

## 1. Die Standard Template Library (STL)

Die STL ist eine standardisierte Sammlung von sogenannten Containerklassen und darauf verallgemeinerten Algorithmen. Unter einem Container sind dynamische Klassenkonstrukte zu verstehen, die jeweils eine Menge von Objekten zu einem größeren Objekt zusammenfassen. Arrays (Felder) sind demnach die einzigen Container, die in der Sprache C/C++ bereits vordefiniert sind.

### 1.1 Containerklassen im Überblick

Wie bereits im Kapitel über die Felder ausgeführt, unterstützt der Standard-Sprachumfang von C/C++ keine dynamischen Arrays, also Felder, die ihre Größe an den aktuellen Bedarf an Einträgen selbst anpassen<sup>1</sup>. Lediglich der Umweg über eine eigene, dynamische Speicherverwaltung mit **new** und **delete** (bzw. **malloc** und **free** unter ANSI-C) ermöglicht es, das Verhalten von dynamischen Arrays nachzuahmen. Möchte man eine eigene, dynamische Felder realisieren, so ist der dafür zu betreibende Aufwand nicht zu unterschätzen:

- Die korrekte Anforderung und Freigabe des für das Feld verwendeten Speicherplatzes muß gewährleistet sein (Constructoren/Destructor)
- Bei jedem Einfügen bzw. Entfernen eines Feldelementes muß zumeist das gesamte Array im Speicher kopiert werden
- Alle im Zusammenhang mit Arrays verwendeten Operatoren müssen sauber überladen werden (Pointerarithmetik, [])

Einen ersten Eindruck des Aufwandes, welcher im einzelnen zu betreiben ist kann anhand des Programmbeispiels zur dynamischen Stringklasse gewonnen werden (siehe Kapitel über dynamische Speicherverwaltung). Unglücklicherweise sind diese Funktionalitäten für jedes Feld über einen bestimmten Datentyp neu zu schreiben, es sei denn man verwendet dafür den Mechanismus der Templates.

*Die STL ist vom ANSI-Komitee anerkannt und in ihrem (minimalen) Funktionsumfang festgelegt. Es gibt mehrere Implementationen<sup>2</sup>, die sich zwar in den Details der Programmierung (und STL-Erweiterungen) unterscheiden können, aber alle einen gleichen, verlässlichen Funktionalitäten bieten. Die Realisierung in Form von Templates hat zur Folge, daß die Implementierung der STL nur aus Headerdateien besteht und daher unabhängig vom verwendeten Compiler und Funktionsbibliotheken ist (sofern der Compiler Templates voll unterstützt).*



<sup>1</sup> Zur Ehrenrettung der Sprachen C und C++ sei hier angemerkt, daß nur sehr wenige Programmiersprachen dynamische Arrays bieten.

<sup>2</sup> Die bekanntesten und am meisten genutzten Implementationen sind die Realisierungen von Hewlett Packard (HP) und Roque Wave.

Da es unsinnig ist, das Rad jeden Tag neu zu erfinden, stellt die STL eine ganze Reihe gängiger Containerklassen in optimierter und verallgemeinerter Form zur Verfügung, wobei aus Gründen der Effizienz auf Vererbung verzichtet wird. Eine ganze Reihe von Experten haben mehrere Jahre darauf verwendet mit der STL eine Lösung zu schaffen, die den schwierigen Spagat zwischen Flexibilität, Geschwindigkeit, Verallgemeinerung und minimalem Aufwand bewältigen kann.



*Verwenden Sie nur den vom ANSI-Komitee festgelegtem Umfang der STL wenn Ihre Programme zwischen mehreren Compilern und/oder Betriebssystem-Plattformen portabel bleiben sollen. Die STL-Implementationen sind zwar grundsätzlich portabel, aber zumeist (sofern mit dem Compiler ausgeliefert, was bei allen aktuellen Compilern der Fall sein sollte) mit den Standard-Headern vermischt, so daß eine Aktualisierung oder ein Austausch der STL-Implementation sehr schwierig und aufwendig ist.*

In einer STL-Implementation enthalten sind (zumindest) die Templates der folgenden Containerklassen:

<b>Sequentielle Containerklassen der STL</b>	
<i>Containername</i>	<i>Bedeutung</i>
vector	dynamisches Array. Alle enthaltenen Datenelemente liegen nebeneinander (benachbart) im Speicher (daher ist ein Zugriff über berechnete Indizes möglich). Das Einfügen neuer Datenelemente ist prinzipiell an beliebiger Stelle, ein schnelles (performantes) Einfügen ist aber nur am Ende des Arrays möglich.
deque	Lineare, einseitig verkettete Liste. Die enthaltenen Datenelemente liegen <b>nicht</b> benachbart im Speicher. Ein performantes Einfügen ist nur am Anfang und am Ende der Liste möglich.
list	Lineare, beidseitig verkettete Liste. Die enthaltenen Datenelemente liegen <b>nicht</b> benachbart im Speicher. Ein performantes Einfügen ist an beliebiger Stelle der Liste möglich.

Tabelle 1-1– Sequentielle Containerklassen der STL

<b>Assoziative Containerklassen der STL</b>	
<i>Containername</i>	<i>Bedeutung</i>
set	Datenmenge (Datensammlung) mit schnellem, assoziativen Zugriff. Duplikate sind <b>nicht</b> zulässig
multiset	Datenmenge (Datensammlung) mit schnellem, assoziativen Zugriff.

<b>Assoziative Containerklassen der STL</b>	
<i>Containername</i>	<i>Bedeutung</i>
	Duplikate sind zulässig
map	Sammlung von Daten in einer 1:1 Relation.
multimap	Sammlung von Daten in einer 1:n Relation.

Tabelle 1-2– Assoziative Containerklassen der STL

<b>Adaptive Containerklassen der STL</b>	
<i>Containername</i>	<i>Bedeutung</i>
stack	Kellerspeicher. Datenstruktur mit strikter FILO <sup>3</sup> -Implementation.
queue	Warteschlange. Datenstruktur mit strikter FIFO <sup>4</sup> -Implementation.
priority_queue	Warteschlange mit Prioritätssortierung.

Tabelle 1-3– Adaptive Containerklassen der STL

Die „Arbeitspferde“ unter den Containerklassen sind üblicherweise die **vector**- und **list**-Templates, daher werden diese beiden Container im weiteren Verlauf dieser Darstellung am intensivsten betrachtet werden.

*Da Containerklassen Abstraktionen über beliebigen anderen Klassen sind, wird in den Beschreibungen der Container statt eines zu verwaltenden Klassennamens zumeist der symbolische Platzhalter  $T$  (für Type) verwendet.*

*Der Platzhalter  $T$  ist im Programm durch den gewünschten Klassennamen zu ersetzen (z.B. `int` oder wie in Beispiel `STL_VECTOR_01` durch `HQG_Person`).*

*Size hingegen ist ein logischer Größentyp für eine Zahl von Elementen vom Typ  $T$ . Es ist davon auszugehen, daß Size üblicherweise mit `int` oder `long` identisch ist.*



Grundsätzlich kann jeder beliebige Datentyp mit einer Container-Klasse verwaltet werden, dies gilt für Standarddatentypen ebenso wie für selbstdefinierte Klassen und Strukturen.

Allerdings müssen selbstdefinierte Datentypen einige Voraussetzungen mitbringen, um korrekt zu funktionieren.

Die Verwendung bestimmter Algorithmen aus den Template-Headerdateien bedingen weitere Funktionalitäten, die der Entwickler sozusagen als Vorleistung zu erbringen hat.

Die Verwendung von Containerklassen bedingt auf jeden Fall, daß sowohl der Standard-Constructor, der Copy-Constructor und der Destructor sauber definiert sind.

<sup>3</sup> FILO = First-In-Last-Out. Das Datenelement das zuerst in den Container hineingelegt wird kann erst als Letztes wieder herausgenommen werden. D.h. die Entnahme erfolgt in umgekehrter Reihenfolge der Füllung.

<sup>4</sup> FIFO = First-In-First-Out. Das Datenelement das zuerst in den Container hineingelegt wird muß als Erstes wieder entnommen werden. D.h. die Entnahme erfolgt in der Reihenfolge der Füllung.



*Der Compiler erstellt immer Standard-Constructor, Copy-Constructor, Destructor und die Zuweisung zwischen zwei Elementen des vorliegenden Datentyps – es sei denn, diese Funktionalitäten werden vom Entwickler bereitgestellt oder durch nicht aufrufbare, leere Methoden verboten.*

*Der vom Compiler erzeugte Standard-Constructor tut nichts, außer den notwendigen Speicherplatz zu reservieren. Eine Initialisierung des Speicherplatzes finden nicht statt.*

*Der vom Compiler erzeugte Copy-Constructor und die Zuweisung kopieren den gesamten Inhalt einer Klasse in das neu zu erzeugende bzw. zugeordnete Objekt. Dies gilt auch für darin enthaltene Pointer, die schlicht kopiert werden. Tiefe Kopien, d.h. Kopien der Objekte auf die ein Pointer in der Klasse zeigt werden nicht erstellt. Dies ist eine erhebliche Fehlerquelle, da nach durchlaufen des Copy-Constructors zwei Pointer auf das gleiche Objekt zeigen und dieses ggf. zweimal freigegeben wird.*

*Der vom Compiler erzeugte Standard-Destructor tut nichts außer den reservierten Speicherplatz wieder freizugeben. Abhängige Objekte, auf die über Pointer referenziert wird werden nicht wieder freigegeben.*

Demnach ist das folgende Programm völlig legal, auch wenn das Verhalten des Programms nicht vorhersehbar ist:



```
//=====
// PROGRAMM: STL_BSP_01
//=====

//-----
// eine Klasse ohne alles
//-----
class a
{
    int b;
};

//-----
// Hauptprogramm
//-----
void main (void)
{
    a x;          // generierten Standard-Constructor aufrufen
    a z (x);     // generierten Copy-Constructor aufrufen
    x = z;       // generierte Zuweisung aufrufen
}               // generierter Destructor wird aufgerufen
```

Die Konsequenzen und Fehlerquellen, die sich aus diesem Umstand ergeben können, sollten jedem erfahrenen Entwickler sofort auffallen und zu denken geben<sup>5</sup>.

*Wenn die vom Compiler erzeugten Funktionalitäten nicht zulassen werden sollen, dann müssen diese ausdrücklich als „private“ deklariert und definiert werden:*



```
class Demo
{
public:
    Demo (int x)    {} // Es muß mind. einen Constr. geben
    ~Demo (void)   {} // Destruct. darf nicht verboten werden

private:
    Demo (void)    {} // Standard-Constructor verbieten
    Demo (Demo& x) {} // Copy-Constructor verbieten
    void operator= (Demo& x) {} // Zuweisung
};
```

*Wenn die beiden vom Compiler erzeugen Constructoren verboten werden, muß mindestens ein weiterer Constructor existieren, sonst ließe sich kein Objekt dieser Klasse instanziiieren.*

*Auch darf der einzig mögliche Destructor natürlich nicht verboten werden.*

```
//=====
// PROGRAMM: STL_BSP_02
//=====

//-----
// eine saubere Klasse
//-----
class a
{
public:
    a (int x) {b = x;} // Es muß mind. einen Constr. geben
    ~a (void) {} // Destruct. darf nicht verboten werden

protected:
    int b;

private:
    a (void) {} // Standard-Constructor verbieten
    a (a& x) {} // Copy-Constructor verbieten
    void operator= (a& x) {} // Zuweisung
};

//-----
// Hauptprogramm
//-----
```



<sup>5</sup> genauer gesagt sollte es jeden Entwickler auf de Palme bringen, die Haare zu Berge stehen und die Röte ins Gesicht treiben lassen – und die Frage ob alle seine Klassen auch sauber sind sollte ihm schlaflose Nächte bereiten...  
Die dynamische Stringklasse aus dem Kapitel über **new** und **delete** ist hinsichtlich der generierten Funktionen vollständig und kann daher als Vorlage dienen.

```

void main (void)
{
    a x;          // erzeugt Fehler "is not accessible"
    a z (x);      // erzeugt Fehler "is not accessible"
    x = z;        // erzeugt Fehler "is not accessible"
}                // generierter Destructor wird aufgerufen

```

Letztlich ist es nur logisch, daß alle verwendbaren Constructoren und der Destructor immer korrekt arbeiten müssen. Leider werden die automatisch generierten Constructoren sehr häufig übersehen, zumal der vom Compiler erzeugte Copy-Constructor – bei Klassen die keine dynamischen Daten verwalten – vollauf genügt.



*Um alle STL-Algorithmen zu unterstützen, sollte eine Klasse die folgenden Constructoren, Methoden und Operatoren bieten:*

*Copy-Constructor*  
*Standard-Constructor*  
*Destructor*  
*Zuweisung, d.h. operator=*  
*Vergleich, d.h. operator==*  
*Kleiner-als, d.h. operator<*

## 1.2 Iteratoren

Die Idee die hinter den Iteratoren (Aufzähler) steckt ist ebenso einfach nachzuvollziehen wie schwierig zu implementieren: eine allgemeingültige Zugriffsform auf Elemente in STL-Containern zu schaffen, die für den Anwender unabhängig vom verwendeten Containertyp ist. Dies hat den Vorteil, daß z.B. der Typ der Containerklasse nachträglich geändert werden kann, ohne daß alle Verarbeitungsfunktionen umgeschrieben werden müssen.

Auch können allgemeine Algorithmen geschrieben werden (z.B. Sortierungen), die sich unabhängig von der zugrundeliegenden Containerklasse implementiert lassen.

Um sich die Vorteile zu verdeutlichen, kann man sich die Umstellung eines **vector**- auf einen **list**-Container vorstellen. Während bei einem **vector**-Container alle Elemente hintereinander liegen (schließlich ist der **vector** ja ein Array) liegen die einzelnen Elemente einer **list** verstreut im Speicher, miteinander verkettet durch Zeiger.

Daher kann man eine Schleife über die **vector**-Elemente einfach durch einen Indexzugriff implementieren, den Zugriff auf **list**-Elemente nicht.



### Zur Erinnerung:

*Das Ansprechen eines Array-Elementes über den Index ist eine Adressenoperation, bei der auf die Basisadresse (= Name des Arrays) ein Offsetwert aufaddiert wird (Indexwert \* Größe eines Array-Elementes).*

Eine solche Indexoperation läßt sich natürlich nicht auf eine verkettete Liste übertragen, da die Elemente eines **list**-Containers nur über ein „hindurchhangeln“ entlang der Verpointerung der Elemente untereinander erreichbar sind.

So enthält z.B. das erste Element einer **list** einen Zeiger auf das zweite Element. Das zweite Element wiederum enthält je einen Zeiger auf das erste und dritte Element usw.

Entscheidender Unterschied zum **vector** ist jedoch, daß dadurch nur eine logische Reihenfolge entsteht, die physikalische Reihenfolge im Speicher des Rechners ist nicht festgelegt. Bei einem **vector** hingegen muß die logische der physikalischen Reihenfolge entsprechen, da sonst die oben dargestellte Adreßoperation fehlschlagen würde.

Die Iteratoren abstrahieren von der physikalischen Reihenfolge und kapseln den Zugriff auf die logische Reihenfolge dergestalt, daß sich hinsichtlich der Iteratoren alle Containerklassen gleichartig benutzen lassen. D.h. der Entwickler muß sich nicht darum kümmern, ob das logisch nächste Element über einen Index oder eine Verzeigerung ermittelt werden muß, dies macht der entsprechende **iterator** (logischerweise bedeutet dies, daß die Iteratoren der verschiedenen Containerklassen unterschiedlich implementiert sind).

Für den Entwickler verhalten sich die Iteratoren alle gleich, da sie die implementierte Funktionalität immer auf die gleiche Art und Weise zur Verfügung stellen.

```
vector<int> aVector;           // dyn. Array vom Typ Integer
vector<int>::iterator aIter; // Iterator (Aufzähler)

// ...

for (aIter=aVector.begin(); aIter<aVector.end(); aIter++)
{
    // ...
}
```

```
list<int> aList;             // verk. Liste vom Typ Integer
list<int>::iterator aIter;  // Iterator (Aufzähler)

// ...

for (aIter=aList.begin(); aIter<aList.end(); aIter++)
{
    // ...
}
```

Die Methode **begin()** liefert eine Iterator-Referenz auf das erste in der Containerklasse enthaltene Element zurück. Die Methode wird in den folgenden Kapiteln bei den jeweiligen Containerklassen ausführlicher erläutert.

### 1.2.1 Iterator-Zugriffsformen

Wie die einzelnen Containerklassen, so haben auch die darauf definierten Iteratoren bestimmte Eigenschaften. Dies scheint zunächst ein Widerspruch gegenüber der oben getätigten Aussage zu sein, daß alle Iteratoren sich gleichartig benutzen lassen, ist es aber nicht.

