obj** kann **obj** sich selbst als Rückgabewert liefern, bei **obj++** ist das nicht der Fall.



C++ für Fortgeschrittene 29

Trick 5: Spätes Konstruieren

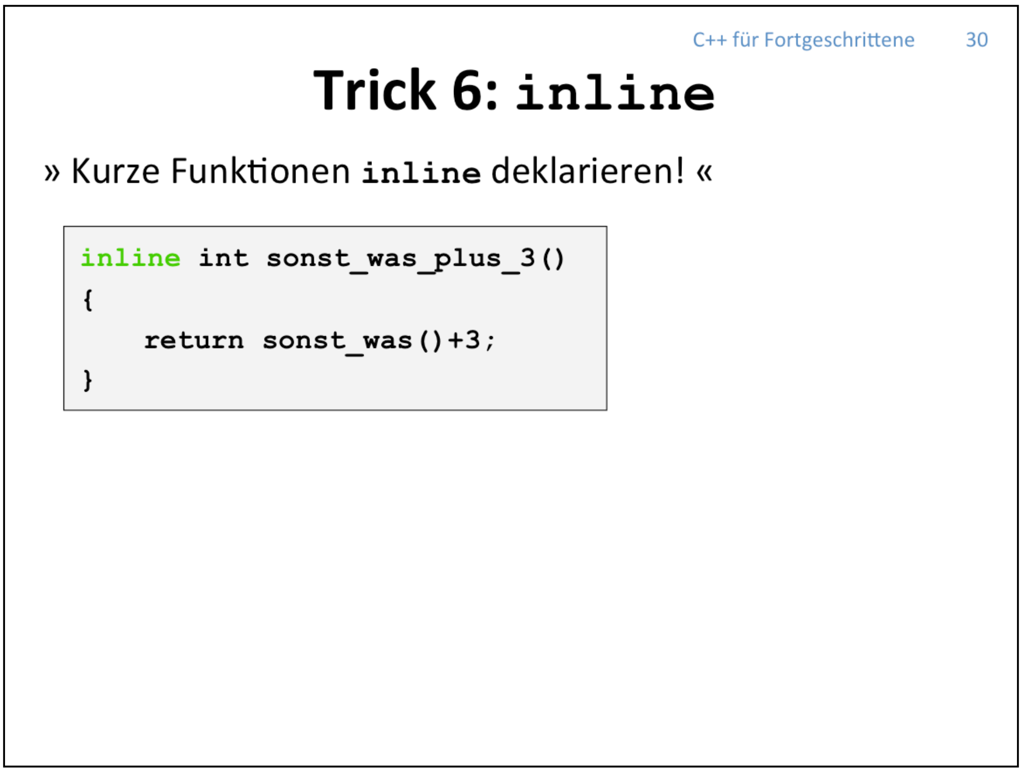
» Rufe C‘tor erst auf, wenn Objekt benötigt wird! «

```
void f ( bool do)
{
    C obj;
    if (! do) return;

    //use obj;
}
```

Wenn `do == false` wird C‘tor umsonst aufgerufen.

... und außerdem auch der D‘tor.

Vorsicht: Dabei sollte man natürlich nicht so weit gehen, die Konstruktion eines Objekts z.B. in eine Schleife zu verlagern, da dann bei jedem Schleifendurchlauf ein neues Objekt erzeugt und am Ende wieder zerstört werden würde.

