Funktionsobjekte in STL-Algorithmen (`<algorithm>`)

In der Standardbibliothek können Prädikate oft über einen zusätzlichen Parameter als Funktionsobjekte übergeben werden.

Bsp.: Implementierung von max_element

```
template <class ForwardIterator>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last) {
  if (first == last) return first;
  ForwardIterator result = first;
  while (++first != last)
    if (*result < *first)
      result = first;
  return result;
}

template <class ForwardIterator, class Compare>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last,
                            Compare comp) {
  if (first == last) return first;
  ForwardIterator result = first;
  while (++first != last)
    if (comp(*result, *first)) result = first;
  return result;
}
```

Bsp.: Verwendung von max_element

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

using namespace std;

int main()
{
  double a[] = {2,4,5,7,3,4,6};
  const int N = sizeof(a)/sizeof(double);
  vector<double> b(a,a+N);

  cout << "max(a)=" << *max_element(a,a+N) << endl;
  cout << "max(b)=" << *max_element(b.begin(),b.end(),less<double>()) << endl;

  return 0;
}
```

Ausgabe:

```
max(a)=7
max(b)=7
```

Das im nächsten Beispiel eingesetzte copy-Template aus der Standardbibliothek könnte folgendermaßen realisiert sein:

```
template <class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first1, InputIterator last1,
                    OutputIterator first2)
{
  while (first1 != last1) {
    *first2 = *first1; ++first1; ++first2;
  }
  return first2;
}
```

Der Zielbereich muss hinreichend groß sein oder es muss ein einfügender Output-Iterator verwendet werden.

Bsp.:

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

using namespace std;

int main()
{
  double a[]={2,4,5,7,3,4,6};
  const int N = sizeof(a)/sizeof(double);
  double b[N];

  copy(a,a+N,b); sort(b,b+N);
  cout << "a: "; copy(a,a+N,ostream_iterator<double>(cout," ")); cout << endl;
  cout << "b: "; copy(b,b+N,ostream_iterator<double>(cout," ")); cout << endl;

  vector<double> c(a,a+N);
  sort(c.begin(),c.end(),greater<double>());
  cout << "c: "; copy(c.begin(),c.end(),ostream_iterator<double>(cout," "));
  cout << endl;

  list<double> d(N);
  copy(c.begin(),c.end(),d.begin());
  d.remove_if(bind2nd(less_equal<double>(),4));

  cout << "d: "; copy(d.begin(),d.end(),ostream_iterator<double>(cout," "));
  cout << endl;
}
```

Ausgabe:

```
a: 2 4 5 7 3 4 6
b: 2 3 4 4 5 6 7
c: 7 6 5 4 4 3 2
d: 7 6 5
```

Algorithmen in der STL (`<algorithm>`)

Die Algorithmen der STL arbeiten auf Folgen, die durch Iteratorbereiche beschrieben werden und greifen nur über diese auf die zugehörigen Behälter zu. Die Anwendbarkeit und ggf. Effizienz bemisst sich nach den Fähigkeiten der Iteratoren, die sich an der Iteratorkategorie ablesen lassen.

Iteratorkategorien

Bezeichnung	Richtung	Zugriff	Behälter/Strom
Input Iterator	vorwärts	lesen	Eingabestrom
Output Iterator	vorwärts	schreiben	Ausgabestrom
Forward Iterator	vorwärts	lesen/schreiben	
Bidirectional Iterator	vorwärts/rückwärts	lesen/schreiben	`list` `map`
Random Access Iterator	wahlfrei	lesen/schreiben	`vector`

Allerdings ist bei Konstanteniteratoren kein schreibender Zugriff möglich.

Einfügeiteratoren

Um Einfügungen im Zielbereich von STL-Algorithmen zu ermöglichen, gibt es Einfügeiteratoren, zu denen auch die einfügende Ausgabeiteratoren gehören. Diese definieren die Zuweisung so um, dass eine Komponente mit dem gleichen Wert erzeugt wird bzw. eine Ausgabe erfolgt.

