Betriebssysteme
Studiengang Informatik

- Dipl.-Inf., Dipl.-Ing. (FH) Michael Wilhelm
- Hochschule Harz
- FB Automatisierung und Informatik
- mwilhelm@hs-harz.de
- Raum 2.202
- Tel. 03943 / 659 338

Gliederung

1. **Einführung**
2. Prozesse und Threads
3. Speicherverwaltung
4. Dateiverwaltung
5. JNI
6. STL
7. Deadlocks

8. Standard Template Library

Eigenschaften:

- Entwickelt von Stepanow und Lee
- STL ist eine template-basierte Klassenbibliothek für generische Datenstrukturen und Algorithmen.
- Sie ist compilerunabhängig, portabel und effizient.

Sie unterteilt sich in folgende Bereiche:

- Container (Vector, Listen, Stack, Map)
- Algorithmen (find, count, swap, for_each, shuffle, sort)
- Iteratoren (forward, backward, for(i=…)
- Adaptoren (Funktionsparameter, less …)
- String
- Streams
- Function Objects

Weitere Eigenschaften

- Jeder Algorithmen kann mit jedem Container, Vector …, verknüpft werden
- Generische Programmierung hat nichts mit objekt-orientierter Programmierung zu tun.
- Die Klassen können aber vererbt werden
- Kompilerunabhängig, effizient, portabel

Container

- vector
- deque
- list
- stack
- queue

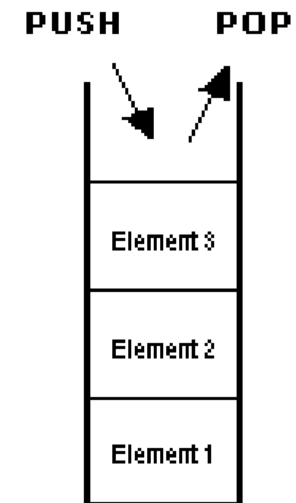
Container

- map
- set
- multimap
- multiset
- bitsets

Stacks (Stapel)

- Ein Stack ist, wie der Name schon verrät, eine Art Stapel, d.h. man kann Elemente nur nach dem LIFO-Prinzip (Last in, First out) von oben wegnehmen (pop) bzw. drauflegen (push).

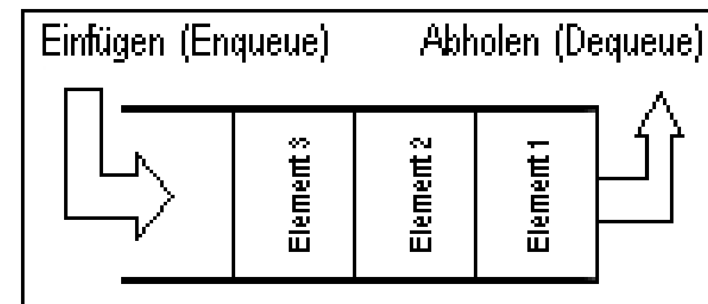
- Im Klartext heißt das: Ein Stack ist eine Liste, zu der man Elemente nur vorne einfügen kann und nur von vorne wegnehmen kann. Die einzigen Methoden sind:
  - push (drauflegen)
  - pop (wegnehmen)
  - get (das oberste Element anschauen)



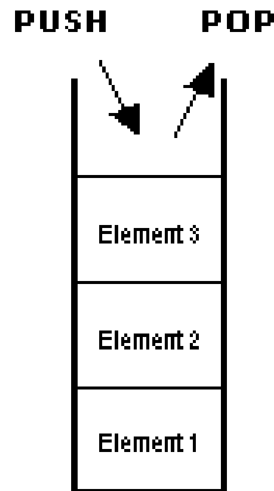
Warteschlange (engl. Queue)