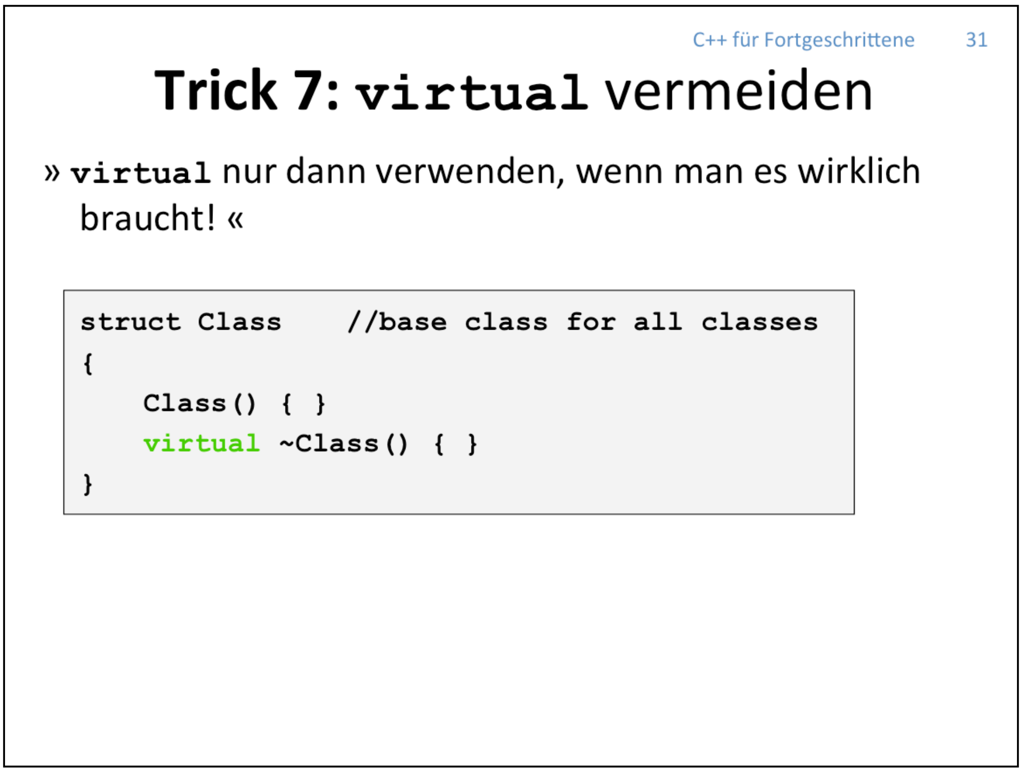


Trick 6: `inline`

» Kurze Funktionen **`inline`** deklarieren! «

```
inline int sonst_was_plus_3()
{
    return sonst_was()+3;
}
```

Bei Funktionen, die ge-inlined werden, entfällt der Overhead für den eigentlichen Funktionsaufruf vollkommen. Dadurch wird jedoch u.U. zusätzlicher Code erzeugt.

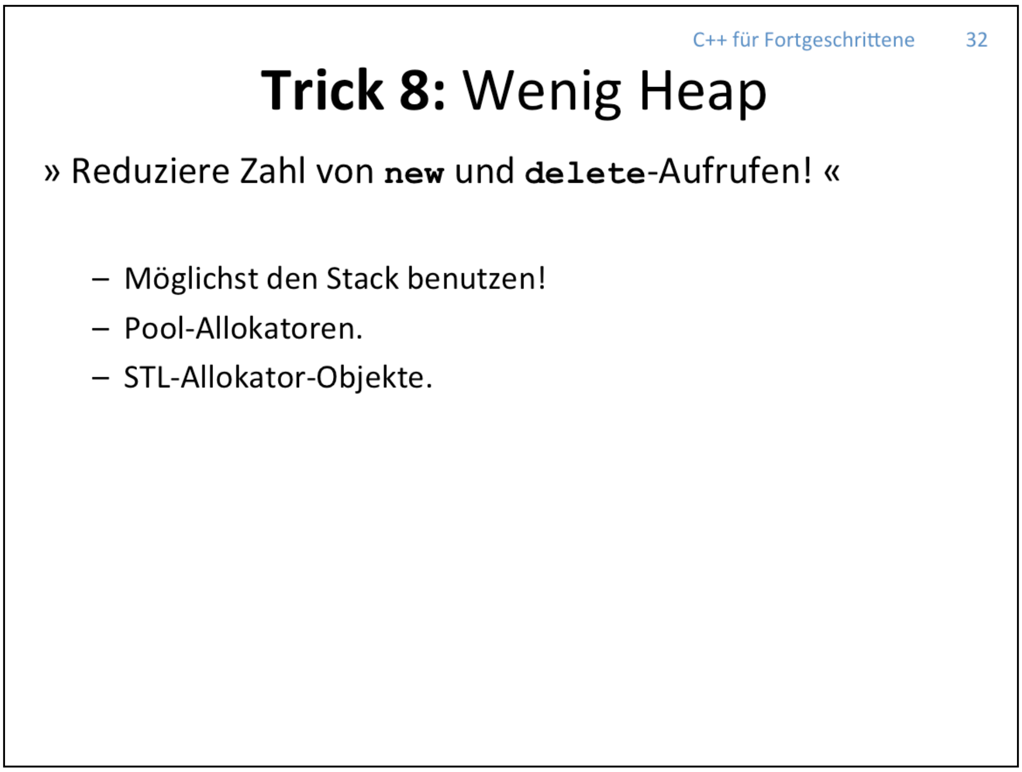


Trick 7: `virtual` vermeiden

» **`virtual`** nur dann verwenden, wenn man es wirklich braucht! «

```
struct Class    //base class for all classes
{
    Class() { }
    virtual ~Class() { }
}
```

Die hier gezeigte Klasse ist eine Art *worst case* für ein Klassendesign und kann sozusagen als abschreckendes Beispiel gelten.