Auf bestimmte Containerklassen läßt sich aufgrund ihrer Definition z.B. nicht wahlweise zugreifen. Dies ist z.B. beim **list**-Container der Fall, da nur eine Verkettung existiert, ist es nicht möglich direkt auf ein bestimmtes Element (z.B. das 7.) zuzugreifen (wie es beim **vector** erlaubt ist). Zwar läßt sich ein entsprechendes Verhalten implementieren, da aber die Position des 7. Elementes nicht berechenbar ist (s.o.), kann man eigentlich nur am ersten Element des Containers beginnen und in einer Schleife entsprechend oft auf den Nachfolger wechseln. Ein solches Verhalten entspricht aber nicht der Definition eines wahlfreien Zugriffs, da das Zeitverhalten nicht konstant ist (der Zugriff auf des 7. Element ist bei Lösung über eine Schleife natürlich schneller als der Zugriff auf das 70000. Element).

Der Zugriff auf einen **vector** hingegen ist wahlfrei, da über den Index direkt jedes Element in konstanter Zeit ermittelt werden kann (siehe Adreßoperation oben).

Fast alle Container (bis auf **stack** und **queue**) sind zudem so definiert, daß auf das erste und letzte Element schnell zugegriffen werden kann und man sich mit Hilfe der Iteratoren vorwärts und rückwärts durch die Sammlung bewegen kann (dies entspricht dem Wechsel auf den Vorgänger und Nachfolger in einer **list** bzw. einer Indexveränderung um Plus oder Minus Eins in einem **vector**), daher werden diese Iteratoren bidirektional genannt. Iteratoren mit wahlfreiem Zugriff sind logischerweise immer bidirektional, denn - wenn man auf jedes beliebige Element wechseln kann, dann auch immer auf den direkten Vorgänger und Nachfolger.

<b>Iterator-Zugriffsformen</b>	
<i>Containername</i>	<i>Iterator-Zugriffsform</i>
Vector	Wahlfreier Zugriff
Deque	Wahlfreier Zugriff
List	Bidirektionaler Zugriff
Set	Bidirektionaler Zugriff
Multiset	Bidirektionaler Zugriff
Map	Bidirektionaler Zugriff
Multimap	Bidirektionaler Zugriff
Stack	Keine Iteratoren
Queue	Keine Iteratoren

*Tabelle 1-1– Iterator-Zugriffsformen*

### 1.2.1.1 Verwendung von Iteratoren mit bidirektionalem Zugriff

Die Verwendung von bidirektionalen Iteratoren ist sehr einfach. Da nur auf das logisch davor- oder dahinterliegende Element zugegriffen werden kann, wird dies durch die aus C/C++ bekannten Operatoren für Inkrement und Dekrement realisiert:

```
vector<int> aVector;           // dyn. Array vom Typ Integer
vector<int>::iterator aIter;  // Iterator (Aufzähler)
vector<int>::iterator aIter2; // Iterator (Aufzähler)

aIter=aVector.begin();       // auf erstes Element setzen
aIter++;                     // Nachfolger, Postinkrement
++aIter;                     // Nachfolger, Präinkrement
Iter--;                       // Nachfolger, Postdekrement
--aIter;                     // Nachfolger, Prädekrement
```

```

*aIter;                // Inhalt des Datensatzes auf
den
                        // aIter zeigt
*aIter = 7             // Inhalt verändern

aItr2 = aIter;         // Zuweisung

```

### 1.2.1.2 Verwendung von Iteratoren mit wahlfreiem Zugriff

Die Verwendung von Iteratoren mit wahlfreiem Zugriff ist eine Erweiterung des bidirektionalen Zugriffs. Zusätzlich zu den Operatoren für Inkrement und Dekrement können auch direkte Zuweisungen mathematische Operationen verwendet werden:

```

vector<int> aVector;    // dyn. Array vom Typ Integer
vector<int>::iterator aIter; // Iterator (Aufzähler)
vector<int>::iterator aItr2; // Iterator (Aufzähler)

aIter=aVector.begin(); // auf erstes Element setzen
aIter++;               // Nachfolger, Postinkrement
++aIter;               // Nachfolger, Präinkrement
aIter--;               // Nachfolger, Postdekrement
--aIter;               // Nachfolger, Prädekrement

*aIter;                // Inhalt des Datensatzes auf
den
                        // aIter zeigt
*aIter = 7             // Inhalt verändern

aItr2 = aIter;         // Zuweisung
aItr2 = aIter + 3;    // drei Elemente vor
aIter += 3;           // drei Elemente vor
aItr2 = aIter - 2;    // zwei Elemente zurück
aIter -= 2;           // zwei Elemente zurück
aItr2 = aIter[2];     // zweites Element hinter aIter

```

### 1.2.2 Iteratortypen

Neben den Zugriffsformen, die letztlich durch die dahinterliegende Containerklasse bestimmt werden, gibt es unterschiedliche Containertypen, welche die Art des Zugriffs und die Richtung der Aufzählung bestimmen. Die häufigste Form ist der einfache Iterator, wie bereits in den oben dargestellten Beispielen verwendet.

Praktischerweise gibt es für alle Containerklassen, die den bidirektionalen Zugriff unterstützen auch einen Iteratortyp, der eine rückwärtsgerichtete Aufzählung vornimmt den **reverse\_iterator**.

```

vector<int> aVector;    // dyn. Array vom Typ
Int
vector<int>::reverse_iterator aRIter; // Iterator (Aufzähler)

aRIter=aVector.rbegin(); // auf Aufzählungsbeginn
setzen

```

Die Methode **rbegin()** für einen **reverse\_iterator** entspricht in ihrer Bedeutung der Methode **begin()** für einen „einfachen“ **iterator**. Neben diesem einfachen Iterator gibt es auch **const**-Iteratoren, die eine

Klasse zwar aufzählen können (z.B. zur Ausgabe), über die aber kein verändernder Zugriff erlaubt ist:



```
vector<int> aVector;           // dyn. Array vom Typ Int
vector<int>::const_iterator aIter; // Iterator (Aufzähler)
vector<int>::const_reverse_iterator aRIter;

aIter=aVector.begin();      // auf Aufzählungsbeginn
setzen
aRIter=aVector.rbegin();    // auf Aufzählungsbeginn
setzen
*aIter = 7;                 // VERBOTEN
```

### 1.3 STL-Exceptions

Im Regelfall<sup>6</sup> unterstützen die Containerklassen den Exception-Mechanismus von C++, wobei die Exceptions der Speicherverwaltung hierbei eine tragende Rolle spielen, da jedes Anfügen eines Datenelementes eine Speicherallokation auslösen kann.

Aus der Speicherverwaltung ist mit den folgenden, abzufangenden Exception-Klassen zu rechnen:

<b>STL-Exceptions</b>	
<i>Klasse</i>	<i>Bedeutung</i>
bad_alloc	Fehler bei der Allokation neuen Speichers
logic_error	Allgemeiner Fehler aufgrund einer Zusicherungsverletzung
invalid_argument	Fehlerhafte Ausführung eines Container-Constructors (z.B. bei einer Größenangabe -1)
length_error	Fehlerhafte Ausführung eines String-Constructors (z.B. bei einer Größenangabe -1)
out_of_range	Zugriff auf einen ungültigen Container-Index oder über einen ungültigen Iterator

Tabelle 1-1– STL-Exceptions

Das folgende Beispielprogramm zeigt den Einsatz des Exception-Handlings:



```
//=====
// PROGRAMM: STL_BSP_03
//=====

#include <iostream.h>
#include <vector>
using namespace std;

void main (void)
{
    vector<int> aVect;
    vector<int>::const_iterator aIter;
    int i;

    aVect.push_back (7);
    aVect.push_back (4);
    aVect.push_back (2);
```

<sup>6</sup> Einige STL-Implementationen verzichten zugunsten eine benutzerdefinierten Fehlerbehandlung darauf Exceptions auszulösen.

```

aIter = aVect.begin();

//-----
-
// Überwacher Bereich für Exceptionhandling von C/C++
//-----
-
try
{
    for (i=0; i<10; i++)
    {
        cout << *aIter;
        aIter++;
    }
}
//-----
-
// Abfangen der unterschiedlichen Fehlertypen
//-----
-
catch (bad_alloc& aError)
{
    cout << "Fehler in Speicherverwaltung" << endl;
}
catch (logic_error& aError)
{
    cout << "Allgemeiner Fehler" << endl;
}
catch (invalid_argument& aError)
{
    cout << "Ungültiger Parameter" << endl;
}
catch (length_error& aError)
{
    cout << "Fehler in Stringlänge" << endl;
}
catch (out_of_range& aError)
{
    cout << "Ungültiges Element angesprochen" << endl;
}
catch (...) // alle anderen
{
    cout << "Unbekannter Fehler" << endl;
}
}

```

## 1.4 Container-Methoden im Überblick

Alle Containerklassen enthalten eine ganze Reihe von Methoden. Um den Umgang mit den Containern zu erleichtern sind gleichartige Methoden auch gleich benannt.

Die folgende Tabelle listet die einzelnen Methoden und ihre Verfügbarkeit in den unterschiedlichen Containerklassen auf.

Die Bedeutung der Methoden und ihre Parameter werden im jeweiligen Kapitel über die Containerklasse erklärt.

<b>Methodenübersicht</b>											
<b>Methoden</b>	<b>Vector</b>	<b>Deque</b>	<b>List</b>	<b>Set</b>	<b>Multiset</b>	<b>Map</b>	<b>Multimap</b>	<b>Stack</b>	<b>Queue</b>	<b>Priority_Queue</b>	<b>String</b>
append											X
assign											X
back	X	X	X						X		
begin	X	X	X	X	X	X	X				
capacity	X										
compare											X
copy											X
count				X	X	X	X				
c_str											X
data											X
empty	X	X	X	X	X	X	X	X	X	X	X
end	X	X	X	X	X	X	X				
equal_range					X	X	X				
erase	X	X	X	X	X	X	X				
exchange											X
find				X	X	X	X				X
find_first_not_of											X
find_first_of											X
find_last_not_of											X
find_last_of											X
front	X	X	X						X		
insert	X	X	X	X	X	X	X				X
key_comp				X	X	X	X				
length											X
lower_bound				X	X	X	X				
max_size	X	X	X	X	X	X	X	X	X	X	
merge			X								
pop								X	X	X	
pop_back	X	X	X								
pop_front		X	X								
push								X	X	X	
push_back	X	X	X								
push_front		X	X								
rbegin	X	X	X	X	X	X	X				
remove			X								X
rend	X	X	X	X	X	X	X				
replace											X
reserve	X										X
resize											X
reverse			X								
rfind											X
size	X	X	X	X	X	X	X	X	X	X	X
sort <sup>7</sup>			X								
splice			X								
substr											X
swap	X	X	X	X	X	X	X	X	X	X	
top								X		X	

<sup>7</sup> Nicht zu verwechseln mit dem generischen STL-Algorithmus **sort**. Die **sort**-Methode ist optimal auf die innere Struktur der Containerklasse abgestellt.

<b>Methodenübersicht</b>											
<b>Methoden</b>	<b>Vector</b>	<b>Deque</b>	<b>List</b>	<b>Set</b>	<b>Multiset</b>	<b>Map</b>	<b>Multimap</b>	<b>Stack</b>	<b>Queue</b>	<b>Priority_Queue</b>	<b>String</b>
unique			X								
upper_bound				X	X	X	X				
value_comp				X	X	X	X				

<b>Operatorenübersicht</b>											
<b>Methoden</b>	<b>Vector</b>	<b>Deque</b>	<b>List</b>	<b>Set</b>	<b>Multiset</b>	<b>Map</b>	<b>Multimap</b>	<b>Stack</b>	<b>Queue</b>	<b>Priority_Queue</b>	<b>String</b>
operator =	X	X	X	X	X	X	X	X	X	X	X
operator ==	X	X	X	X	X	X	X	X	X		
operator !=	X	X	X	X	X	X	X	X	X		
operator <	X	X	X	X	X	X	X	X	X		
operator >	X	X	X	X	X	X	X	X	X		
operator <=	X	X	X	X	X	X	X	X	X		
operator >=	X	X	X	X	X	X	X	X	X		
operator []	X	X				X					X
operator +=											X
operator +											X

## 2. Die STL Hilfsklasse Pair

Einige Containerklassen, –methoden und Algorithmen verwenden Instanzen der Hilfsklasse **pair** um Parameter entgegenzunehmen, oder um mehr als ein Funktionsergebnis zurückliefern zu können (z.B. **multimap**).

Die Hilfsklasse **pair** ist eine Templateklasse, welche im STL Headerfile **utility** definiert ist. Durch die Implementierung als Template können beliebige Datenpaare als **pair** gehalten werden.

### 2.1 Die pair Constructoren

Die folgende Tabelle faßt die Constructoren der Hilfsklasse **pair** zusammen. In den nachstehenden Abschnitten werden die Constructoren einzeln behandelt und mit Beispielen erläutert.

<b>Constructoren der Hilfsklasse PAIR</b>	
<i>Constructor</i>	<i>Bedeutung</i>
pair<const T& t, const U& u> VarName;	Standard-Constructor, erzeugt ein neues, leeres <b>pair</b> -Objekt für zwei Datenelemente mit den Datentypen <b>T</b> und <b>U</b> ( <b>T</b> und <b>U</b> dürfen den gleichen Datentyp repräsentieren)

Tabelle 2-1– Constructoren der Hilfsklasse pair

#### 2.1.1 Der pair Standard-Constructor

Der **pair** Standard-Constructor, erzeugt ein neues, leeres **pair**-Objekt für die Datentypen **T** und **U**.

```
Syntax:
pair<const T& t, const U& u> Variablenname;
```

Das folgende Beispielprogramm zeigt den Einsatz des Standard-Constructors innerhalb eines Programms:

```
//=====
// PROGRAMM: PAIR_BSP_01
//=====

#include <utility>

using namespace std;

void main (void)
{
    pair<int, double>    aPair;
    pair<char *, char *> aPair2;
}
```



## 2.2 Die pair Member

Die folgende Tabelle faßt die Member (Datenelemente) der **pair** Hilfsklasse zusammen. Im Gegensatz zu den Container- und Adapterklassen beinhaltet ein **pair**-Objekt praktisch keinerlei Funktionalität, so daß eine Methoden-Schnittstelle überflüssig ist. Statt dessen wird auf die gespeicherten Daten

direkt zugegriffen:

<b>Member der Adapterklasse PAIR</b>	
<i>Member</i>	<i>Bedeutung</i>
first	Erster Wert des <b>pair</b> (Element des Typs <b>T</b> )
second	Zweiter Wert des <b>pair</b> (Element des Typs <b>U</b> )

Tabella 2-1– Member der Adapterklasse pair

### 2.2.1 Der pair Member first

Der Member **first** bezeichnet das erste im **pair** gespeicherte Datenelement.

Syntax:  
T first;



```
//=====
// PROGRAMM: PAIR_BSP_02
//=====

#include <iomanip.h>
#include <iostream.h>
#include <utility>

using namespace std;

void main (void)
{
    pair<int, int> aPair;

    aPair.first = 7;

    cout << "Inhalt: " << aPair.first << endl;
}

```

### 2.2.2 Der pair Member second

Der Member **second** bezeichnet das zweite im **pair** gespeicherte Datenelement.

Syntax:  
U second;



```
//=====
// PROGRAMM: PAIR_BSP_03
//=====

#include <iomanip.h>
#include <iostream.h>
#include <utility>

using namespace std;

void main (void)
{
    int z = 17;
    pair<int, int> aPair;

    aPair.first = z;
    aPair.second = 4;
}

```

```

cout << "Inhalt 1: " << aPair.first << endl;
cout << "Inhalt 2: " << aPair.second << endl;
}

```

## 2.3 Die pair Hilfsfunktionen

Die folgende Tabelle faßt Hilfsfunktionen zusammen, die die Klasse **pair** benutzen oder den Umgang mit **pair** erleichtern:

<b>Hilfsfunktionen der Klasse PAIR</b>	
<i>Funktion</i>	<i>Bedeutung</i>
make_pair	Erzeugt ein <b>pair</b>

Tabelle 2-1– Hilfsfunktionen der Klasse pair

### 2.3.1 Die Funktion make\_pair

Der Funktion **make\_pair** erzeugt ein neues **pair**-Objekt mit den als Parameter übergebenen Datenelementen..

Syntax:  

```
pair<T,U>& make_pair (T& t, U& u);
```

```

//=====
// PROGRAMM: PAIR_BSP_04
//=====

#include <iomanip.h>
#include <iostream.h>
#include <utility>

using namespace std;

void main (void)
{
    pair<int, int> aPair = make_pair(1,7);

    cout << "Inhalt 1: " << aPair.first << endl;
    cout << "Inhalt 2: " << aPair.second << endl;
}

```



## 2.4 Die pair Operatoren

Mit den folgenden Operatoren kann auf ein **pair** zugegriffen werden:

<b>Operatoren der Hilfsklasse PAIR</b>	
<i>Operator</i>	<i>Bedeutung</i>
==	Vergleich zwischen zwei <b>pair</b> -Objekten gleichen Typs
!=	Vergleich zwischen zwei <b>pair</b> -Objekten gleichen Typs
<	Lexikographischer Vergleich zwischen zwei <b>pair</b> -Objekten gleichen Typs
>	Lexikographischer Vergleich zwischen zwei <b>pair</b> -Objekten gleichen Typs
>=	Lexikographischer Vergleich zwischen zwei <b>pair</b> -Objekten gleichen Typs
<=	Lexikographischer Vergleich zwischen zwei <b>pair</b> -Objekten gleichen Typs

Tabelle 2-1– Operatoren der Hilfsklasse pair

Wenn es sich bei den Datentypen **T** und/oder **U** um selbstdefinierte Datentypen handelt, ist der Zugriff über die oben aufgeführten Operatoren nur dann möglich, wenn diese für **T** bzw. **U** vom Entwickler bereitgestellt wurden.