Einfügeiterator	Benutzung von	Behälter/Strom	Wirkung
`back_inserter(a)`	`push_back`	`list vector`	in a hinten anhängen
`front_inserter(a)`	`push_front`	`list`	in a vorne einfügen
`inserter(a,pos)`	`insert`	`list map vector`	in a vor Position pos einfügen
`ostream_iterator<T>(stream,str)`	`<<`	Ausgabestrom	auf *stream* mit Trenner *str* ausgeben

Einfügeiteratoren können nützlich sein, wenn die Zahl der einzufügenden Komponenten nicht von vorneherein bekannt ist.

STL-Funktionen unique und unique_copy *(Beispiel)*

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

using namespace std;

template<class Container> void println(const Container& container, string str="",
                                       string delim =" ",ostream& stream = cout)
// Hilfsfunktionstemplate zur Ausgabe eines Behaelters
{
  if (str!="") stream << str << ": ";
  for (typename Container::const_iterator cpos=container.begin();
                cpos!=container.end();cpos++)
    stream << *cpos << delim;
  stream << endl;
}
```

```
int main()
{
  array<double,11> a={2,4,5,5,5,6,3,3,4,4,6};
  println(a,"a");

  vector<double> b;
  unique_copy(a.begin(),a.end(),back_inserter(b));
  println(b,"b");

  // Direkte Aenderung ebenfalls moeglich,
  // c ist allerdings NICHT verkuerzt, unique liefert Endposition
  vector<double> c(a.begin(),a.end());
  vector<double>::iterator pos = unique(c.begin(),c.end());
  println(c,"c");
  // Entfernen der ueberschuessigen Komp. mit Komp.fkt. erase
  c.erase(pos,c.end());
  println(c,"c");

  return 0;
}
```

Das im Beispiel gezeigte Verhalten der Nichtverkürzung trifft auch auf andere löschende STL-Algorithmen wie `remove, remove_if` zu.

Häufig verwendete STL-Algorithmen

`max(s,t) min(s,t)`	Maximum und Minimum typgleicher Werte
`swap(s,t)`	Vertauscht s und t
`count(anf,end,t)`	Liefert Anzahl der Komp. mit Wert t
`reverse(anf,end)`	Kehrt Reihenfolge um
`sort(anf,end) sort(anf,end,cmp)`	Sortiert $[anf, end)$ bzgl. $<$ bzw. *cmp*
`sort_stable(anf,end)`	Stabile Sortierung (Reihenfolge äquiv. Komp. unverändert)
`sort_stable(anf,end,cmp)`	bzgl. $<$ bzw. *cmp*

STL-Algorithmen für sortierte Bereiche (Auszug)

Die folgenden STL-Algorithmen setzen voraus, dass die Bereiche sortiert sind. Dadurch ergibt sich i. allg. ein besseres Zeitverhalten (z.B. $O(\ln n)$ anstelle $O(n)$ bei Komp.zahl n für Suche). Generell sind die behälterspezifischen Komponentenfunktionen den STL-Algorithmen vorzuziehen.

`lower_bound(anf,end,t)`	liefert erste Einfügeposition für t ohne Verletzung
`lower_bound(anf,end,t,cmp)`	der Ordnung bzgl. $<$ bzw. *cmp*
`upper_bound(anf,end,t)`	liefert letzte Einfügeposition für t ohne Verletzung
`upper_bound(anf,end,t,cmp)`	der Ordnung bzgl. $<$ bzw. *cmp*
`binary_search(anf,end,t)`	wahr, falls t in $[anf, end)$, sonst falsch
`binary_search(anf,end,t,cmp)`	
`includes(anf1,end1,anf2,end2)`	wahr, falls $[anf2, end2)$ Teilfolge von $[anf1, end1)$,
`includes(anf1,end1,anf2,end2,cmp)`	sonst falsch
`merge(anf1,end1,anf2,end2,anf3)`	erzeugt ab *anf3* einen zusammensortierten Bereich
`merge(anf1,end1,anf2,end2,anf3,cmp)`	aus $[anf1, end1)$ und $[anf2, end2)$; liefert *end3*
`set_union(anf1,end1,anf2,end2,anf3)`	erzeugt ab *anf3* einen sortierten vereinigten Bereich
`set_union(anf1,end1,anf2,end2,anf3,cmp)`	aus $[anf1, end1)$ und $[anf2, end2)$; liefert *end3*.

Entsprechend sind `set_intersection`, `set_difference`, `set_symmetric_difference` definiert.