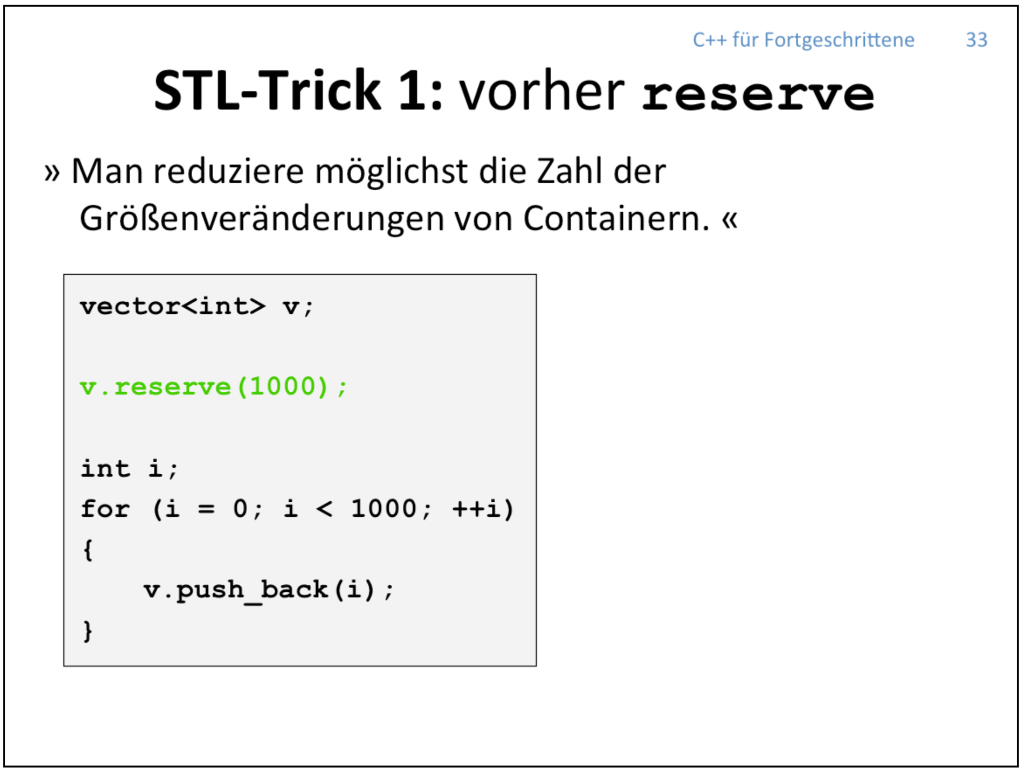


Trick 8: Wenig Heap

» Reduziere Zahl von `new` und `delete`-Aufrufen! «

- Möglichst den Stack benutzen!
- Pool-Allokatoren.
- STL-Allokator-Objekte.

Gute Implementierungen der STL stellen für die Allokation kleiner Speicherblöcke bereits einen eingebauten Pool-Allokator zur Verfügung. (Mehr zu Pool-Allokatoren findet sich in Skript Nr. 3)



STL-Trick 1: vorher `reserve`

» Man reduziere möglichst die Zahl der Größenveränderungen von Containern. «

```
vector<int> v;

v.reserve(1000);

int i;
for (i = 0; i < 1000; ++i)
{
    v.push_back(i);
}
```

Das ist natürlich nur bei Containern nötig, die ihre Elemente in zusammenhängenden Speicherblöcken halten, also insbesondere bei vector oder deque.

Im Beispiel würde es sich natürlich ganz besonders anbieten, **v** bereits im C‘tor auf die richtige Größe zu bringen.

STL-Trick 2: Memberfunktionen

» Bevorzuge Memberfunktion gegenüber gleichnamigen Algorithmus! «

```
set<int> s;
set<int>::iterator i;
...

i = s.find(123);

i = find(s.begin(), s.end(), 123);
```