Da die STL selbst einen Teil der notwendigen Operatoren ableiten kann, ist es ausreichend die Operatoren **operator=** , **operator==** und **operator<** zu überladen. Die restlichen Operatoren können daraus erschlossen werden.

#### 2.4.1 Der pair Operator ==

Der Operator prüft, ob die Inhalte des linken und rechten Operanden (**pair**-Objekte gleichen Typs) übereinstimmen. Dazu werden die beiden Elemente **first** und **second** verglichen, indem deren Vergleichsoperator aufgerufen werden.

Das Ergebnis der Operation ist **true** wenn beide **pair**-Objekte gleiche Datenelemente oder in gleicher Reihenfolge enthalten.

Syntax:

```
bool operator== (const pair<T,U>& p1,
                const pair<T,U>& p2);
```



```
//=====
// PROGRAMM: PAIR_BSP_05
//=====

#include <iomanip.h>
#include <iostream.h>
#include <utility>

using namespace std;

void main (void)
{
    typedef pair<double, double> mypairtype;
    mypairtype aPair1;
    mypairtype aPair2;

    aPair1.first  = 1.1;
    aPair1.second = 1.1;
    aPair2.first  = 1.1;
    aPair2.second = 1.1;

    if (aPair1 == aPair2)
    {
        cout << "Die Pair-Objekte sind gleich" << endl;
    }

    aPair2.first = 1.2;

    if (aPair1 == aPair2)
    {
        cout << "Die Pair-Objekte sind gleich" << endl;
    }
}
```

#### 2.4.2 Der pair Operator !=

Der Operator prüft, ob die Inhalte des linken und rechten Operanden (**pair**-

Objekte gleichen Typs) voneinander abweichen. Dazu werden die beiden Elemente **first** und **second** verglichen, indem deren Vergleichsoperator aufgerufen werden.

Das Ergebnis der Operation ist **true** wenn beide **pair**-Objekte unterschiedliche Datenelemente oder gleiche Datenelemente in unterschiedlicher Reihenfolge enthalten.

Syntax:

```
bool operator!= (const pair<T,U>& p1,
                const pair<T,U>& p2);
```

```
//=====
// PROGRAMM: PAIR_BSP_06
//=====

#include <iomanip.h>
#include <iostream.h>
#include <utility>

using namespace std;

void main (void)
{
    typedef pair<double, double> mypairtype;
    mypairtype aPair1;
    mypairtype aPair2;

    aPair1.first  = 1.1;
    aPair1.second = 1.1;
    aPair2.first  = 1.2;
    aPair2.second = 1.1;

    if (aPair1 != aPair2)
    {
        cout << "Die Pair-Objekte sind ungleich" << endl;
    }

    aPair2.first = 1.1;

    if (aPair1 != aPair2)
    {
        cout << "Die Pair-Objekte sind ungleich" << endl;
    }
}
```



### 2.4.3 Der pair Operator <

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch kleiner ist, als der des rechten Operanden (**pair**-Objekte gleichen Typs). Dazu werden die beiden Datenelemente jeweils miteinander verglichen, indem deren Vergleichsoperator „<“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch kleiner ist als der rechte Operand.

Syntax:

```
bool operator< (const pair<T,U>& p1,
               const pair<T,U>& p2);
```



```
//=====
// PROGRAMM: PAIR_BSP_07
//=====

#include <iomanip.h>
#include <iostream.h>
#include <utility>

using namespace std;

void main (void)
{
    typedef pair<double, double> mypairtype;
    mypairtype aPair1;
    mypairtype aPair2;

    aPair1.first  = 1.1;
    aPair1.second = 1.1;
    aPair2.first  = 1.2;
    aPair2.second = 1.1;

    if (aPair1 < aPair2)
    {
        cout << "aPair1 ist kleiner" << endl;
    }
}
```

#### 2.4.4 Der pair Operator >

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch größer ist, als der des rechten Operanden (**pair**-Objekte gleichen Typs). Dazu werden die beiden Datenelemente jeweils miteinander verglichen, indem deren Vergleichsoperator „>“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch größer ist als der rechte Operand.

Syntax:

```
bool operator> (const pair<T,U>& p1,
               const pair<T,U>& p2);
```



```
//=====
// PROGRAMM: PAIR_BSP_08
//=====

#include <iomanip.h>
#include <iostream.h>
#include <utility>

using namespace std;

void main (void)
{
    typedef pair<double, double> mypairtype;
    mypairtype aPair1;
    mypairtype aPair2;

    aPair1.first  = 1.1;
    aPair1.second = 1.1;
    aPair2.first  = 1.2;
    aPair2.second = 1.1;
```

```

    if (aPair2 > aPair1)
    {
        cout << "aPair2 ist größer" << endl;
    }
}

```

#### 2.4.5 Der pair Operator <=

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch kleiner oder gleich dem rechten Operanden ist (**pair**-Objekte gleichen Typs). Dazu werden die beiden Datenelemente jeweils miteinander verglichen, indem deren Vergleichsoperator „<=“ aufgerufen wird. Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch kleiner oder gleich dem rechten Operanden ist.

Syntax:

```

bool operator<= (const pair<T,U>& p1,
                const pair<T,U>& p2);

```

```

//=====
// PROGRAMM: PAIR_BSP_09
//=====

#include <iomanip.h>
#include <iostream.h>
#include <utility>

using namespace std;

void main (void)
{
    typedef pair<double, double> mypairtype;
    mypairtype aPair1;
    mypairtype aPair2;

    aPair1.first  = 1.1;
    aPair1.second = 1.1;
    aPair2.first  = 1.2;
    aPair2.second = 1.1;

    if (aPair1 <= aPair2)
    {
        cout << "aPair1 ist kleiner oder gleich aPair2" << endl;
    }
}

```



#### 2.4.6 Der pair Operator >=

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch größer oder gleich dem rechten Operanden ist (**pair**-Objekte gleichen Typs). Dazu werden die beiden Datenelemente jeweils miteinander verglichen, indem deren Vergleichsoperator „>=“ aufgerufen wird. Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch größer oder gleich dem rechten Operanden ist.

Syntax:

```

bool operator>= (const pair<T,U>& p1,
                const pair<T,U>& p2);

```



```
//=====
// PROGRAMM: PAIR_BSP_10
//=====

#include <iomanip.h>
#include <iostream.h>
#include <utility>

using namespace std;

void main (void)
{
    typedef pair<double, double> mypairtype;
    mypairtype aPair1;
    mypairtype aPair2;

    aPair1.first = 1.1;
    aPair1.second = 1.1;
    aPair2.first = 1.2;
    aPair2.second = 1.1;

    if (aPair2 >= aPair1)
    {
        cout << "aPair2 ist größer oder gleich aPair1" << endl;
    }
}
```

## 2.5 Beispielprogramme

Das folgende Programmbeispiel zeigt den einfachen Umgang mit der **pair** Hilfsklasse:



```
//=====
// PROGRAMM: PAIR_BSP_11
//=====

#include <iomanip.h>
#include <iostream.h>
#include <utility>
#include <set>

using namespace std;

void main (void)
{
    typedef multiset<int, less<int> > mysettype;
    mysettype aMultiset;

    aMultiset.insert(23);
    aMultiset.insert(21);
    aMultiset.insert(20);
    aMultiset.insert(20);
    aMultiset.insert(20);

    pair<mysettype::const_iterator,
        mysettype::const_iterator> aP =
aMultiset.equal_range(22);

    cout << "Lowerbound für 22: " << *(aP.first) << endl;
    cout << "Upperbound für 22: " << *(aP.second) << endl;
}
```

### 3. Die Containerklasse vector

Der **vector** realisiert das in C/C++ fehlende, dynamische Array, optimiert für das schnelle Anfügen von Datensätzen am Ende des Arrays.

Dynamisch bedeutet daß die Größe eines **vector** sich zur Laufzeit des Programms ändern kann. Die dazu notwendige Speicherverwaltung wird automatisch vorgenommen und ist so effizient wie möglich realisiert. Trotz aller effizienten Programmierung ist aber zu bedenken, daß auch ein dynamisches Array einen geschlossenen Speicherbereich benötigt, damit die darauf basierende Pointerarithmetik<sup>1</sup> funktioniert. Um es zu erweitern muß das Feld daher relativ häufig im Speicher komplett umkopiert werden. Das Anfügen von vielen Elementen in einer Schleife wird somit schnell ineffizient.

Wie ein gewöhnliches Array ist **vector** eine komplexe Datenstruktur, die mit Indexwerten angesprochen werden kann. Auch der STL-**vector** beginnt, wie ein normales Array, bei der Indexzählung mit Null.

Da der STL-**vector** den **operator[]** (Zugriff über Indexklammer) korrekt überlädt, kann auf die Inhalte des dynamischen Arrays wie gewohnt zugegriffen werden.

Neue Datenelemente können an beliebiger Stelle eines Vektors eingefügt werden. Allerdings ist zu beachten, das die Erweiterung eines **vector** nur am Ende des dynamischen Arrays mit hoher Effizienz erfolgt.

Soll primär in der Mitte einer Datenstruktur angefügt werden, so empfiehlt sich statt dessen der Einsatz der Containerklasse **list** (die allerdings nicht auf einem Index basiert).

Wird primär am Anfang und am Ende des Arrays angefügt sollte man über den Einsatz der Containerklasse **deque** nachdenken, die wie **vector** auf einem Index basiert, aber ein schnelles Anfügen am Anfang und Ende der Struktur ermöglicht.

#### 3.1 Die vector Constructoren

Die folgende Tabelle faßt die Constructoren der **vector** Containerklasse zusammen. In den nachstehenden Abschnitten werden die Constructoren einzeln behandelt und mit Beispielen erläutert.

<b>Constructoren der Containerklasse VECTOR</b>	
<i>Constructor</i>	<i>Bedeutung</i>
<code>vector&lt;T&gt; VarName;</code>	Standard-Constructor, erzeugt ein neues, leeres <b>vector</b> -Objekt
<code>vector&lt;T&gt; VarName (Size n);</code>	Dieser Constructor erzeugt einen neuen <b>vector</b> mit <b>n</b> vordefinierten Elementen vom Typ <b>T</b> . Für jedes der <b>n</b> Elemente wird der Standard-Constructor von <b>T</b> aufgerufen.
<code>vector&lt;T&gt; VarName (Size n, T x);</code>	Dieser Constructor erzeugt einen neuen <b>vector</b> mit <b>n</b> vordefinierten Elementen vom Typ <b>T</b> . Für jedes der <b>n</b>

<sup>1</sup> Die Pointerarithmetik ist nachzulesen im Kapitel über Arrays (Grundkurs-Skript)

<b>Constructoren der Containerklasse VECTOR</b>	
<i>Constructor</i>	<i>Bedeutung</i>
	Elemente wird der Copy-Constructor von <b>T</b> aufgerufen und somit werden insgesamt <b>n</b> Kopien des Objektes <b>x</b> erzeugt.
vector<T> VarName (T* pFirst, T* pLast);	Dieser Constructor erzeugt einen neuen <b>vector</b> anhand eines bereits bestehenden Arrays. Die Pointer <b>pFirst</b> und <b>pLast</b> bezeichnen dabei Anfang- und Endwert eines Bereiches. Das Element, auf welches der Zeiger <b>pLast</b> verweist gehört <b>nicht</b> mehr zum kopierten Bereich.
vector<T> VarName (const vector<T>& v);	Dieser Constructor erzeugt einen neuen <b>vector</b> anhand eines bereits bestehenden <b>vector</b> -Objektes. Das neue <b>vector</b> -Objekt ist eine vollständige Kopie des übergebenen <b>vector</b> -Objektes <b>v</b> .

Tabelle 3-1– Constructoren der Containerklasse vector

### 3.1.1 Der vector Standard-Constructor

Der **vector** Standard-Constructor, erzeugt ein neues, leeres **vector**-Objekt vom Typ **T**<sup>2</sup>.

Syntax:

```
vector<T> Variablenname;
```

Das folgende Beispielprogramm zeigt den Einsatz des Standard-Constructors innerhalb eines Programms:



```
//=====
// PROGRAMM: VECTOR_BSP_01
//=====

#include <vector>

using namespace std;

void main (void)
{
    vector<int> aMyIntVector;
}
```

### 3.1.2 Der vector Constructor mit Größenangabe

Dieser Constructor erzeugt einen neuen **vector** mit **n** vordefinierten Elementen vom Typ **T**. Für jedes der **n** Elemente wird der Standard-Constructor von **T** (zwecks Initialisierung) aufgerufen.

Syntax:

<sup>2</sup> **T** ist der Platzhalter für den im **vector** verwalteten Datentyp

```
vector<T> Variablenname (size_type3 n);
```

Das folgende Beispielprogramm zeigt den Einsatz des Constructors innerhalb eines Programms:

```
//=====
// PROGRAMM: VECTOR_BSP_02
//=====

#include <vector>

using namespace std;

void main (void)
{
    vector<int> aMyIntVector (5); // mit 5 Elementen beginnen
}
```



Diese Form des Constructors ist besonders nützlich, wenn man bereits eine ungefähre Menge an zu erwartenden Elementen abschätzen kann. Durch die Erzeugung eines großen Arrays zu Beginn kann die oben angesprochene Ineffizienz durch häufiges Umkopieren im Speicher leicht vermieden werden.

### 3.1.3 Der vector Constructor mit Größenangabe und Vorlage

Dieser Constructor erzeugt einen neuen **vector** mit **n** vordefinierten Elementen vom Typ **T**. Für jedes der **n** Elemente wird der Copy-Constructor von **T** (zwecks Initialisierung) aufgerufen und somit werden insgesamt **n** Kopien des Objektes **x** erzeugt.

Syntax:

```
vector<T> Variablenname (size_type n, T& x);
```

Das folgende Beispielprogramm zeigt den Einsatz des Constructors innerhalb eines Programms:

```
//=====
// PROGRAMM: VECTOR_BSP_03
//=====

#include <vector>

using namespace std;

void main (void)
{
    int Vorlage = 17;
    vector<int> aMyIntVector (5, Vorlage);
}
```



Auch diese Form des Constructors ist nützlich, wenn man bereits eine ungefähre Menge an zu erwartenden Elementen abschätzen kann.

<sup>3</sup> Logischer Größentyp für eine Anzahl von Elementen vom Typ **T**

Wie beim Constructor mit Größenangabe (s.o.) kann durch die Erzeugung eines großen Arrays zu Beginn die Ineffizienz durch häufiges Umkopieren vermieden werden. Zudem hat man hier die Möglichkeit, eine gezielte Vorbelegung für alle zu erzeugenden Elemente zu treffen, die von der Vorbelegung durch den Standard-Constructor der zu speichernden Elemente **T** abweicht.

### 3.1.4 Der vector Constructor mit Bereichsangabe

Dieser Constructor erzeugt einen neuen **vector** anhand eines bereits bestehenden Arrays. Die Pointer **pFirst** und **pLast** bezeichnen dabei Anfang- und Endwert eines Bereiches. Das Element, auf welches der Zeiger **pLast** verweist, gehört *nicht* mehr zum kopierten Bereich – d.h. **pLast** verweist auf das erste Element *hinter* dem zu kopierenden Bereich.

Für jedes der im Bereich befindlichen Elemente wird einmal der Copy-Constructor von **T** mit dem korrespondierenden Element des übergebenen Bereiches aufgerufen.

Syntax:

```
vector<T> Variablenname (T* pFirst, T* pLast);
```

Das folgende Beispielprogramm zeigt, wie der Constructor innerhalb eines Programms verwendet werden kann:



```
//=====
// PROGRAMM: VECTOR_BSP_04
//=====

#include <vector>

using namespace std;

int Feld [7] = {1, 2, 3, 4, 5, 6, 7};

void main (void)
{
    int Wert = 13;
    vector<int> aTest (6, Wert);

    // 1. Die Elemente 0..2 kopieren
    vector<int> aMyIntVec1 (aTest.begin(), aTest.begin()+3);

    // 2. den gesamten vector kopieren
    vector<int> aMyIntVec2 (aTest.begin(), aTest.end());

    // 3. Index 1..4 aus einen einfachen Array kopieren
    vector<int> aMyIntVec3 (Feld+1, Feld+5);
}
```

Anhand der übergebenen Pointer kann der Constructor die benötigte Anzahl an Elementen ermitteln und eine entsprechend großen Block reservieren. Da der Copy-Constructor von **T** nur einmal für jedes Element aufgerufen wird (im Unterschied zum Constructor mit Größenangabe und Vorlage sind es hier verschiedene Objekte des

Typs **T** die kopiert werden), ist diese Constructor-Form ebenso effizient wie der bereits erwähnte Constructor mit Größenangabe und Vorlage.