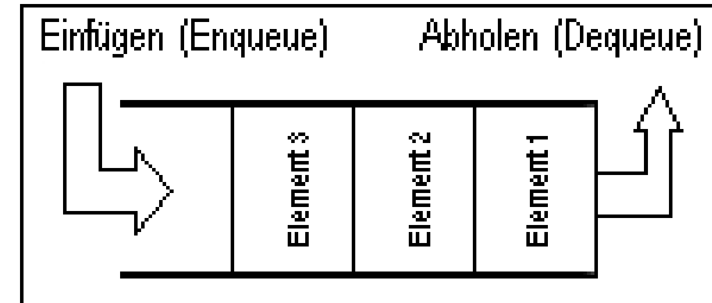
- Ein Stack ist eine LIFO (Last in, first out) Datenstruktur. Im Gegensatz hierzu handelt es sich bei einer Warteschlange, Queue, um eine FIFO (first in, first out) Datenstruktur. Auch diese kann effizient mittels einer verketteten Liste implementiert werden:

```
public void add (T element) {
    if (head != null) {
        Node<T> n = head;
        while ( n.next != null ) n = n.next;
        n.next = new Node<T>();
        n = n.next;
        n.data = element;
    } else {
        head = new Node<T>();
        head.data = element;
    }
}
public T remove() {
    T element = null;
    if ( head != null ) {
        element = head.data ;
        head = head.next;
    }
    return element;
}
```



Unterschied Stack vs. Queue





Eingetragen:	**Entfernt:**	**Eingetragen:**	**Entfernt:**
1 (1. Eintrag)	44	1 (1. Eintrag)	1
66	33	66	66
33	66	33	33
44	1	44	44

Spezielle Warteschlange: Deque

- Deque = (Stack und Queue)
  - Eine Deque ist eine Warteschlange bei der Elemente sowohl am Anfang wie am Ende entfernt und hinzugefügt werden können.
  - Sie wird am einfachsten mittels einer doppelt verketteten Liste implementiert.
- Java
  - LinkedList implementiert die Deque Schnittstelle.

Eingetragen:	**Entfernt:**	**Entfernt:**
1 (1. Eintrag)	1	44
66	66	1
33	33	66
44	44	33

Algorithmen der STL

- Lineares Suchen (find, find_if)
- Zählen (count, count_if)
- Der Vergleich von Bereichen
- Suche nach Teilfolgen
- Minimum
- Maximum
- Elemente vertauschen (swap)
- Kopieren von Bereichen
- Elemente transformieren und ersetzen
- Elemente entfernen
- Permutation
- Bereiche sortieren
- Binäres Suchen
- Mischen in sortieren Bereichen
- Mengenoperationen auf sortierte Bereiche
- Heap-Operationen
- Numerische Operationen

Beispiel:

Procedure Tausche

```
void change( int& a, int& b) {
  int h = a;
  a = b;
  b = h;
} //
void change( double& a, double& b) {
  double h = a;
  a = b;
  b = h;
} //
```

Problem:

- Die Funktionsweise ist für beide Methoden identisch.
- Nur der Speicherbereich für die Datentypen ist unterschiedlich.
- Für komplexere Funktionen ist der „Kopieraufwand“ erheblich und fehlerträchtig!
- Mit einem „Funktions-Template“ definiert man die Funktion nur einmal!
- Einem Template werden die Datentypen als Parameter übergeben.
- Ein Template kann also nur mit einem Datentyp, Klasse arbeiten.
- Der Kompiler erzeugt aus den Templates und den Datentypen die Methoden.

Beispiel für die Definition


```
template <class T>
inline void change( T & a, T &b) {
   T h = a;
   a = b;
   b = h;
}
```

Aufruf:

```
int i1=1,  i2=2;
change(i1,i2);

string s1, s2;
change(s1,s2);
```


Bsp1.cpp

Vollständiger Quellcode: bsp1.cpp

```
#include "stdafx.h"
#include <iostream>
#include <string>
using namespace std;

template <class T>
inline void change( T& a, T& b) {
        T h = a;
        a = b;
        b = h;
}
```

Vollständiger Quellcode: bsp1.cpp

```
int main(int argc, char* argv[]) {
        int i1=1,i2=2;
        string s1, s2;

        change(i1,i2);
        cout << "i1: " << i1 << endl;
        cout << "i2: " << i2 << endl;
        s1 = "ABCD";
        s2 = "XYZ";
        change(s1,s2);
        cout << "s1: " << s1.c_str() << endl;
        cout << "s2: " << s2.c_str() << endl;

        return 0;
}
```