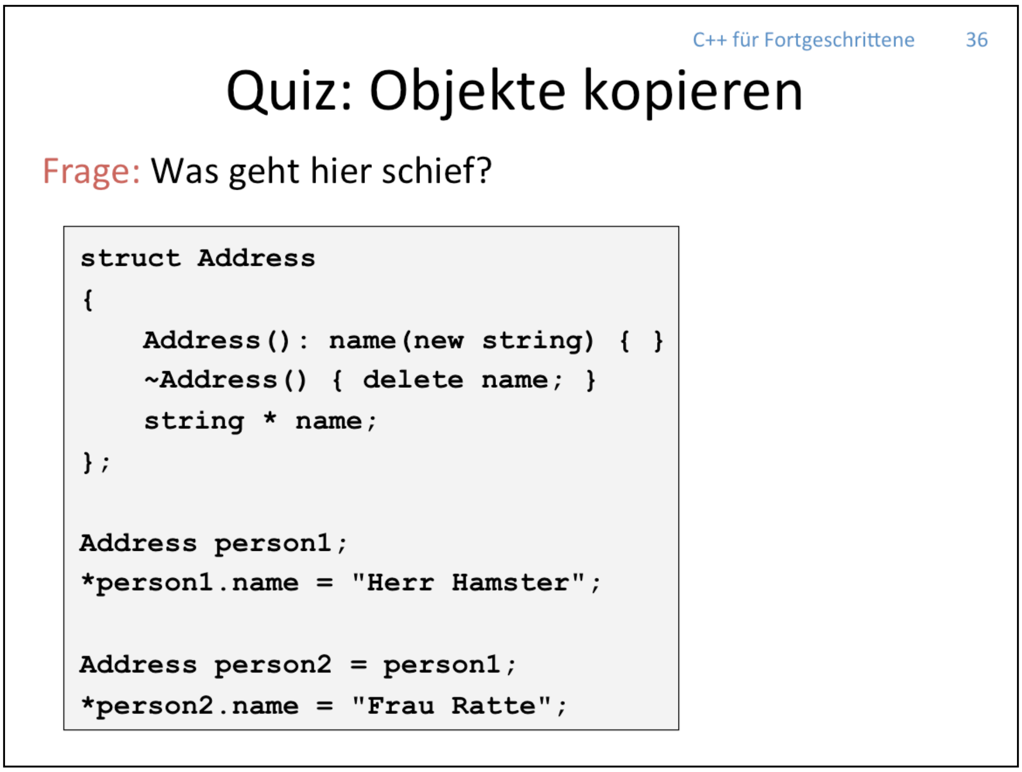
Lauzeit O(log n)

Lauzeit O(n)

STL-Trick 3: Container wählen

» Wähle den richtigen Container aus! «

- Wo neue Elemente hinzufügen?
- Welche Iteratoren benötigt?
- Elemente geordnet?
- ...



Quiz: Objekte kopieren

Frage: Was geht hier schief?

```
struct Address
{
    Address(): name(new string) { }
    ~Address() { delete name; }
    string * name;
};

Address person1;
*person1.name = "Herr Hamster";

Address person2 = person1;
*person2.name = "Frau Ratte";
```

Address hat keinen korrekten Copy-C‘tor. Die beiden Objekte teilen sich ein gemeinsames **string**-Objekt, und das kann fatale Folgen haben. Spätestens beim Aufruf der Destruktoren tritt das Problem zu Tage, dann wird nämlich versucht, den String doppelt freizugeben.

Regel: Man definiere für Klassen stets **die Großen Vier**: Default C‘tor, Copy-C‘tor, Copy Assignment-Operator und D‘tor.

Aufgaben zum Freitag:

1. Aufgabe: (Prüfungsaufgabe)

Schreiben Sie einen generischen Algorithmus **removeRepeats()** im Stil der STL, der aus einem Container diejenigen Elemente löscht, die ihrem Vorgängerelement im Container gleichen. Wie kann man dafür sorgen, dass der Container anschließend verkleinert wird?

Testen Sie ihre Funktion sowohl auf **std::vector** als auch auf **std::list.**

2. Aufgabe:

Gegeben eine Reihe von Datumsangaben in einem vorgegebenen Format, z.B. "13.4.2007". Sortieren Sie ein Array solcher Datumsangaben mit Hilfe des **std::sort**.- Algorithmus.

3. Aufgabe:

Dijkstra: Implementieren sie den Dijkstra Algorithmus zur Berechnung eines kürzesten Pfades von einem Startknoten zu allen anderen Knoten in einem Graphen ohne negativen Kantengewichten. Die Implementierung des Graphen können Sie frei wählen (Adjazenzlisten, Adjazenzmatrix ...).

Sie sollte mindestens das Hinzufügen von Knoten und Kanten unterstützen.

Eine Beschreibung des Dijkstra Algorithmus mit Pseudocode finden sie z.B. bei Wikipedia. Verwenden Sie die heap Funktionen aus den STL alglorithms (make_heap, push_heap, pop_heap...) um die im Algorithmus verwendete Prioritätswarteschlange zu implementieren.

4. Aufgabe

Gehen Sie sicher, dass sie alle 4 Prüfungsaufgaben komplett und lauffähig haben und diese morgen vorstellen können.