### 3.1.5 Der vector Copy-Constructor

Dieser Constructor erzeugt einen neuen **vector** anhand eines bereits bestehenden **vector**-Objektes. Das neue **vector**-Objekt ist eine vollständige Kopie des übergebenen **vector**-Objektes **v**.

Syntax:

```
vector<T> Variablenname (const vector<T>& v);
```

Das folgende Beispielprogramm zeigt, wie der Copy-Constructor verwendet werden kann:

```
//=====
// PROGRAMM: VECTOR_BSP_05
//=====

#include <vector>

using namespace std;

void main (void)
{
    vector<int> aMyIntVec1 (3);

    // den vector kopieren
    vector<int> aMyIntVec2 (aMyIntVec1);
}
```



Da auch bei diesem Constructor die benötigte Anzahl an Elementen ermittelt und ein entsprechend großer Block reserviert werden kann, ist er ebenfalls sehr viel effizienter als eine Kopie Datenelement für Datenelement.

## 3.2 Die vector Methoden

Die folgende Tabelle faßt die Methoden der **vector** Containerklasse zusammen. In den nachstehenden Abschnitten werden die Methoden einzeln behandelt und mit Beispielen erläutert.

<b>Methoden der Containerklasse VECTOR</b>	
<i> Methode </i>	<i> Bedeutung </i>
back	Gibt eine Referenz auf das letzte im <b>vector</b> enthaltene Datenelement zurück (also DatentypT&)
begin	Rückgabe eines <b>iterator</b> , der auf das erste im <b>vector</b> enthaltene Element zeigt
capacity	Gibt die Anzahl der Elemente zurück, die der <b>vector</b> aufnehmen kann, bevor neuer Speicherplatz angefordert werden muß
empty	Gibt den Wert <b>true</b> zurück, wenn der <b>vector</b> keine Datenelemente enthält.
end	Rückgabe eines <b>iterator</b> , der hinter das letzte im <b>vector</b> enthaltene Element zeigt

<b>Methoden der Containerklasse VECTOR</b>	
<i>Methode</i>	<i>Bedeutung</i>
erase	Löscht ein oder mehrere Datenelemente an der angegebenen Position aus dem <b>vector</b>
front	Gibt eine Referenz auf das erste im <b>vector</b> enthaltene Datenelement zurück (also DatentypT&)
insert	Fügt ein oder mehrere Datenelemente an der angegebenen Position in den <b>vector</b> ein
max_size	Gibt die maximale Anzahl an Datenelementen zurück, die der <b>vector</b> enthalten kann
pop_back	Löscht das letzte Datenelement aus dem <b>vector</b>
push_back	Fügt ein Datenelement am Ende des <b>vector</b> an
rbegin	Rückgabe eines <b>reverse_iterator</b> , der auf das letzte im <b>vector</b> enthaltene Element zeigt
rend	Rückgabe eines <b>reverse_iterator</b> , der vor das erste im <b>vector</b> enthaltene Element zeigt
reserve	Anweisung an den <b>vector</b> eine bestimmte Menge an Speicherplatz zu reservieren, so daß die angegebene Menge an Datenelementen gespeichert werden kann, ohne daß neuer Speicherplatz angefordert werden muß. Die Methode <b>reserve</b> beeinflusst den Rückgabewert von <b>capacity</b> aber nicht den Wert von <b>size</b>
size	Gibt die Anzahl von Datensätzen zurück, die aktuell im <b>vector</b> enthalten sind
swap	Tauscht den Inhalt zweier <b>vector</b> -Objekte aus

Tabelle 3-1– Methoden der Containerklasse vector

### 3.2.1 Die vector Methode back

Die Methode **back()** gibt eine Referenz oder einen **const**-Referenz auf das letzte im **vector** gespeicherte Datenelement zurück. Im Gegensatz zu **end()** handelt es sich um eine Objektreferenz vom Typ **T** und nicht einen **iterator** (welcher zudem hinter das letzte Element zeigen würde). Der Aufruf von **back()** selbst verändert den Inhalt des dynamischen Arrays nicht.

Syntax:

```
T& back ();
const T& back () const;
```

Das nachfolgende Beispielprogramm zeigt den Unterschied zwischen **back()** und **end()** anhand einer Ausgabeanweisung:



```
//=====
// PROGRAMM: VECTOR_BSP_06
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>

using namespace std;

void main (void)
```

```

{
    vector<double> aVect;
    vector<double>::const_iterator aI;

    aVect.push_back (7.0);

    aI = aVect.end()-1;
    cout << *aI << endl;           // Ausgabe Dereferenzierung
von aI
    cout << aVect.back() << endl; // Ausgabe über Referenz
}

```

### 3.2.2 Die vector Methode begin

Rückgabe eines **iterator** oder **const\_iterator**, der auf das erste im **vector** enthaltene Element zeigt.

```

Syntax:
    iterator begin ();
    const_iterator begin () const;

```

Die wohl häufigste Anwendung der Methode **begin()** liegt in der Verarbeitung von Daten durch Schleifen.

Der Aufruf von **begin()** verändert den Inhalt des dynamischen Arrays nicht.

```

//=====
// PROGRAMM: VECTOR_BSP_07
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>

using namespace std;

void main (void)
{
    vector<double> aVect;
    vector<double>::const_iterator aI;

    aVect.push_back (7.0);
    aVect.push_back (1.2);
    aVect.push_back (3.14);

    for (aI=aVect.begin(); aI<aVect.end(); aI++)
    {
        cout << "Wert: " << *aI << endl;
    }
}

```



### 3.2.3 Die vector Methode capacity

Gibt die Anzahl der Elemente zurück, die der **vector** insgesamt aufnehmen kann, bevor neuer Speicherplatz angefordert werden muß.

```

Syntax:
    size_type capacity () const;

```

Es ist zu beachten, daß **capacity()** nicht die aktuell im **vector** enthaltene Anzahl an Datenelementen liefert, sondern die mögliche Anzahl, bevor zusätzlicher Speicherplatz angefordert werden muß. Aus Gründen der Verarbeitungsgeschwindigkeit sind viele **vector**-Implementationen so ausgelegt, daß bei jeder Anforderung von Speicherplatz ein zusätzlicher Pufferbereich angefordert wird. Dadurch wird vermieden daß bei jedem Hinzufügen eines Datensatzes das gesamte dynamische Array im Speicher umkopiert werden muß.

Der Aufruf von **capacity()** verändert den Inhalt des dynamischen Arrays nicht.

Die noch vorhandene Reserve an Speicherplatz kann durch Subtraktion der Rückgabewerte von **capacity()** und **size()** ermittelt werden:



```
//=====
// PROGRAMM: VECTOR_BSP_08
//=====

#include <iostream.h>
#include <vector>

using namespace std;

void main (void)
{
    vector<double> aVect;

    aVect.push_back (7.0);
    aVect.push_back (3.14);

    cout << "Reserve: " << (int)(aVect.capacity() -
aVect.size());
}
```

### 3.2.4 Die vector Methode empty

Die Methode **empty()** prüft, ob der **vector** Datenelemente enthält oder nicht. Sind Datenelemente enthalten (dies entspricht einer Rückgabe von **size()** größer Null), so wird der Wert **false** zurückgegeben, sind keine Datenelemente vorhanden ist der Rückgabewert **true**.

Der Aufruf von **empty()** verändert den Inhalt des dynamischen Arrays nicht.

Syntax:  

```
bool empty () const;
```



```
//=====
// PROGRAMM: VECTOR_BSP_09
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>

using namespace std;

void main (void)
{
```

```

vector<double> aVect;

if (aVect.empty())
{
    cout << "Vector ist leer" << endl;
}
else
{
    cout << "Vector enthält Daten" << endl;
}

aVect.push_back (7.0);
aVect.push_back (3.14);

if (aVect.empty())
{
    cout << "Vector ist leer" << endl;
}
else
{
    cout << "Vector enthält Daten" << endl;
}
}

```

### 3.2.5 Die vector Methode end

Rückgabe eines **iterator** oder **const\_iterator**, der *hinten* das letzte im **vector** enthaltene Element zeigt.

Syntax:

```

iterator end ();
const_iterator end () const;

```

Die wohl häufigste Anwendung der Methode **end()** liegt in der Verarbeitung von Daten durch Schleifen.

Es ist zu beachten, daß **end()** hinter das letzte gültige Datenelement zeigt, alle Schleifen müssen also auf „ungleich“ (Operator !=) oder „echt kleiner als“ prüfen (Operator <) und nicht auf „kleiner oder gleich“ (Operator <=). Zeigt ein **iterator** auf **end()** führt jeder dereferenzierende Zugriff zu einer Exception vom Typ **out\_of\_range**. Wird die Exception nicht abgefangen (wie zweiten Beispiel), bricht das Programm unkontrolliert ab (Absturz).

Der Aufruf von **end()** verändert den Inhalt des dynamischen Arrays nicht.

```

//=====
// PROGRAMM: VECTOR_BSP_10
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>

using namespace std;

void main (void)
{
    vector<double> aVect;

```



```

vector<double>::const_iterator aI;

aVect.push_back (7.0);
aVect.push_back (1.2);
aVect.push_back (3.14);

for (aI=aVect.begin(); aI<aVect.end(); aI++)
{
    cout << "Wert: " << *aI << endl;
}
}

```

Das folgende Beispiel zeigt die Auslösung einer Exception aufgrund eines illegalen Zugriffs auf `end()`.



```

//=====
// PROGRAMM: VECTOR_BSP_11
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>

using namespace std;

void main (void)
{
    vector<double> aVect;
    vector<double>::const_iterator aI;

    aVect.push_back (7.0);
    aVect.push_back (1.2);
    aVect.push_back (3.14);

    try
    {
        aI = aVect.begin();
        cout << "Wert: " << *aI << endl;
        aI = aVect.end();
        cout << "Wert: " << *aI << endl;
    }
    catch (out_of_range& aError) // wirkt nicht bei jedem
Compiler
    {
        cout << "Zugriff auf illegales Element" << endl;
    }
}

```

### 3.2.6 Die vector Methode erase

Löscht ein oder mehrere Datenelemente an der angegebenen Position aus dem **vector**.

Syntax:

```

void erase(iterator toDel);
void erase(iterator First, iterator Last);

```

Die Methode **erase()** kann verwendet werden, um einzelne Datensätze oder ganze Bereiche zu löschen.

Durch das Löschen werden alle Iteratoren – sowie Verweise und Referenzen auf Datensätze im gelöschten Bereich – ungültig, da es sich um Pointer handelt und das Array gegebenenfalls im Speicher verschoben wurde.

Es ist daher darauf zu achten, daß alle Iteratoren, Verweise und Referenzen nach einem Befehl, der die Größe des Vektors verändert, neu ermittelt werden müssen.

Die erste Form der Methode sorgt dafür, daß ein einzelner Datensatz aus dem **vector** gelöscht und der Destructor der Klasse **T** für diesen Datensatz aufgerufen wird.

Bei der zweiten Syntaxform wird ein zu löschender Bereich angegeben, für jedes zu löschende Element wird der Destructor der Klasse **T** aufgerufen. Die Angabe erfolgt durch zwei Iteratoren, die auf das erste und das letzte zu löschende Element zeigen.

Es ist besonders bei den Bereichsangaben zu beachten, daß nicht versehentlich eine **out\_of\_range** Exception erzeugt wird.

So bricht z.B. der folgende Code, der dazu gedacht ist den gesamten **vector** löschen soll mit einem Fehler ab:

```
//=====
// PROGRAMM: VECTOR_BSP_12
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>

using namespace std;

void main (void)
{
    vector<double> aVect;
    vector<double>::iterator aI;

    aVect.push_back (7.0);
    aVect.push_back (1.2);
    aVect.push_back (3.14);

    try
    {
        aI = aVect.begin()+1; // Auf 2. Element zeigen
        aVect.erase (aI);    // 2. Element löschen
        aVect.erase (aVect.begin(), aVect.end()); // Fehler
    }
    catch (out_of_range& aError) // Wirkt nicht bei jedem
Compiler
    {
        cout << "Zugriff auf illegales Element" << endl;
    }
}
```



Der Fehler liegt hierbei in der irrigen (wie leicht zu treffenden) Annahme, daß – analog zur Methode **begin()**, welche auf das erste

Datenelement im **vector** verweist – die Methode **end()** einen Zeiger auf das letzte Datenelement zurückgibt. Die Methode **end()** zeigt jedoch *hinter* den letzten gültigen Datensatz.

Da die Methode **erase** auch für das Objekt auf das der Parameter **Last** zeigt den Destructor aufzurufen (hier mit dem **iterator** auf **end()** gefüllt), wird ein versuch den Destructor eines Null-Pointers auszuführen, was natürlich eine Exception auslösen muß.

Das letzte gültige Element eines **vector** läßt sich mit **end()-1** ermitteln. Richtig muß das Programm daher lauten:



```
//=====
// PROGRAMM: VECTOR_BSP_13
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>

using namespace std;

void main (void)
{
    vector<double> aVect;
    vector<double>::iterator aI;

    aVect.push_back (7.0);
    aVect.push_back (1.2);
    aVect.push_back (2.4);
    aVect.push_back (3.14);

    try
    {
        aI = aVect.begin()+1; // Auf 2. Element zeigen
        aVect.erase (aI);    // 2. Element löschen
        aVect.erase (aVect.begin(), aVect.end()-1); // Richtig
    }
    catch (out_of_range& aError)
    {
        cout << "Zugriff auf illegales Element" << endl;
    }
}
```

### 3.2.7 Die vector Methode front

Die Methode **front()** gibt eine Referenz oder einen **const**-Referenz auf das erste im **vector** gespeicherte Datenelement zurück. Im Gegensatz zu **begin()** handelt es sich um eine Objektreferenz vom Typ **T** und nicht einen **iterator**.

Der Aufruf von **front()** selbst verändert den Inhalt des dynamischen Arrays nicht.

Syntax:

```
T& front ();
const T& front () const;
```

Das nachfolgende Beispielprogramm zeigt den Unterschied zwischen **front()** und **begin()** anhand einer Ausgabeanweisung:

```

//=====
// PROGRAMM: VECTOR_BSP_14
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>

using namespace std;

void main (void)
{
    vector<double> aVect;
    vector<double>::const_iterator aI;

    aVect.push_back (7.0);

    aI = aVect.begin();
    cout << *aI << endl;           // Ausgabe Dereferenzierung
v. aI
    cout << aVect.front() << endl; // Ausgabe über Referenz
}

```



### 3.2.8 Die vector Methode insert

Fügt ein oder mehrere Datenelemente vor der angegebenen Position in den **vector** ein.

Syntax:

```

iterator insert (iterator pos, T& value);
void          insert (iterator pos, size_type Anz,
                    T& value);
void          insert (iterator pos, const T *First,
                    const T *Last);

```

Die Methode **insert()** kann verwendet werden um einzelne Datensätze oder ganze Bereiche in einen **vector** einzufügen.

Da ein **vector** wie ein Array aufgebaut ist, müssen beim Einfügen von Datensätzen zumindest die nachfolgenden Datenelemente im **vector** umkopiert werden, was die Methode **insert()** für Vektoren (insbesondere große Vektoren) relativ ineffizient macht. Gegebenenfalls muß sogar der gesamte **vector** umkopiert werden (Reallokation), da keine Pufferbereiche mehr vorhanden sind (siehe Methoden **capacity()** und **reserve()**).

Wird eine Reallokation notwendig, so ist zu beachten, daß alle Iteratoren, Verweise und Referenzen ungültig werden, da es sich um Pointer handelt und das Array gegebenenfalls im Speicher verschoben wurde.

Es ist ferner darauf zu achten, daß alle Iteratoren, Verweise und Referenzen nach einem Befehl, der die Größe des Vektors verändert, neu ermittelt werden müssen.

Die erste Form der Methode sorgt dafür, daß ein einzelner Datensatz (**value**) vor der angegebenen Position (**pos**) in den **vector** eingefügt wird.

Dazu wird der Copy-Constructor der Klasse **T** für den Parameter **value** aufgerufen und die so entstandene Kopie in den **vector** eingefügt.

Die zweite Form der **insert()** Methode fügt mehrere (**Anz**) Kopien des Datensatzes **value** an der Position **pos** ein. Der Copy-Constructor der Klasse **T** wird **Anz**-mal für den Parameter **value** aufgerufen und die so entstandenen Kopien in den **vector** eingefügt.

Mit der dritten Methode können Bereiche aus einem anderen Array in den **vector** kopiert werden (hier anhand eines Ausschnittes aus dem Array **fArray** dargestellt).