Klasse string

- Gehört nicht zur STL, aber zur C++ Standardbibliothek
- Vereinfacht die Handhabung mit Charakter-Feldern
  - strcpy
  - strcat
  - strncpy
  - delete []
- Eigenschaften
  - Operator-Overloading (+,=, +=)
  - Dynamische Speicherverwaltung (intern)
  - Pointer zum Array (c_str)
  - find / length, substr, size, insert, erase, replace, compare
  - find_first_of, find_last_of, find_first_not_of, find_last_not_of

```
void test2() {
        char text[] = "Dies ist eine kurzer Text";
        // Anfangsadresse, EndAdressse  [)
        string s1(text, text+sizeof(text)-1);

        string s2 = s1;     string s3 = s1;     string s4 = s1;     string s5 = s1;

        reverse( s2.begin()+2, s2.end()-1 );
        transform( s3.begin(), s3.end(), s3.begin(),  Upper() );
        replace( s4.begin(), s4.end(), 'e','E' );
        sort( s5.begin(), s5.end() ); // benötigt Vergleichsoperator, wo ist er ?

        cout << "s1: " << s1.c_str() << endl;
        cout << "s2: " << s2.c_str() << endl;
        cout << "s3: " << s3.c_str() << endl;
        cout << "s4: " << s4.c_str() << endl;
        cout << "s5: " << s5.c_str() << endl;
}  // test2
```

Beispiel für den Container Vector (Einfügen am Ende)

```
#include <vector>
using namespace std;
typedef vector < int > INTVector;                    // Definition eines int-Vektors

void test1() {
   int i;
   INTVector vect;              // 1. Version
    vector < int > vect;        // 2. Version

   vect.push_back(1);  // ans Ende einfügen
   vect.push_back(2);

   // Ausgabe der Elemente
   cout << "Ausgabe der Elemente for-Schleife" << endl;
   for (i=0; i<vect.size(); i++ ) {
          cout << vect[i];
   }  // for
```

Beispiel für den Container Vector (Einfügen am Ende)

```
void test2() {
  int i;

   // Beispiele für die Definition
  Seite s1(1);       // Seite ist eine Klasse
  Seite s2(2);
  Seite s3(3);
  Seite s4(4);

  Seite *ps;  // Pointer auf eine Seite
   SeitenVector vect;

   cout << endl << endl <<  "Testroutine  test2()" << endl;
```

Beispiel für den Container Vector (Einfügen am Ende)

```
// Einfügen am Ende
vect.push_back(s1);
vect.push_back(s2);

// Einfügen vor dem ersten Element
// insert(p,x)  =>  x wird vor p eingefügt
vect.insert( vect.begin(), s5 );        // Am Anfang einfügen
vect.insert( vect.begin(), s6 );        // Am Anfang einfügen

vect.insert( vect.begin()+2, s7 ); // ?????????
```

Beispiel für den Container Vector (Einfügen am Ende)

```
// Ausgabe der Elemente
cout << "Ausgabe der Elemente for-Schleife (mit Pointer)" << endl;
for (i=0; i<vect.size(); i++ ) {
        ps = & vect[i];
        ps = & vect.at(i);
        ps->print();
}  // for
cout << endl << endl;

// Ausgabe der Elemente als Iterator
SeitenVector::iterator cur;    // Iterator für die Schleife
cout << "Ausgabe der Elemente for-Schleife (Iterator mit Pointer)" << endl;
for (cur=vect.begin(); cur != vect.end(); ++cur ) {
         // in cur steht das aktuelle Element als Pointer
        cout << "Iterator-Forschleife: " << cur->getId() << endl;
}
```