Der Bereich wird durch Anfangs- und Endadresse festgelegt. Es sollte darauf geachtet werden, daß die Endadresse nicht vor der Anfangsadresse liegen darf und daß beide Adressen zum gleichen Array bzw. gleichen **vector** gehören.



```
//=====
// PROGRAMM: VECTOR_BSP_15
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>

using namespace std;

double fArray [6] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0};

void main (void)
{
    vector<double> aVect;
    vector<double> aV2;
    vector<double>::iterator aI;

    aVect.push_back (1.1);
    aVect.push_back (2.2);
    aVect.push_back (3.3);

    aV2.push_back (11.1);
    aV2.push_back (22.2);
    aV2.push_back (33.3);

    //-----
    // 1. Form
    //-----
    aI = aVect.begin()+1;      // Auf 2. Element zeigen
    aVect.insert (aI, 4.4);    // vor 2. Element einfügen

    //-----
    // 2. Form
    //-----
    aI = aVect.begin()+1;      // wegen möglicher Reallokation
    aVect.insert (aI, 6, 5.5); // 6 Kopien einfügen

    //-----
}
```

```

// 3. Form
//-----
aI = aVect.begin()+1;      // wegen möglicher Reallokation
aVect.insert(aI, fArray+1, fArray+4); // Elemente aus Array

// 3. Form wie oben, aber mit Indexschreibweise
aI = aVect.begin()+1;      // wegen möglicher Reallokation
aVect.insert(aI, &fArray[1], &fArray[4]); // Elem aus Array

// 3. Form aber aus einem anderen Vektor
aI = aVect.begin()+4;      // wegen möglicher Reallokation
aVect.insert(aI, &aV2[0], &aV2[5]); // Elemente aus Vektor

for (aI=aVect.begin(); aI<aVect.end(); aI++)
{
    cout << *aI << endl;
}
}

```

### 3.2.9 Die vector Methode max\_size

Gibt die maximale Anzahl an Datenelementen zurück, die der **vector** insgesamt enthalten kann.

Syntax:

```
size_type max_size () const;
```

Der Wert ist abhängig von der Implementation und vom verwendeten Betriebssystem. Bei modernen Compilern und entsprechender Einstellung kann man davon ausgehen, dass 32-Bit Adressen verwendet werden, also theoretisch 4 Gigabytes angesprochen werden können.

Beschränkt wird die Anzahl der Datenelemente durch die Tatsache, daß der zu verwendende Speicher (wie bei jedem Array) als ein ungeteilter Block vorhanden sein muß. D.h. die Größe des Arrays wird durch alle im RAM-Speicher befindlichen Daten begrenzt, die Größe der zu verwaltenden Datenelemente, sowie die Segmentierung (Zersplitterung) des Speichers.

Die Methode **max\_size()** dient dazu abschätzen zu können, ob ein aufzubauender **vector** überhaupt in den Speicher paßt.

Der Aufruf der Methode verändert den Inhalt des dynamischen Arrays nicht.

```

//=====
// PROGRAMM: VECTOR_BSP_16
//=====

#include <iostream.h>
#include <vector>

using namespace std;

void main (void)
{
    vector<double> aVect;

    cout << "Vektor kann " << aVect.max_size ()
         << " Datenelemente aufnehmen";
}

```



### 3.2.10 Die vector Methode pop\_back

Löscht das letzte Datenelement aus dem **vector**.

Syntax:

```
void pop_back ();
```

Die Methode **pop\_back()** wird verwendet um einen einzelnen Datensatz vom Ende des **vector** zu entfernen. Sollen mehrere Datensätze am Ende des **vector** gelöscht werden, ist die Methode **erase()** günstiger, als ein wiederholter Aufruf von **pop\_back()**.

Durch das Löschen werden daß alle Iteratoren – sowie Verweise und Referenzen auf Datensätze im gelöschten Bereich – ungültig, da es sich um Pointer handelt und das Array gegebenenfalls im Speicher verschoben wurde.

Es ist daher darauf zu achten, daß alle Iteratoren, Verweise und Referenzen nach einen Befehl, der die Größe des Vektors verändert, neu ermittelt werden müssen.

Zum Löschen wird der Destructor der Klasse **T** für das letzte Element im **vector** aufgerufen.



```
//=====
// PROGRAMM: VECTOR_BSP_17
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>

using namespace std;

void main (void)
{
    vector<double> aVect;
    vector<double>::iterator aI;

    aVect.push_back (1.1);
    aVect.push_back (2.2);
    aVect.push_back (3.3);

    aVect.pop_back ();

    for (aI=aVect.begin(); aI<aVect.end(); aI++)
    {
        cout << *aI << endl;
    }
}
```

### 3.2.11 Die vector Methode push\_back

Fügt ein Datenelement am Ende des Vektors an.

Syntax:

```
void push_back (const T& value);
```

Die Methode **push\_back()** wird verwendet um einen einzelnen Datensatz an das Ende des **vector** einzufügen. Sollen mehrere Datensätze am Ende des **vector** angefügt werden, ist die Methode **insert()** günstiger, als ein wiederholter Aufruf von **push\_back()**.

Da ein **vector** wie ein Array aufgebaut ist, muß beim Anfügen eines Datensatzes eventuell eine Reallokation (s.o.) durchgeführt werden, da keine Pufferbereiche mehr vorhanden sind (siehe Methoden **capacity()** und **reserve()**).

Wird eine Reallokation notwendig, so ist zu beachten, daß alle Iteratoren, Verweise und Referenzen ungültig werden, da es sich um Pointer handelt und das Array gegebenenfalls im Speicher verschoben wurde.

Es ist ferner darauf zu achten, daß alle Iteratoren, Verweise und Referenzen nach einem Befehl, der die Größe des Vektors verändert, neu ermittelt werden müssen.

Zum Anfügen wird der Copy-Constructor der Klasse **T** für den Parameter **value** aufgerufen und die so entstandene Kopie an den **vector** als letzter Datensatz angehängt.

```
//=====
// PROGRAMM: VECTOR_BSP_18
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>

using namespace std;

void main (void)
{
    vector<double> aVect;
    vector<double>::iterator aI;

    aVect.push_back (1.1);
    aVect.push_back (2.2);

    for (aI=aVect.begin(); aI<aVect.end(); aI++)
    {
        cout << *aI << endl;
    }
}
```



### 3.2.12 Die vector Methode rbegin

Rückgabe eines **reverse\_iterator** oder **const\_reverse\_iterator**, der auf das letzte im **vector** enthaltene Element zeigt.

Syntax:

```
reverse_iterator rbegin ();
const_reverse_iterator rbegin () const;
```

Die wohl häufigste Anwendung der Methode **rbegin()** liegt in der Verarbeitung von Daten durch rückwärts gerichtete Schleifen. Der Aufruf von **rbegin()** verändert den Inhalt des dynamischen Arrays nicht.



```
//=====
// PROGRAMM: VECTOR_BSP_19
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>

using namespace std;

void main (void)
{
    vector<double> aVect;
    vector<double>::reverse_iterator aI;

    aVect.push_back (7.0);
    aVect.push_back (1.2);
    aVect.push_back (3.14);

    for (aI=aVect.rbegin(); aI!=aVect.rend(); aI++)
    {
        cout << "Wert: " << *aI << endl;
    }
}
```

### 3.2.13 Die vector Methode rend

Rückgabe eines **reverse\_iterator** oder **const\_reverse\_iterator**, der vor das erste im **vector** enthaltene Element zeigt.

Syntax:

```
reverse_iterator rend ();
const_reverse_iterator rend () const;
```

Die wohl häufigste Anwendung der Methode **rend()** liegt in der Verarbeitung von Daten durch rückwärts gerichtete Schleifen.

Es ist zu beachten, daß **rend()** vor das erste gültige Datenelement zeigt, alle Schleifen müssen also auf „ungleich“ prüfen (Operator !=). Zeigt ein **iterator** auf **rend()** führt jeder dereferenzierende Zugriff zu einer Exception vom Typ **out\_of\_range**. Wird die Exception nicht wie zweiten Beispiel abgefangen bricht das Programm unkontrolliert ab (Absturz).

Der Aufruf von **rend()** verändert den Inhalt des dynamischen Arrays nicht.



```
//=====
// PROGRAMM: VECTOR_BSP_20
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
```

```

using namespace std;

void main (void)
{
    vector<double> aVect;
    vector<double>::reverse_iterator aI;

    aVect.push_back (7.0);
    aVect.push_back (1.2);
    aVect.push_back (3.14);

    for (aI=aVect.rbegin(); aI!=aVect.rend(); aI++)
    {
        cout << "Wert: " << *aI << endl;
    }
}

```

Das folgende Beispiel zeigt die Auslösung einer Exception aufgrund eines illegalen Zugriffs auf **rend()**.

```

//=====
// PROGRAMM: VECTOR_BSP_21
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>

using namespace std;

void main (void)
{
    vector<double> aVect;
    vector<double>::reverse_iterator aI;

    aVect.push_back (7.0);
    aVect.push_back (1.2);
    aVect.push_back (3.14);

    try
    {
        aI = aVect.rbegin();
        cout << "Wert: " << *aI << endl;
        aI = aVect.rend();
        cout << "Wert: " << *aI << endl;
    }
    catch (out_of_range& aError)
    {
        cout << "Zugriff auf illegales Element" << endl;
    }
}

```



### 3.2.14 Die vector Methode reserve

Anweisung an den **vector** eine bestimmte Menge an Speicherplatz zu reservieren, so daß die angegebene Menge (**Anzahl**) an Datenelementen gespeichert werden kann, ohne daß neuer Speicherplatz angefordert werden muß.

Syntax:

```
void reserve (size_type Anzahl);
```

Die Methode **reserve()** beeinflusst den Rückgabewert von **capacity()** aber nicht den Wert von **size()**.

Die Methode ist besonders nützlich, wenn die Menge der zu speichernden Werte bereits im Voraus bekannt ist oder gut abgeschätzt werden kann.

Durch die Verwendung von **reserve()** kann die Anzahl der notwendigen Reallokationen (s.o.) gering gehalten werden, so daß sich gegebenenfalls erhebliche Vorteile im Laufzeitverhalten des Programms ergeben.

Durch den Aufruf von `reserve()` wird eine Reallokation notwendig. Es ist zu beachten, daß alle Iteratoren, Verweise und Referenzen ungültig werden, da es sich um Pointer handelt und das Array im Speicher verschoben wurde.

Alle Iteratoren, Verweise und Referenzen müssen nach einem Befehl, der die Größe des Vektors verändert, neu ermittelt werden.



```
//=====
// PROGRAMM: VECTOR_BSP_22
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>

using namespace std;

void main (void)
{
    vector<double> aVect;

    aVect.reserve (1000);

    aVect.push_back (1.2);
    aVect.push_back (3.14);

    cout << "Anzahl bis Reallokation: " << aVect.capacity() <<
endl;
    cout << "Anzahl aktuell          : " << aVect.size() <<
endl;
}
```

### 3.2.15 Die vector Methode size

Gibt die Anzahl von Datensätzen zurück, die aktuell im **vector** enthalten sind.

Syntax:

```
size_type size () const;
```

Der Aufruf von **size()** verändert den Inhalt des dynamischen Arrays nicht.

```
//=====
// PROGRAMM: VECTOR_BSP_23
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>

using namespace std;

void main (void)
{
    vector<double> aVect;

    aVect.push_back (1.2);
    aVect.push_back (3.14);

    cout << "Anzahl aktuell: " << aVect.size() << endl;
}

```



### 3.2.16 Die vector Methode swap

Tauscht die Inhalte zweier **vector**-Objekte aus. Die Vektoren müssen von gleichem Typ sein, also gleiche Datenelementtypen enthalten.

Syntax:

```
void swap (vector<T>& v);
```

Durch den Aufruf von `swap()` wird wahrscheinlich für beide Vektoren eine Reallokation notwendig. Es ist zu beachten, daß somit alle Iteratoren, Verweise und Referenzen ungültig werden, da es sich um Pointer handelt und die Arrays im Speicher verschoben wurden.

Alle Iteratoren, Verweise und Referenzen müssen nach einem Befehl, der die Größe des Vektors verändert, neu ermittelt werden.

```
//=====
// PROGRAMM: VECTOR_BSP_24
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>

using namespace std;

void main (void)
{
    vector<double> aVect;
    vector<double> aV2;
    vector<double>::iterator aI;

    aVect.push_back (1.2);

```



```

aVect.push_back (3.14);

aV2.push_back (222.22);
aV2.push_back (333.33);
aV2.push_back (111.11);
aV2.push_back (444.44);

for (aI=aVect.begin(); aI<aVect.end(); aI++)
{
    cout << "aVect Wert: " << *aI << endl;
}

for (aI=aV2.begin(); aI<aV2.end(); aI++)
{
    cout << "aV2 Wert: " << *aI << endl;
}

aVect.swap (aV2);

for (aI=aVect.begin(); aI<aVect.end(); aI++)
{
    cout << "aVect Wert: " << *aI << endl;
}

for (aI=aV2.begin(); aI<aV2.end(); aI++)
{
    cout << "aV2 Wert: " << *aI << endl;
}
}
    
```

### 3.3 Die vector Operatoren

Mit den folgenden Operatoren kann auf eine Containerklasse vom Typ **vector** zugegriffen werden:

<b>Operatoren der Containerklasse VECTOR</b>	
<i>Operato r</i>	<i>Bedeutung</i>
[ ]	Indezzugriff auf den Inhalt des <b>vector</b>
=	Zuweisung zwischen zwei <b>vector</b> -Objekten gleichen Typs
==	Vergleich zwischen zwei <b>vector</b> -Objekten gleichen Typs
!=	Vergleich zwischen zwei <b>vector</b> -Objekten gleichen Typs
<	Lexikographischer Vergleich zwischen zwei <b>vector</b> -Objekten gleichen Typs
>	Lexikographischer Vergleich zwischen zwei <b>vector</b> -Objekten gleichen Typs
>=	Lexikographischer Vergleich zwischen zwei <b>vector</b> -Objekten gleichen Typs
<=	Lexikographischer Vergleich zwischen zwei <b>vector</b> -Objekten gleichen Typs

Tabelle 3-1– Operatoren der Containerklasse vector

#### 3.3.1 Der vector Operator [ ]

Mittels der eckigen Klammern kann direkt auf ein im **vector** gespeichertes Element zugegriffen werden. Die eckigen Klammern sind im Template der Containerklasse überladen (Operator-Overloading).

Syntax:

```
T& operator[] (int index);
```

Wird ein ungültiger Index angegeben, so wird eine **out\_of\_range** Exception ausgelöst:

```
//=====
// PROGRAMM: VECTOR_BSP_25
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>

using namespace std;

void main (void)
{
    vector<double> aVect;

    aVect.push_back (7.0);
    aVect.push_back (1.2);
    aVect.push_back (3.14);

    try
    {
        cout << "Wert: " << aVect[1] << endl;
        cout << "Wert: " << aVect[50] << endl; // Exception
    }
    catch (out_of_range& aError)
    {
        cout << "Zugriff auf illegales Element" << endl;
    }
}
```



### 3.3.2 Der vector Operator =

Der Operator ersetzt den Inhalt eines **vector**-Objektes durch den Inhalt des zugewiesenen **vector**-Objektes gleichen Typs.

Syntax:

```
vector<T>& operator= (const vector<T>& v);
```

Durch die Zuweisung wird für den linken Operanden (d.h. den Vektor, dem Werte zugewiesen werden) eine Reallokation notwendig. Es ist zu beachten, daß somit alle Iteratoren, Verweise und Referenzen auf diesem Vektor ungültig werden.

Alle Iteratoren, Verweise und Referenzen müssen nach einem Befehl, der die Größe des Vektors verändert, neu ermittelt werden.

```
//=====
// PROGRAMM: VECTOR_BSP_26
//=====

#include <iomanip.h>
#include <iostream.h>
```



```

#include <vector>

using namespace std;

void main (void)
{
    vector<double> aVect;
    vector<double> aV2;
    vector<double>::iterator aI;

    aVect.push_back (7.0);
    aVect.push_back (1.2);
    aVect.push_back (3.14);

    aV2 = aVect;

    for (aI=aVect.begin(); aI<aVect.end(); aI++)
    {
        cout << "aVect Wert: " << *aI << endl;
    }

    for (aI=aV2.begin(); aI<aV2.end(); aI++)
    {
        cout << "aV2 Wert: " << *aI << endl;
    }
}

```

### 3.3.3 Der vector Operator ==

Der Operator prüft, ob die Inhalte des linken und rechten Operanden (**vector**-Objekte gleichen Typs) übereinstimmen. Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn beide Vektoren die gleichen Elemente in der gleichen Reihenfolge enthalten.

Syntax:

```
bool operator==(const vector<T>& v) const;
```



```

//=====
// PROGRAMM: VECTOR_BSP_27
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>

using namespace std;

void main (void)
{
    vector<double> aVect;
    vector<double> aV2;

    aVect.push_back (3.14);

    if (aVect == aV2)
    {
        cout << "Die Vektoren sind gleich" << endl;
    }
    else

```

```

{
    cout << "Die Vektoren sind nicht gleich" << endl;
}

aV2 = aVect;

if (aVect == aV2)
{
    cout << "Die Vektoren sind gleich" << endl;
}
else
{
    cout << "Die Vektoren sind nicht gleich" << endl;
}
}

```

### 3.3.4 Der vector Operator !=

Der Operator prüft, ob die Inhalte des linken und rechten Operanden (**vector**-Objekte gleichen Typs) verschieden sind. Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn beide Vektoren unterschiedliche Elemente oder gleiche Elemente in unterschiedlicher Reihenfolge enthalten.

Syntax:

```
bool operator!=(const vector<T>& v) const;
```

```

//=====
// PROGRAMM: VECTOR_BSP_28
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>

using namespace std;

void main (void)
{
    vector<double> aVect;
    vector<double> aV2;

    aVect.push_back (3.14);

    if (aVect != aV2)
    {
        cout << "Die Vektoren sind nicht gleich" << endl;
    }
    else
    {
        cout << "Die Vektoren sind gleich" << endl;
    }

    aV2 = aVect;

    if (aVect != aV2)
    {
        cout << "Die Vektoren sind nicht gleich" << endl;
    }
}

```



```

    }
    else
    {
        cout << "Die Vektoren sind gleich" << endl;
    }
}

```

### 3.3.5 Der vector Operator <

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch kleiner ist, als der des rechten Operanden (**vector**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „<“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch kleiner ist als der rechte Operand.

Syntax:

```
bool operator< (const vector<T>& v) const;
```



```

//=====
// PROGRAMM: VECTOR_BSP_29
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>

using namespace std;

void main (void)
{
    vector<double> aVect;
    vector<double> aV2;

    aVect.push_back (3.14);
    aV2.push_back (3.15);

    if (aVect < aV2)
    {
        cout << "aVect ist kleiner als aV2" << endl;
    }
    else
    {
        cout << " aVect ist nicht kleiner als aV2" << endl;
    }
}

```

### 3.3.6 Der vector Operator >

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch größer ist, als der des rechten Operanden (**vector**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „>“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch größer ist als der rechte Operand.

Syntax:

```
bool operator> (const vector<T>& v) const;
```

```

//=====
// PROGRAMM: VECTOR_BSP_30
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>

using namespace std;

void main (void)
{
    vector<double> aVect;
    vector<double> aV2;

    aVect.push_back (3.14);
    aV2.push_back   (3.15);

    if (aVect > aV2)
    {
        cout << "aVect ist größer als aV2" << endl;
    }
    else
    {
        cout << " aVect ist nicht größer als aV2" << endl;
    }
}

```



### 3.3.7 Der vector Operator <=

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch kleiner oder gleich mit dem rechten Operanden ist (**vector**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „<=“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch kleiner oder gleich mit dem rechten Operanden ist.

Syntax:

```
bool operator<= (const vector<T>& v) const;
```

```

//=====
// PROGRAMM: VECTOR_BSP_31
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>

using namespace std;

void main (void)
{
    vector<double> aVect;
    vector<double> aV2;

    aVect.push_back (3.14);
    aV2.push_back   (3.15);

    if (aVect <= aV2)

```



```

    {
        cout << "aVect ist kleiner-gleich aV2" << endl;
    }
    else
    {
        cout << " aVect ist nicht kleiner-gleich als aV2" <<
endl;
    }
}

```

### 3.3.8 Der vector Operator >=

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch größer oder gleich mit dem rechten Operanden ist (**vector**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „>=“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch größer oder gleich mit dem rechten Operanden ist.

Syntax:

```
bool operator>= (const vector<T>& v) const;
```



```

//=====
// PROGRAMM: VECTOR_BSP_32
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>

using namespace std;

void main (void)
{
    vector<double> aVect;
    vector<double> aV2;

    aVect.push_back (3.14);
    aV2.push_back (3.15);

    if (aVect >= aV2)
    {
        cout << "aVect ist größer-gleich aV2" << endl;
    }
    else
    {
        cout << " aVect ist nicht größer-gleich als aV2" <<
endl;
    }
}

```

## 3.4 Beispielprogramme

Das folgende Programmbeispiel zeigt den einfachen Umgang mit der **vector** Containerklasse:



```

//=====
// PROGRAMM: VECTOR_BSP_33
//=====

```

```

// Beispielprogramm zur Verwendung der Standard Template
// Library (STL) -Verwendung von VECTOREN (dynamische Arrays)
//-----
// Einfaches Einlesen auf einen Vector und Ausgabe über Index
// und Iterator
//=====

//-----
// Header-Einbindung (alphabetisch)
//-----
#include <iomanip.h>
#include <iostream.h>

//-----
// Ab hier Vorcompilierung und Caching von Headern stoppen, da
// die Header der STL Codedefinitionen und nicht nur
// -deklarationen enthalten
// #pragma hdrstop entfernt das Warning:
//-----
// x.h(nn,n):Cannot create pre-compiled header: code in header
//-----
// Alle Header die KEINE Codedefinitionen enthalten sind vor
// dem Pragma einzubinden
//-----
#pragma hdrstop
#include <vector>      // dynamische Arrays
using namespace std;

//=====
// Hauptprogramm
//=====
void main (void)
{
    //-----
    // Variablen-Deklaration / -definition
    //-----
    vector<int> aMyIntVector;      // dyn. Array vom Typ Integer
    vector<int>::iterator aIter;  // Iterator (Aufzähler)
    int          iEingabe;        // Eingabe-Puffer
    int          iAnzahlInts;    // Anz. der zu lesenden Werte
    int          i;              // Zählvariable
    char         cEnde;          // DOS nicht sofort schließen

    //-----
    // Einlesen der gewünschten Array-Größe zur Laufzeit
    //-----
    cout << "Bitte die gewünschte Anzahl an Integer-Werten"
         << " angeben: ";
    cin  >> iAnzahlInts;

    //-----
    // Einlesen der Arrayelemente
    //-----
    for (i=0; i<iAnzahlInts; i++)
    {
        cout << setw(2) << (i+1) << ". Wert: ";

        //-----
        // Anders als bei statischen Arrays kann man nicht
        // einfach direkt auf dem Array einlesen, da das Array
        // nur genau die Anzahl an Elementen besitzt, die ihm
        // übergeben worden sind. Daher muß man über eine
        // Puffer-Variable (hier iEingabe) gehen, deren Inhalt

```

```

// nach dem Einlesen an den Vector übergeben wird.
// Die Vector-Methode PUSH_BACK fügt den Integer am ENDE
// des dynamischen Arrays an
//-----
cin >> iEingabe;
aMyIntVector.push_back (iEingabe);
}

//-----
// Ausgabe der Arrayelemente über Index-Angabe
//-----
cout << endl << endl << "gespeicherte Werte (über Index): "
    << endl;

for (i=0; i<iAnzahlInts; i++)
{
    cout << setw(2) << (i+1) << ". Wert: "
        << aMyIntVector[i] << endl;
}

//-----
// Ausgabe der Arrayelemente über Iterator (Aufzähler)
//-----
cout << endl << endl
    << "gespeicherte Werte (Iterator-Zugriff): " << endl;
for (aIter=aMyIntVector.begin(); aIter<aMyIntVector.end();
    aIter++)
{
    cout << "Wert: " << *aIter << endl;
}

//-----
// Verhindern, das das DOS-Fenster bei Programmende sofort
// geschlossen wird
//-----
cin >> cEnde;
}

```



```

//=====
// PROGRAMM: VECTOR_BSP_34
//-----
// Beispielprogramm zur Verwendung der Standard Template
// Library (STL) - Verwendung von VECTOREN (dynamische Arrays)
//-----
// Einfaches Einlesen auf einen Vector und Ausgabe über Index
// und Iterator
//=====

//-----
// Header-Einbindung (alphabetisch)
//-----
#include <iomanip.h>
#include <iostream.h>

//-----
// Ab hier Vorcompilierung und Caching von Headern stoppen, da
// die Header der STL Codedefinitionen und nicht nur
// -deklarationen enthalten
// #pragma hdrstop entfernt das Warning:
//-----
// x.h(nn,n):Cannot create pre-compiled header: code in header
//-----

```

```

// Alle Header die KEINE Codedefinitionen enthalten sind vor
// dem Pragma einzubinden
//-----
#pragma hdrstop
#include <algorithm>
#include <vector> // dynamische Arrays
using namespace std;

//-----
// Hauptprogramm
//-----
void main (void)
{
    //-----
    // Variablen-Deklaration / -definition
    //-----
    vector<int> aMyIntVector; // dyn. Array vom Typ Integer
    vector<int>::iterator aIter; // Array-Iterator (Aufzähler)
    int iEingabe; // Eingabe-Puffer
    int iAnzahlInts; // Anz. einzulesende Werte
    int i; // Zählvariable
    char cEnde; // DOS nicht sofort schließen

    //-----
    // Einlesen der gewünschten Array-Größe zur Laufzeit
    //-----
    cout << "Bitte die gewünschte Anzahl an Integer-Werten"
         << " angeben: ";
    cin >> iAnzahlInts;

    //-----
    // Einlesen der Arrayelemente
    //-----
    for (i=0; i<iAnzahlInts; i++)
    {
        cout << setw(2) << (i+1) << ". Wert: ";

        //-----
        // Anders als bei statischen Arrays kann man nicht
        // einfach direkt auf dem Array einlesen, da das Array
        // nur genau die Anzahl an Elementen besitzt, die ihm
        // übergeben worden sind. Daher muß man über eine
        // Puffer-Variable (hier iEingabe) gehen, deren Inhalt
        // nach dem Einlesen an den Vector übergeben wird.
        // Die Vector-Methode PUSH_BACK fügt den Integer am ENDE
        // des dynamischen Arrays an
        //-----
        cin >> iEingabe;
        aMyIntVector.push_back (iEingabe);
    }

    //-----
    // Sortierung über generischen Algorithmus der STL
    // (Quicksort) Dazu ist es nötig die Grenzen des
    // dynamischen Arrays als Iteratoren für den Start- und
    // Endpunkt der Sortierung anzugeben (hier: X.BEGIN() und
    // X.END())
    //-----
    sort (aMyIntVector.begin(), aMyIntVector.end());

    //-----
    // Ausgabe der Arrayelemente über Index-Angabe
    //-----

```

```
    cout << endl << endl
        << "gespeicherte Werte (Indexzugriff): " << endl;

    for (i=0; i<iAnzahlInts; i++)
    {
        cout << setw(2) << (i+1) << ". Wert: "
            << aMyIntVector[i] << endl;
    }

    //-----
    // Ausgabe der Arrayelemente über Iterator (Aufzähler)
    //-----
    cout << endl << endl
        << "gespeicherte Werte (Iterator-Zugriff): " << endl;

    for (aIter=aMyIntVector.begin(); aIter<aMyIntVector.end();
        aIter++)
    {
        cout << "Wert: " << *aIter << endl;
    }

    //-----
    // Verhindern, das das DOS-Fenster bei Programmende sofort
    // geschlossen wird
    //-----
    cin >> cEnde;
}
```

## 4. Die Containerklasse deque

Ein **deque** realisiert einen sequentiellen Behälter, optimiert für den wahlfreien Zugriff über Index und das schnelle Anfügen von Datensätzen am Anfang und Ende.

Ein **deque** hat eine ähnlich dynamische Speicherverwaltung wie ein **vector**. Dies bedeutet daß die Größe eines **deque** sich zur Laufzeit des Programms ändern kann.

Die dazu notwendige Speicherverwaltung wird automatisch vorgenommen und ist so effizient wie möglich realisiert. Trotz aller effizienten Programmierung ist aber zu bedenken, daß (wie beim **vector**) auch für **deque**-Objekte ein geschlossener Speicherbereich benötigt wird, damit die darauf basierende Pointerarithmetik<sup>1</sup> funktioniert. Um das Objekt zu erweitern muß das Feld daher relativ häufig im Speicher komplett umkopiert werden. Das Anfügen von vielen Elementen in einer Schleife wird somit schnell ineffizient.

Implementiert wird ein **deque** zumeist als Array, ähnlich wie ein **vector**, nur daß ein **deque** Speicherplatzreserven am Anfang und am Ende bereithält. Somit kann schnell eingefügt und der Aufwand für Reallokationen möglichst gering gehalten werden.

Ähnlich wie ein gewöhnliches Array und der **vector** ist ein **deque** eine komplexe Datenstruktur, die über Indexwerte angesprochen werden kann (wahlfreier Zugriff). Wie bei jedem Indexzugriff beginnt die Indexzählung (erstes Element) mit Null.

Der **operator[]** (Zugriff über Indexklammer) wird korrekt überladen, so daß auf den Inhalt des **deque** wie auf ein Array zugegriffen werden kann.

Neue Datenelemente können an beliebiger Stelle eines Containerobjektes eingefügt werden. Allerdings ist zu beachten, daß die Erweiterung eines **deque** nur am Anfang und Ende des mit hoher Effizienz erfolgt.

Soll primär in der Mitte einer Datenstruktur angefügt werden, so empfiehlt sich statt dessen der Einsatz der Containerklasse **list** (die allerdings nicht auf einem Index basiert).

### 4.1 Die deque Constructoren

Die folgende Tabelle faßt die Constructoren der Containerklasse **deque** zusammen. In den nachstehenden Abschnitten werden die Constructoren einzeln behandelt und mit Beispielen erläutert.

<b>Constructoren der Containerklasse DEQUE</b>	
<i>Constructor</i>	<i>Bedeutung</i>
<code>deque&lt;T&gt; VarName;</code>	Standard-Constructor, erzeugt ein neues, leeres <b>deque</b> -Objekt
<code>deque&lt;T&gt; VarName (Size n);</code>	Dieser Constructor erzeugt ein neues <b>deque</b> mit <b>n</b> vordefinierten Elementen vom Typ <b>T</b> . Für jedes der <b>n</b> Elemente wird der Standard-Constructor von <b>T</b> aufgerufen.
<code>deque&lt;T&gt; VarName (Size n, T x);</code>	Dieser Constructor erzeugt ein neues <b>deque</b> mit <b>n</b> vordefinierten Elementen

<sup>1</sup> Die Pointerarithmetik ist nachzulesen im Kapitel über Arrays (Grundkurs-Skript)

<b>Constructoren der Containerklasse DEQUE</b>	
<i>Constructor</i>	<i>Bedeutung</i>
	vom Typ <b>T</b> . Für jedes der <b>n</b> Elemente wird der Copy-Constructor von <b>T</b> aufgerufen und somit werden insgesamt <b>n</b> Kopien des Objektes <b>x</b> erzeugt.
deque<T> VarName (T* pFirst, T* pLast);	Dieser Constructor erzeugt ein neues <b>deque</b> anhand eines bereits bestehenden Arrays. Die Pointer <b>pFirst</b> und <b>pLast</b> bezeichnen dabei Anfang- und Endwert eines Bereiches. Das Element, auf welches der Zeiger <b>pLast</b> verweist gehört <b>nicht</b> mehr zum kopierten Bereich.
deque<T> VarName (const deque<T>& d);	Dieser Constructor erzeugt einen neuen <b>deque</b> anhand eines bereits bestehenden <b>deque</b> -Objektes. Das neue <b>deque</b> -Objekt ist eine vollständige Kopie des übergebenen <b>deque</b> -Objektes <b>d</b> .

Tabelle 4-1– Constructoren der Containerklasse deque

#### 4.1.1 Der deque Standard-Constructor

Der **deque** Standard-Constructor, erzeugt ein neues, leeres **deque**-Objekt vom Typ **T**<sup>2</sup>.

Syntax:  
`deque<T> Variablenname;`

Das folgende Beispielprogramm zeigt den Einsatz des Standard-Constructors innerhalb eines Programms:



```
//=====
// PROGRAMM: DEQUE BSP_01
//=====

#include <deque>
using namespace std;

void main (void)
{
    deque<int> aMyIntDeque;
}
```

#### 4.1.2 Der deque Constructor mit Größenangabe

Dieser Constructor erzeugt ein neues **deque** mit **n** vordefinierten Elementen vom Typ **T**. Für jedes der **n** Elemente wird der Standard-Constructor von **T** (zwecks Initialisierung) aufgerufen.

Syntax:

<sup>2</sup> **T** ist der Platzhalter für den im **deque** verwalteten Datentyp

```
deque<T> Variablenname (size_type3 n);
```

Das folgende Beispielprogramm zeigt den Einsatz des Constructors innerhalb eines Programms:

```
//=====
// PROGRAMM: DEQUE_BSP_02
//=====

#include <deque>
using namespace std;

void main (void)
{
    deque<int> aMyIntDeque (5); // mit 5 Elementen beginnen
}
```



Diese Form des Constructors ist besonders nützlich, wenn man bereits eine ungefähre Menge an zu erwartenden Elementen abschätzen kann. Durch die Erzeugung eines **deque**-Objektes zu Beginn kann die oben angesprochene Ineffizienz durch häufiges Umkopieren im Speicher leicht vermieden werden.

#### 4.1.3 Der deque Constructor mit Größenangabe und Vorlage

Dieser Constructor erzeugt ein neues **deque**-Objekt mit **n** vordefinierten Elementen vom Typ **T**. Für jedes der **n** Elemente wird der Copy-Constructor von **T** (zwecks Initialisierung) aufgerufen und somit werden insgesamt **n** Kopien des Objektes **x** erzeugt.