Beispiel für den Container Vector (Löschen)

```
// Löschen von Elementen
  // löscht das 3. Element
if ( ! vect.empty() ) {
      vect.erase(vect.begin() + 2);

      // Ausgabe der Elemente als Iterator
      cout << "Ausgabe der Elemente nach dem Löschen" << endl;
      for (cur=vect.begin(); cur != vect.end(); cur++ ) {
          // in cur steht das aktuelle Element als Pointer
        cout << "Iterator-Forschleife: " << cur->getId() << endl;
      }
}
cout << endl << endl;
```

Beispiel für den Container Vector (Löschen)

```
        // Löschen aller Elemente
        vect.erase( vect.begin(), vect.end() );

        // Löschen der Elemente, von 0 bis n-4
        vect.erase( vect.begin(), vect.end()-4 );
    }
    // Weitere Funktionen des Template Vector
    // clear
    // resize
    // empty
```

Sequenzanfordungen der Container (Iterator)

	vector	deque	list	stack
front	✓	✓	✓	push
back	✓	✓	✓	pop
push_back	✓	✓	✓	
pop_back	✓	✓	✓	
push_front		✓	✓	
pop_front		✓	✓	
at	✓	✓		
operator []	✓	✓		

Sequenzanfordungen der Container

- operator [] nimmt keine Bereichsüberprüfung vor
- at nimmt eine Bereichsüberprüfung vor
  - vect.at(0);
- Einfügen am Anfang
  - v.push_front(item);
  - a.insert(a.begin(), item);
- Löschen eines Elements am Anfang
  - v.pop_front(item);
  - a.erase(a.begin());
- Eigenschaften der Container
  - Vector: Einfügen optimal am Ende
  - Deque: Einfügen optimal am Anfang und Ende, Performance etwas schlechter als Vector
  - List: Einfügen überall optimal, kein Indexzugriff

Sequenzanfordungen der Container

- Vector
  - front, back Pointer
  - insert(iterator)
  - push_back (insert)
  - pop_back
  - at, operator[]
- Deque
  - front, back Pointer
  - insert(iterator)
  - push_front (insert)
  - pop_front
  - push_back (insert)
  - pop_back
  - at, operator[]
- List
  - front, back Pointer
  - insert(iterator)
  - erase
  - push_front (insert)
  - pop_front
  - push_back (insert)
  - pop_back

Sequenzanfordungen der Container

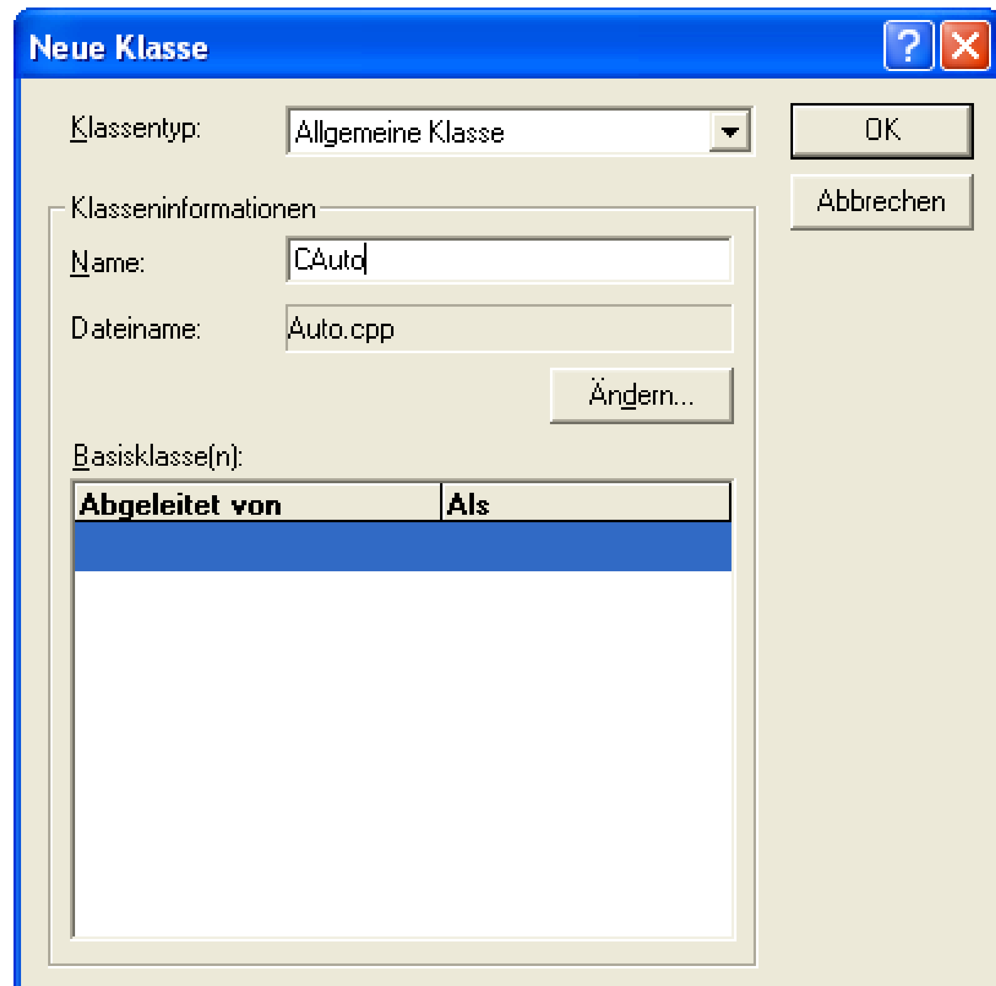
- Stack
  - push (insert Anfang)
  - pop
  - top
- Queue
  - push (insert Anfang)
  - back
  - front
  - pop
- priority_queue
  - push (insert Mitte)
  - top
  - pop