Syntax:

```
deque<T> Variablenname (size_type n, T& x);
```

Das folgende Beispielprogramm zeigt den Einsatz des Constructors innerhalb eines Programms:

```
//=====
// PROGRAMM: DEQUE_BSP_03
//=====

#include <deque>
using namespace std;

void main (void)
{
    int Vorlage = 17;
    deque<int> aMyIntDeque (5, Vorlage);
}
```



Auch diese Form des Constructors ist nützlich, wenn man bereits eine ungefähre Menge an zu erwartenden Elementen abschätzen kann. Wie beim Constructor mit Größenangabe (s.o.) kann durch die Erzeugung eines großen Objekts zu Beginn die Ineffizienz durch häufiges Umkopieren vermieden werden. Zudem hat man hier die Möglichkeit, eine gezielte

<sup>3</sup> Logischer Größentyp für eine Anzahl von Elementen vom Typ **T**

Vorbelegung für alle zu erzeugenden Elemente zu treffen, die von der Vorbelegung durch den Standard-Constructor der zu speichernden Elemente **T** abweicht.

#### 4.1.4 Der deque Constructor mit Bereichsangabe

Dieser Constructor erzeugt ein neues **deque** anhand eines bereits bestehenden **deque**-Objektes. Die Pointer **pFirst** und **pLast** bezeichnen dabei Anfang- und Endwert eines Bereiches. Das Element, auf welches der Iterator **pLast** verweist gehört *nicht* mehr zum kopierten Bereich – d.h. **pLast** verweist auf das erste Element hinter dem zu kopierenden Bereich. Für jedes der im Bereich befindlichen Elemente wird einmal der Copy-Constructor von **T** mit dem korrespondierenden Element des übergebenen Bereiches aufgerufen.

Syntax:

```
deque<T> Variablenname (T* pFirst, T* pLast);
```

Das folgende Beispielprogramm zeigt, wie der Constructor innerhalb eines Programms verwendet werden kann:



```
//=====
// PROGRAMM: DEQUE_BSP_04
//=====

#include <deque>
using namespace std;

int Feld [7] = {1, 2, 3, 4, 5, 6, 7};

void main (void)
{
    int Wert = 13;
    deque<int> aTest (6, Wert);

    // 1. Die Elemente 0..2 kopieren
    deque<int> aMyIntDeque1 (aTest.begin(), aTest.begin()+3);

    // 2. das gesamte deque kopieren
    deque<int> aMyIntDeque2 (aTest.begin(), aTest.end());
}
```

Anhand der übergebenen Zeiger kann der Constructor die benötigte Anzahl an Elementen ermitteln und eine entsprechend großen Block reservieren. Da der Copy-Constructor von **T** nur einmal für jedes Element aufgerufen wird (im Unterschied zum Constructor mit Größenangabe und Vorlage sind es hier verschiedene Objekte des Typs **T** die kopiert werden), ist diese Constructor-Form ebenso effizient wie der bereits erwähnte Constructor mit Größenangabe und Vorlage.

#### 4.1.5 Der deque Copy-Constructor

Dieser Constructor erzeugt ein neues **deque** anhand eines bereits bestehenden **deque**-Objektes. Das neue **deque**-Objekt ist eine vollständige Kopie des übergebenen Objektes **d**.

Syntax:

```
deque<T> Variablenname (const deque<T>& d);
```

Das folgende Beispielprogramm zeigt, wie der Copy-Constructor verwendet werden kann:

```
//=====
// PROGRAMM: DEQUE_BSP_05
//=====

#include <deque>
using namespace std;

void main (void)
{
    deque<int> aMyIntDeque1 (3);

    // das deque kopieren
    deque<int> aMyIntDeque2 (aMyIntDeque1);
}
```



Da auch bei diesem Constructor die benötigte Anzahl an Elementen ermittelt und ein entsprechend großer Block reserviert werden kann, ist er ebenfalls sehr viel effizienter als eine Kopie Datenelement für Datenelement.

## 4.2 Die deque Methoden

Die folgende Tabelle faßt die Methoden der **deque** Containerklasse zusammen. In den nachstehenden Abschnitten werden die Methoden einzeln behandelt und mit Beispielen erläutert.

<b>Methoden der Containerklasse DEQUE</b>	
<i>Methoden</i>	<i>Bedeutung</i>
back	Gibt eine Referenz auf das letzte im <b>deque</b> enthaltene Datenelement zurück (also DatentypT&)
begin	Rückgabe eines <b>iterator</b> , der auf das erste im <b>deque</b> enthaltene Element zeigt
empty	Gibt den Wert <b>true</b> zurück, wenn das <b>deque</b> keine Datenelemente enthält.
end	Rückgabe eines <b>iterator</b> , der hinter das letzte im <b>deque</b> enthaltene Element zeigt
erase	Löscht ein oder mehrere Datenelemente an der angegebenen Position aus dem <b>deque</b>
front	Gibt eine Referenz auf das erste im <b>deque</b> enthaltene Datenelement zurück (also DatentypT&)
insert	Fügt ein oder mehrere Datenelemente an der angegebenen Position in das <b>deque</b> ein
max_size	Gibt die maximale Anzahl an Datenelementen zurück, die das <b>deque</b> enthalten kann
pop_back	Löscht das letzte Datenelement aus dem <b>deque</b>
pop_front	Löscht das erste Datenelement aus dem <b>deque</b>
push_back	Fügt ein Datenelement am Ende des <b>deque</b> an
push_front	Fügt ein Datenelement am Anfang des <b>deque</b> an

<b>Methoden der Containerklasse DEQUE</b>	
<i>Methode</i>	<i>Bedeutung</i>
rbegin	Rückgabe eines <b>reverse_iterator</b> , der auf das letzte im <b>deque</b> enthaltene Element zeigt
rend	Rückgabe eines <b>reverse_iterator</b> , der vor das erste im <b>deque</b> enthaltene Element zeigt
size	Gibt die Anzahl von Datensätzen zurück, die aktuell im <b>deque</b> enthalten sind
swap	Tauscht den Inhalt zweier <b>deque</b> -Objekte aus

Tabelle 4-1– Methoden der Containerklasse deque

#### 4.2.1 Die deque Methode back

Die Methode **back()** gibt eine Referenz oder einen **const**-Referenz auf das letzte im **deque** gespeicherte Datenelement zurück. Im Gegensatz zu **end()** handelt es sich um eine Objektreferenz vom Typ **T** und nicht einen **iterator** (welcher zudem hinter das letzte Element zeigen würde).

Der Aufruf von **back()** selbst verändert den Inhalt des **deque** nicht.

Syntax:

```
T& back ();
const T& back () const;
```

Das nachfolgende Beispielprogramm zeigt den Unterschied zwischen **back()** und **end()** anhand einer Ausgabeanweisung:



```
//=====
// PROGRAMM: DEQUE_BSP_06
//=====

#include <iomanip.h>
#include <iostream.h>
#include <deque>
using namespace std;

void main (void)
{
    deque<double> aDeq;
    deque<double>::const_iterator aI;

    aDeq.push_back (7.0);

    aI = aDeq.end()-1;
    cout << *aI << endl;          // Ausgabe Dereferenzierung
    von aI
    cout << aDeq.back() << endl; // Ausgabe über Referenz
}
```

#### 4.2.2 Die deque Methode begin

Rückgabe eines **iterator** oder **const\_iterator**, der auf das erste im **deque** enthaltene Element zeigt.

Syntax:

```
iterator begin ();
const_iterator begin () const;
```

Die wohl häufigste Anwendung der Methode **begin()** liegt in der Verarbeitung von Daten durch Schleifen.

Der Aufruf von **begin()** verändert den Inhalt des **deque** nicht.

```
//=====
// PROGRAMM: DEQUE_BSP_07
//=====

#include <iomanip.h>
#include <iostream.h>
#include <deque>
using namespace std;

void main (void)
{
    deque<double> aDeq;
    deque<double>::const_iterator aI;

    aDeq.push_back (7.0);
    aDeq.push_back (1.2);
    aDeq.push_back (3.14);

    for (aI=aDeq.begin(); aI<aDeq.end(); aI++)
    {
        cout << "Wert: " << *aI << endl;
    }
}
```



### 4.2.3 Die deque Methode empty

Die Methode **empty()** prüft, ob das **deque** Datenelemente enthält oder nicht. Sind Datenelemente enthalten (dies entspricht einem Rückgabewert von **size()** größer als Null), so wird der Wert **false** zurückgegeben, sind keine Datenelemente vorhanden ist der Rückgabewert **true**.

Der Aufruf von **empty()** verändert den Inhalt des **deque** nicht.

Syntax:

```
bool empty () const;
```

```
//=====
// PROGRAMM: DEQUE_BSP_08
//=====

#include <iomanip.h>
#include <iostream.h>
#include <deque>
using namespace std;

void main (void)
{
    deque<double> aDeq;

    if (aDeq.empty())
    {
        cout << "Deque ist leer" << endl;
    }
    else
    {
        cout << "Deque enthält Daten" << endl;
    }
}
```



```

}

aDeq.push_back (7.0);
aDeq.push_back (3.14);

if (aDeq.empty())
{
    cout << "Deque ist leer" << endl;
}
else
{
    cout << "Deque enthält Daten" << endl;
}
}

```

#### 4.2.4 Die deque Methode end

Rückgabe eines **iterator** oder **const\_iterator**, der *hinter* das letzte im **deque** enthaltene Element zeigt.

Syntax:

```

iterator end ();
const_iterator end () const;

```

Die wohl häufigste Anwendung der Methode **end()** liegt in der Verarbeitung von Daten durch Schleifen.

Es ist zu beachten, daß **end()** hinter das letzte gültige Datenelement zeigt, alle Schleifen müssen also auf „ungleich“ (Operator !=) oder „echt kleiner als“ prüfen (Operator <) und nicht auf „kleiner oder gleich“ (Operator <=). Zeigt ein **iterator** auf **end()** führt jeder dereferenzierende Zugriff zu einer Exception vom Typ **out\_of\_range**. Wird die Exception nicht abgefangen (wie zweiten Beispiel), bricht das Programm unkontrolliert ab (Absturz).

Der Aufruf von **end()** verändert den Inhalt des **deque** nicht.



```

//=====
// PROGRAMM: DEQUE_BSP_09
//=====

#include <iomanip.h>
#include <iostream.h>
#include <deque>
using namespace std;

void main (void)
{
    deque<double> aDeq;
    deque<double>::const_iterator aI;

    aDeq.push_back (7.0);
    aDeq.push_back (1.2);
    aDeq.push_back (3.14);

    for (aI=aDeq.begin(); aI<aDeq.end(); aI++)
    {
        cout << "Wert: " << *aI << endl;
    }
}

```

Ein Beispiel, welches die Auslösung einer Exception aufgrund eines illegalen Zugriffs auf **end()** zeigt, ist im Abschnitt zur Methode **end()** bei der Containerklasse **vector** aufgeführt.

#### 4.2.5 Die deque Methode erase

Löscht ein oder mehrere Datenelemente an der angegebenen Position aus dem **deque**.

Syntax:

```
void erase(iterator toDel);
void erase(iterator First, iterator Last);
```

Die Methode **erase()** kann verwendet werden, um einzelne Datensätze oder ganze Bereiche zu löschen.

*Durch das Löschen werden alle Iteratoren – sowie Verweise und Referenzen auf Datensätze im gelöschten Bereich – ungültig, da es sich um Pointer handelt und das Array gegebenenfalls im Speicher verschoben wurde.*

*Es ist daher darauf zu achten, daß alle Iteratoren, Verweise und Referenzen nach einen Befehl, der die Größe des Deque verändert, neu ermittelt werden müssen.*

Die erste Form der Methode sorgt dafür, daß ein einzelner Datensatz aus dem **deque** gelöscht und der Destructor der Klasse **T** für diesen Datensatz aufgerufen wird.

Bei der zweiten Syntaxform wird ein zu löschender Bereich angegeben, für jedes zu löschende Element wird der Destructor der Klasse **T** aufgerufen. Die Angabe erfolgt durch zwei Iteratoren, die auf das erste und das letzte zu löschende Element zeigen.

Es ist besonders bei den Bereichsangaben zu beachten, daß nicht versehentlich eine **out\_of\_range** Exception erzeugt wird.

Ein Beispiel, welches die Auslösung einer Exception aufgrund eines illegalen Zugriffs durch **erase()** zeigt, ist im Abschnitt zur Methode **erase()** bei der Containerklasse **vector** aufgeführt.

#### 4.2.6 Die deque Methode front

Die Methode **front()** gibt eine Referenz oder einen **const**-Referenz auf das erste im **deque** gespeicherte Datenelement zurück. Im Gegensatz zu **begin()** handelt es sich um eine Objektreferenz vom Typ **T** und nicht einen **iterator**.

Der Aufruf von **front()** selbst verändert den Inhalt des **deque** nicht.

Syntax:

```
T& front ();
const T& front () const;
```

Das nachfolgende Beispielprogramm zeigt den Unterschied zwischen **front()** und **begin()** anhand einer Ausgabeanweisung:



```

//=====
// PROGRAMM: DEQUE_BSP_10
//=====

#include <iomanip.h>
#include <iostream.h>
#include <deque>
using namespace std;

void main (void)
{
    deque<double> aDeq;
    deque<double>::const_iterator aI;

    aDeq.push_back (7.0);

    aI = aDeq.begin();
    cout << *aI << endl;           // Ausgabe Dereferenzierung
v. aI
    cout << aDeq.front() << endl; // Ausgabe über Referenz
}

```

#### 4.2.7 Die deque Methode insert

Fügt ein oder mehrere Datenelemente vor der angegebenen Position in den **deque** ein.

```

Syntax:
    iterator insert (iterator pos, T& value);
    void          insert (iterator pos, size_type Anz,
                        T& value);
    void          insert (iterator pos, const T *First,
                        const T *Last);

```

Die Methode **insert()** kann verwendet werden um einzelne Datensätze oder ganze Bereiche in einen **deque** einzufügen.

Da ein **deque** wie ein Array aufgebaut ist, müssen beim Einfügen von Datensätzen zumindest die nachfolgenden Datenelemente im **deque** umkopiert werden, was die Methode **insert()** für **deque**-Objekte (insbesondere große Objekte) relativ ineffizient macht. Gegebenenfalls muß sogar der gesamte **deque** umkopiert werden (Reallokation), da keine Pufferbereiche mehr vorhanden sind.

*Wird eine Reallokation notwendig, so ist zu beachten, daß alle Iteratoren, Verweise und Referenzen ungültig werden, da es sich um Pointer handelt und das Array gegebenenfalls im Speicher verschoben wurde. Es ist ferner darauf zu achten, daß alle Iteratoren, Verweise und Referenzen nach einem Befehl, der die Größe des Deque verändert, neu ermittelt werden müssen.<sup>4</sup>*

<sup>4</sup> Eine Graphik zur Illustration des Problems ist bei der Methode **insert()** der Containerklasse **vector** zu finden.

Die erste Form der Methode sorgt dafür, daß ein einzelner Datensatz (**value**) vor der angegebenen Position (**pos**) in den **deque** eingefügt wird. Dazu wird der Copy-Constructor der Klasse **T** für den Parameter **value** aufgerufen und die so entstandene Kopie in den **deque** eingefügt.

Die zweite Form der **insert()** Methode fügt mehrere (**Anz**) Kopien des Datensatzes **value** an der Position **pos** ein. Der Copy-Constructor der Klasse **T** wird **Anz**-mal für den Parameter **value** aufgerufen und die so entstandenen Kopien in den **deque** eingefügt.

Mit der dritten Methode können Bereiche aus einem anderen Array in den **deque** kopiert werden (hier anhand eines Ausschnittes aus dem Array **fArray** dargestellt).

Der Bereich wird durch Anfangs- und Endadresse festgelegt. Es sollte darauf geachtet werden, daß die Endadresse nicht vor der Anfangsadresse liegen darf und daß beide Adressen zum gleichen Array bzw. gleichen **deque** gehören.

```
//=====
// PROGRAMM: DEQUE_BSP_11
//=====

#include <iomanip.h>
#include <iostream.h>
#include <deque>
using namespace std;

double fArray [6] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0};

void main (void)
{
    deque<double> aDeq;
    deque<double> aD2;
    deque<double>::iterator aI;

    aDeq.push_back (1.1);
    aDeq.push_back (2.2);
    aDeq.push_back (3.3);

    aD2.push_back (11.1);
    aD2.push_back (22.2);
    aD2.push_back (33.3);

    //-----
    // 1. Form
    //-----
    aI = aDeq.begin()+1;      // Auf 2. Element zeigen
    aDeq.insert (aI, 4.4);   // vor 2. Element einfügen

    //-----
    // 2. Form
    //-----
    aI = aDeq.begin()+1;     // wegen möglicher Reallokation
```



```

aDeq.insert (aI, 6, 5.5); // 6 Kopien einfügen

//-----
-----
// 3. Form
//-----
-----
aI = aDeq.begin()+1; // wegen möglicher Reallokation
aDeq.insert(aI, *(fArray+1), *(fArray+4)); // Elemente aus
Array

// 3. Form wie oben, aber mit Indeschreibweise
aI = aDeq.begin()+1; // wegen möglicher Reallokation
aDeq.insert(aI, fArray[1], fArray[4]); // Elemente aus
Array

// 3. Form aber aus einem anderen Deque
aI = aDeq.begin()+4; // wegen möglicher Reallokation
aDeq.insert(aI, aD2[0], aD2[5]); // Elemente aus Deque

for (aI=aDeq.begin(); aI<aDeq.end(); aI++)
{
    cout << *aI << endl;
}
}

```

#### 4.2.8 Die deque Methode max\_size

Gibt die maximale Anzahl an Datenelementen zurück, die der **deque** insgesamt enthalten kann.