Beispiel mit einer eigenen Klasse

- Erzeugen eines Projektes
- Definition der Klasse
- import der Headerdateien
- Definition
  - Definition des Konstruktors
  - Definition des Copy-Konstruktors
  - Definition des = Operators
  - bool operator() <   // sort
  - bool operator==   // sort, map, find
- Methoden einbauen
- Tests in main

Klasse CAuto

- Names des Projektes: bsp2
- Klassenname: CAuto
- Attribut: int ps;



Klasse CAuto: Auto.h

```
#include <iostream>
using namespace std;
class CAuto {
public:
        CAuto();
        CAuto(int ps);
        virtual ~CAuto();
        CAuto(const CAuto &a);
        virtual const CAuto& operator = (const CAuto& a);
        bool operator<(const CAuto &a) ;    // sort
                // Vergleichsoperator für find_end, find_first_of
        bool operator() (const CAuto &a1,const CAuto &a2) const;
        bool operator==(const CAuto &a) {                   // find
          return ps = = a.ps;
        }
        void print();
        int ps;
}
```

Klasse CAuto: Auto.cpp

```
CAuto::CAuto() {
        ps=0;
}

CAuto::CAuto(int ps) {
        this->ps = ps;
}
CAuto::~CAuto(){
}

CAuto::CAuto(const CAuto &a) {
        this->ps = a.ps;
}
```

Klasse CAuto: Auto.cpp

```
const CAuto& CAuto::operator = (const CAuto& a) {
 // Zuweisung an sich selbst?
        if (this == &a) return *this;
        this->ps = a.ps;
        return *this;
}
bool CAuto::operator() (const CAuto &a1,const CAuto &a2) const {
         if (a1.ps  ==  a2.ps)
                  return true;
         else
                  return false;
}
void CAuto::print() {
        cout << ps << endl;
}
```

Klasse CAuto: Auto.cpp

```
bool CAuto::operator< (const CAuto &a1) {
        if (ps  <  a1.ps)
                return true;
        else
                return false;
}
```

Klasse CAuto: Vergleichsmethoden

- bool CAuto::operator== (const CAuto &a1,const CAuto &a2) ;
- bool CAuto::operator!= (const CAuto &a1,const CAuto &a2) ;

- bool CAuto::operator< (const CAuto &a1,const CAuto &a2) ;
- bool CAuto::operator> (const CAuto &a1,const CAuto &a2) ;

- bool CAuto::operator<= (const CAuto &a1,const CAuto &a2) ;
- bool CAuto::operator>= (const CAuto &a1,const CAuto &a2) ;

- bool operator() (const CAuto &a1,const CAuto &a2) const;

Klasse CAuto: Test der Klasse CAuto

- Allgemein

```
void test1() {
        CAuto a1;
        CAuto a2(233);
        a1.print();
        cout << "a1: " << a1 << endl;
        cout << "a2: " << a2 << endl << endl;
        CAuto a3 = a2;
        a2.ps = 100;
        cout << "a2: " << a2 << endl;
        cout << "a3: " << a3 << endl << endl;
} // test1
```

Klasse CAuto: Einbau eines Vectors

```
typedef vector < CAuto > AutoVector;

void test2() {
        char line[80];
        AutoVector vect;
        CAuto a1(1);
        CAuto a2(100);
        CAuto a3(200);
        CAuto a4(1001);
```

Klasse CAuto: Einbau eines Vectors

```
vect.push_back(a1); // ans Ende einfügen
vect.push_back(a2); // ans Ende einfügen
vect.push_back(a3); // ans Ende einfügen
// Am Anfang einfügen
vect.insert( vect.begin(), a4 );

// löschen eines Autos, zuwenig PS  [ )
vect.erase( vect.begin()+1, vect.begin()+2 );

// löschen aller Autos, [ )
vect.erase( vect.begin()+1, vect.end() );
```

Klasse CAuto: Einbau eines Vectors

```
    // Ausgabe per Feld
    for (int i=0; i< vect.size(); i++ ) {
            vect[i].print();
    }  // for

    // Ausgabe per Iterator
    vector <CAuto>::iterator it;
    for (it= vect.begin(); it!=  vect.end(); ++it) {
            it->print();
    }
} // test2
```

Klasse CAuto: Einbau eines Stacks

```
typedef stack < CAuto > AutoStack;

void test3() {
        char line[80];
        AutoStack stack;
        CAuto a1(1);
        CAuto a2(100);
        CAuto a3(200);
        CAuto a4(1001);
```

Klasse CAuto: Einbau eines Stacks

```
void test3() {
        AutoStack stack;
        CAuto a1(1);
        CAuto a2(100);
        CAuto a3(200);
        CAuto a4(1001);
        stack.push(a1);
        stack.push(a2);
        stack.push(a3);
        stack.push(a4);
```

Klasse CAuto: Einbau eines Stacks

```
        // Ausgabe , immer den obersten wegnehmen
        while ( ! stack.empty() ) {
                stack.top().print();
                stack.pop();
        }
} // test3
```

Klasse CAuto: Anzahl, Suchen

```
void test4() {
        char line[80];
        int num;
        AutoVector vect;
        CAuto a1(1,1001);
        CAuto a2(2,100);
        CAuto a3(3,1001);
        CAuto a4(4,200);
        CAuto a5(5,500);
        vect.push_back(a1); // ans Ende einfügen
        vect.push_back(a2); // ans Ende einfügen
        vect.push_back(a3); // ans Ende einfügen
        // Am Anfang einfügen
        vect.insert(vect.begin(), a4);
        vect.push_back(a5); // ans Ende einfügen
```

Klasse CAuto: Anzahl, Suchen

```
vector <CAuto>::iterator it; // Ausgabe per Iterator
for (it=vect.begin(); it!= vect.end(); ++it) {
    it->print();
}
```

```
bool isMonster( CAuto& a) {
        return a.ps>=500;
}
```

```
puts("\n\nPause fuer count 'PS=1001' ");
gets(line);
CAuto asearch1(1,1001);  // Suchobjekt, == Vergleich
num  = count(vect.begin(), vect.end(), asearch1 );
cout << "Es gibt " << num << ". Autos mit der 1001 PS"  << endl;

puts("\n\nPause fuer count_if, Suche Auto mit PS>=500");
gets(line);
num = count_if(vect.begin(), vect.end(), isMonster ); // mit Funktion
cout << "Es gibt " << num << ". Autos mit 500 oder mehr PS" << endl;
```

Klasse CAuto: Anzahl, Suchen

```
    // suche nach einem Auto
    puts("\n\nPause fuer find des 1. Eintrags 'Auto mit 1001 PS");
    gets(line);
    CAuto asearch2(1,1001);   // Suchobjekt

    //it = find(vect.begin(), vect.end(), asearch2 );
    it = find(vect.begin(), vect.end(), 1001 );  // int-Konstruktor
    for (; it!= vect.end(); ++it) {
            it->print();
    }

} // test4
```

STL-Algorithmen: Definitionen

- **for_each(Inputiterator first, Inputiterator last, Funktion f);**
- Aufruf der Funktion F mit jedem Element in den Grenzen

- **find(Inputiterator first, Inputiterator last, const T&value);**
- Suchen des ersten Elements mit value in den Grenzen

- **find_if(Inputiterator first, Inputiterator last, Predicate pred);**
- Suchen des ersten Elements mit einem Funktionsaufruf pred

- **find_end(Inputiterator first, Inputiterator last, const T&value);**
- Suchen des letzten Elements mit value in den Grenzen

STL-Algorithmen: Definitionen

- **count(Inputiterator first, Inputiterator last, const T&value);**
- Bestimmen der Anzahl im Container nach dem Wert value in den Grenzen

- **count_if(Inputiterator first, Inputiterator last, Predicate pred);**
- Bestimmen der Anzahl im Container in den Grenzen mit einem Funktionsaufruf pred

```
bool isItemOkay( CClass& c) {
    return Bedingung;
}
```

STL-Algorithmen: Definitionen

- **mismatch**
- Suchen zweier Paare, die sich unterscheiden
- **equal**
- Zwei Bereiche werden verglichen
- **search**
- Suchen von Teilbereichen in einem großen Bereich
- **search_n**
- Suchen in einem großen Bereich nach n- gleichen Werten hintereinander

STL-Algorithmen: Definitionen

- **copy**
- Kopieren von Bereichen
- **copy_backward**
- Kopieren von Bereichen, Überlappende Bereiche
- **swap**
- Vertauschen von Elementen
- **iter_swap**
- Vertauschen der beiden Elementen von zwei Iteratoren
- swap_ranges
- transform
- replace
- replace_if

STL-Algorithmen: Definitionen

- replace_copy
- replace_if
- fill
- fill_n
- generate
- generate_n
- remove_copy
- remove_copy_if
- remove
- remove_if
- unique_copy
- unique

- reverse_copy
- rotate
- rotate_copy
- random_shuffle
- partition
- stable_partition
- sort
- stable_sort
- partial_sort
- partial_sort_copy
- nth_element

Such- Mischalgorithmen: Definitionen

- lower_bound
- upper_bound
- equal_range
- binary_search
- merge
- inplace_merge
- includes
- set_union
- set_intersection
- set_difference
- set_symmetric_difference
- make_heap
- pop_heap
- push_heap
- sort_heap
- min
- max
- min_element
- max_element
- lexiographical_compare
- next_permutation
- prev_permutation

Numerische Algorithmen: Definitionen

- **accumulate**
- **inner_product**
- **adjacent_difference**
- **partial_sum**

```
#include <numeric>  // accumulate

void test_accumulate() {
        CAuto summe;
        CAuto initAuto;
        summe = accumulate(vect.begin(), vect.end(),initAuto );
        printf("Summe: %d PS\n",summe.getPS() );
}
const CAuto& operator+(const CAuto &a) {
        ps = ps + a.PS;
        return *this;
}
```

- **initAuto erhält nacheinander alle Autos des Vectors**

Literatur

STL - kurz & gut

Ray Lischner
Deutsche Übersetzung von Torsten Wilhelm
1. Auflage April 2004
ISBN 3-89721-266-8
Seiten 134
EUR9.90, SFR16.90
Englischsprachige Ausgabe:
STL Pocket Reference

Literatur

Die C++ Standardbibliothek

Stefan Kuhlins
Martin Schader

ISBN 3-540-65052-0
380 Seiten

Literatur

Die C++ Standard Library

A Tutorial and Reference

Nicolai M. Josuttis

ISBN 0201379260
800 Seiten