Syntax:

```
size_type max_size () const;
```

Der Wert ist abhängig von der Implementation und vom verwendeten Betriebssystem. Bei modernen Compilern und entsprechender Einstellung kann man davon ausgehen, das 32-Bit Adressen verwendet werden, also theoretisch 4 Gigabytes angesprochen werden können.

Beschränkt wird die Anzahl der Datenelemente durch die Tatsache, daß der zu verwendende Speicher (wie bei jedem Array) als ein ungeteilter Block vorhanden sein muß. D.h. die Größe des Arrays wird durch alle im RAM-Speicher befindlichen Daten begrenzt, die Größe der zu verwaltenden Datenelemente, sowie die Segmentierung (Zersplitterung) des Speichers.

Die Methode **max\_size()** dient dazu abschätzen zu können, ob ein aufzubauender **deque** überhaupt in den Speicher paßt.

Der Aufruf der Methode verändert den Inhalt des **deque** nicht.



```

//=====
// PROGRAMM: DEQUE_BSP_12
//=====

#include <iostream.h>
#include <deque>
using namespace std;

void main (void)
{
    deque<double> aDeq;
}

```

```

cout << "Deque kann " << aDeq.max_size ()
      << " Datenelemente aufnehmen";
}

```

#### 4.2.9 Die deque Methode pop\_back

Löscht das letzte Datenelement aus dem **deque**.

```

Syntax:
void pop_back ();

```

Die Methode **pop\_back()** wird verwendet um einen einzelnen Datensatz vom Ende des **deque** zu entfernen. Sollen mehrere Datensätze am Ende des **deque** gelöscht werden, ist die Methode **erase()** günstiger, als ein wiederholter Aufruf von **pop\_back()**.

*Durch das Löschen werden daß alle Iteratoren – sowie Verweise und Referenzen auf Datensätze im gelöschten Bereich – ungültig, da es sich um Pointer handelt und das Array gegebenenfalls im Speicher verschoben wurde.  
Es ist daher darauf zu achten, daß alle Iteratoren, Verweise und Referenzen nach einen Befehl, der die Größe des Deque verändert, neu ermittelt werden müssen.*

Zum Löschen wird der Destructor der Klasse **T** für das letzte Element im **deque** aufgerufen.

```

//=====
// PROGRAMM: DEQUE_BSP_13
//=====

#include <iomanip.h>
#include <iostream.h>
#include <deque>
using namespace std;

void main (void)
{
    deque<double> aDeq;
    deque<double>::iterator aI;

    aDeq.push_back (1.1);
    aDeq.push_back (2.2);
    aDeq.push_back (3.3);

    aDeq.pop_back ();

    for (aI=aDeq.begin(); aI<aDeq.end(); aI++)
    {
        cout << *aI << endl;
    }
}

```



#### 4.2.10 Die deque Methode pop\_front

Löscht das erste Datenelement aus dem **deque**.

Syntax:  

```
void pop_front ();
```

Die Methode **pop\_front()** wird verwendet um einen einzelnen Datensatz vom Beginn des **deque** zu entfernen. Sollen mehrere Datensätze am Anfang des **deque** gelöscht werden, ist die Methode **erase()** günstiger, als ein wiederholter Aufruf von **pop\_front()**.

*Durch das Löschen werden daß alle Iteratoren – sowie Verweise und Referenzen auf Datensätze im gelöschten Bereich – ungültig, da es sich um Pointer handelt und das Array gegebenenfalls im Speicher verschoben wurde.  
 Es ist daher darauf zu achten, daß alle Iteratoren, Verweise und Referenzen nach einen Befehl, der die Größe des Deque verändert, neu ermittelt werden müssen.*

Zum Löschen wird der Destructor der Klasse **T** für das erste Element im **deque** aufgerufen.



```
//=====
// PROGRAMM: DEQUE_BSP_14
//=====

#include <iomanip.h>
#include <iostream.h>
#include <deque>
using namespace std;

void main (void)
{
    deque<double> aDeq;
    deque<double>::iterator aI;

    aDeq.push_back (1.1);
    aDeq.push_back (2.2);
    aDeq.push_back (3.3);

    aDeq.pop_front ();

    for (aI=aDeq.begin(); aI<aDeq.end(); aI++)
    {
        cout << *aI << endl;
    }
}
```

#### 4.2.11 Die deque Methode push\_back

Fügt ein Datenelement am Ende des **deque** an.

Syntax:

```
void push_back (const T& value);
```

Die Methode **push\_back()** wird verwendet um einen einzelnen Datensatz an das Ende des **deque** einzufügen. Sollen mehrere Datensätze am Ende des **deque** angefügt werden, ist die Methode **insert()** günstiger, als ein wiederholter Aufruf von **push\_back()**.

Da ein **deque** wie ein Array aufgebaut ist, muß beim Anfügen eines Datensatzes eventuell eine Reallokation (s.o.) durchgeführt werden, da keine Pufferbereiche mehr vorhanden sind.

*Wird eine Reallokation notwendig, so ist zu beachten, daß alle Iteratoren, Verweise und Referenzen ungültig werden, da es sich um Pointer handelt und das Array gegebenenfalls im Speicher verschoben wurde. Es ist ferner darauf zu achten, daß alle Iteratoren, Verweise und Referenzen nach einen Befehl, der die Größe des Deque verändert, neu ermittelt werden müssen.*

Zum Anfügen wird der Copy-Constructor der Klasse **T** für den Parameter **value** aufgerufen und die so entstandene Kopie an den **deque** als letzter Datensatz angehängt.

```
//=====
// PROGRAMM: DEQUE_BSP_15
//=====

#include <iomanip.h>
#include <iostream.h>
#include <deque>
using namespace std;

void main (void)
{
    deque<double> aDeq;
    deque<double>::iterator aI;

    aDeq.push_back (1.1);
    aDeq.push_back (2.2);

    for (aI=aDeq.begin(); aI<aDeq.end(); aI++)
    {
        cout << *aI << endl;
    }
}
```



#### 4.2.12 Die deque Methode push\_front

Fügt ein Datenelement am Anfang des **deque** an.

Syntax:

```
void push_front (const T& value);
```

Die Methode **push\_front()** wird verwendet um einen einzelnen Datensatz an den Beginn des **deque** einzufügen. Sollen mehrere Datensätze am Anfang

des **deque** angefügt werden, ist die Methode **insert()** günstiger, als ein wiederholter Aufruf von **push\_front()**.

Da ein **deque** wie ein Array aufgebaut ist, muß beim Anfügen eines Datensatzes eventuell eine Reallokation (s.o.) durchgeführt werden, da keine Pufferbereiche mehr vorhanden sind.

*Wird eine Reallokation notwendig, so ist zu beachten, daß alle Iteratoren, Verweise und Referenzen ungültig werden, da es sich um Pointer handelt und das Array gegebenenfalls im Speicher verschoben wurde. Es ist ferner darauf zu achten, daß alle Iteratoren, Verweise und Referenzen nach einem Befehl, der die Größe des Deque verändert, neu ermittelt werden müssen.*

Zum Anfügen wird der Copy-Constructor der Klasse **T** für den Parameter **value** aufgerufen und die so entstandene Kopie an den **deque** als erster Datensatz angefügt.



```
//=====
// PROGRAMM: DEQUE_BSP_16
//=====

#include <iomanip.h>
#include <iostream.h>
#include <deque>
using namespace std;

void main (void)
{
    deque<double> aDeq;
    deque<double>::iterator aI;

    aDeq.push_front (1.1);
    aDeq.push_front (2.2);

    for (aI=aDeq.begin(); aI<aDeq.end(); aI++)
    {
        cout << *aI << endl;
    }
}
```

#### 4.2.13 Die deque Methode rbegin

Rückgabe eines **reverse\_iterator** oder **const\_reverse\_iterator**, der auf das letzte im **deque** enthaltene Element zeigt.

Syntax:

```
reverse_iterator rbegin ();
const_reverse_iterator rbegin () const;
```

Die wohl häufigste Anwendung der Methode **rbegin()** liegt in der Verarbeitung von Daten durch rückwärts gerichtete Schleifen. Der Aufruf von **rbegin()** verändert den Inhalt des **deque** nicht.

```
//=====
// PROGRAMM: DEQUE_BSP_17
//=====

#include <iomanip.h>
#include <iostream.h>
#include <deque>
using namespace std;

void main (void)
{
    deque<double> aDeq;
    deque<double>::reverse_iterator aI;

    aDeq.push_back (7.0);
    aDeq.push_back (1.2);
    aDeq.push_back (3.14);

    for (aI=aDeq.rbegin(); aI!=aDeq.rend(); aI++)
    {
        cout << "Wert: " << *aI << endl;
    }
}
```



#### 4.2.14 Die deque Methode rend

Rückgabe eines **reverse\_iterator** oder **const\_reverse\_iterator**, der vor das erste im **deque** enthaltene Element zeigt.

Syntax:

```
reverse_iterator rend ();
const_reverse_iterator rend () const;
```

Die wohl häufigste Anwendung der Methode **rend()** liegt in der Verarbeitung von Daten durch rückwärts gerichtete Schleifen.

Es ist zu beachten, daß **rend()** vor das erste gültige Datenelement zeigt, alle Schleifen müssen also auf „ungleich“ prüfen (Operator !=). Zeigt ein **iterator** auf **rend()** führt jeder dereferenzierende Zugriff zu einer Exception vom Typ **out\_of\_range**. Wird die Exception nicht wie zweiten Beispiel abgefangen bricht das Programm unkontrolliert ab (Absturz).

Der Aufruf von **rend()** verändert den Inhalt des **deque** nicht.

```
//=====
// PROGRAMM: DEQUE_BSP_18
//=====

#include <iomanip.h>
#include <iostream.h>
#include <deque>
using namespace std;

void main (void)
{
    deque<double> aDeq;
    deque<double>::reverse_iterator aI;

    aDeq.push_back (7.0);
    aDeq.push_back (1.2);
    aDeq.push_back (3.14);
}
```



```

for (aI=aDeq.rbegin(); aI!=aDeq.rend(); aI++)
{
    cout << "Wert: " << *aI << endl;
}

```

Ein Beispiel, welches die Auslösung einer Exception aufgrund eines illegalen Zugriffs auf **rend()** zeigt, ist im Abschnitt über die Methode **rend()** der Containerklasse **vector** zu finden.

#### 4.2.15 Die deque Methode size

Gibt die Anzahl von Datensätzen zurück, die aktuell im **deque** enthalten sind.

Syntax:

```
size_type size () const;
```

Der Aufruf von **size()** verändert den Inhalt des **deque** nicht.



```

//=====
// PROGRAMM: DEQUE_BSP_19
//=====

#include <iomanip.h>
#include <iostream.h>
#include <deque>
using namespace std;

void main (void)
{
    deque<double> aDeq;

    aDeq.push_back (1.2);
    aDeq.push_back (3.14);

    cout << "Anzahl aktuell: " << aDeq.size() << endl;
}

```

#### 4.2.16 Die deque Methode swap

Tauscht die Inhalte zweier **deque**-Objekte aus. Die **deque**-Objekte müssen von gleichem Typ sein, also gleiche Datenelementtypen enthalten.

Syntax:

```
void swap (deque<T>& v);
```

*Durch den Aufruf von **swap()** wird wahrscheinlich für beide Deque-Objekte eine Reallokation notwendig. Es ist zu beachten, daß somit alle Iteratoren, Verweise und Referenzen ungültig werden, da es sich um Pointer handelt und die Arrays im Speicher verschoben wurden.*

*Alle Iteratoren, Verweise und Referenzen müssen nach einem Befehl, der die Größe des Deque verändert, neu ermittelt werden.*

```

//=====
// PROGRAMM: DEQUE_BSP_20
//=====

#include <iomanip.h>
#include <iostream.h>
#include <deque>
using namespace std;

void main (void)
{
    deque<double> aDeq;
    deque<double> aD2;
    deque<double>::iterator aI;

    aDeq.push_back (1.2);
    aDeq.push_back (3.14);

    aD2.push_back (222.22);
    aD2.push_back (333.33);
    aD2.push_back (111.11);
    aD2.push_back (444.44);

    for (aI=aDeq.begin(); aI<aDeq.end(); aI++)
    {
        cout << "aDeq Wert: " << *aI << endl;
    }

    for (aI=aD2.begin(); aI<aD2.end(); aI++)
    {
        cout << "aD2 Wert: " << *aI << endl;
    }

    aDeq.swap (aD2);

    for (aI=aDeq.begin(); aI<aDeq.end(); aI++)
    {
        cout << "aDeq Wert: " << *aI << endl;
    }

    for (aI=aD2.begin(); aI<aD2.end(); aI++)
    {
        cout << "aD2 Wert: " << *aI << endl;
    }
}

```



### 4.3 Die deque Operatoren

Mit den folgenden Operatoren kann auf eine Containerklasse vom Typ **deque** zugegriffen werden:

<b>Operatoren der Containerklasse DEQUE</b>	
<i>Operator</i>	<i>Bedeutung</i>
[ ]	Indezzugriff auf den Inhalt des <b>deque</b>
=	Zuweisung zwischen zwei <b>deque</b> -Objekten gleichen Typs
==	Vergleich zwischen zwei <b>deque</b> -Objekten gleichen Typs
!=	Vergleich zwischen zwei <b>deque</b> -Objekten gleichen Typs
<	Lexikographischer Vergleich zwischen zwei <b>deque</b> -

<b>Operatoren der Containerklasse DEQUE</b>	
<i>Operator</i>	<i>Bedeutung</i>
	Objekten gleichen Typs
>	Lexikographischer Vergleich zwischen zwei <b>deque</b> -Objekten gleichen Typs
>=	Lexikographischer Vergleich zwischen zwei <b>deque</b> -Objekten gleichen Typs
<=	Lexikographischer Vergleich zwischen zwei <b>deque</b> -Objekten gleichen Typs

Tabelle 4-1– Operatoren der Containerklasse deque

### 4.3.1 Der deque Operator [ ]

Mittels der eckigen Klammern kann direkt auf ein im **deque** gespeichertes Element zugegriffen werden. Die eckigen Klammern sind im Template der Containerklasse überladen (Operator-Overloading).

Syntax:

```
T& operator[] (int index);
```

Wird ein ungültiger Index angegeben, so wird eine **out\_of\_range** Exception ausgelöst:



```
//=====
// PROGRAMM: DEQUE_BSP_21
//=====

#include <iomanip.h>
#include <iostream.h>
#include <deque>
using namespace std;

void main (void)
{
    deque<double> aDeq;

    aDeq.push_back (7.0);
    aDeq.push_back (1.2);
    aDeq.push_back (3.14);

    try
    {
        cout << "Wert: " << aDeq[1] << endl;
        cout << "Wert: " << aDeq[50] << endl; // Exception
    }
    catch (out_of_range& aError)
    {
        cout << "Zugriff auf illegales Element" << endl;
    }
}
```

### 4.3.2 Der deque Operator =

Der Operator ersetzt den Inhalt eines **deque**-Objektes durch den Inhalt des zugewiesenen **deque**-Objektes gleichen Typs.

Syntax:

```
deque<T>& operator= (const deque<T>& v);
```

*Durch die Zuweisung wird für den linken Operanden (d.h. den Deque, dem Werte zugewiesen werden) eine Reallokation notwendig. Es ist zu beachten, daß somit alle Iteratoren, Verweise und Referenzen auf diesem Deque ungültig werden.*

*Alle Iteratoren, Verweise und Referenzen müssen nach einem Befehl, der die Größe des Deque verändert, neu ermittelt werden.*

```
//=====
// PROGRAMM: DEQUE_BSP_22
//=====

#include <iomanip.h>
#include <iostream.h>
#include <deque>
using namespace std;

void main (void)
{
    deque<double> aDeq;
    deque<double> aD2;
    deque<double>::iterator aI;

    aDeq.push_back (7.0);
    aDeq.push_back (1.2);
    aDeq.push_back (3.14);

    aD2 = aDeq;

    for (aI=aDeq.begin(); aI<aDeq.end(); aI++)
    {
        cout << "aDeq Wert: " << *aI << endl;
    }

    for (aI=aD2.begin(); aI<aD2.end(); aI++)
    {
        cout << "aD2 Wert: " << *aI << endl;
    }
}
```



### 4.3.3 Der deque Operator ==

Der Operator prüft, ob die Inhalte des linken und rechten Operanden (**deque**-Objekte gleichen Typs) übereinstimmen. Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn beide **deque**-Objekte die gleichen Elemente in der gleichen Reihenfolge enthalten.

Syntax:

```
bool operator== (const deque<T>& v) const;
```



```
//=====
// PROGRAMM: DEQUE_BSP_23
//=====

#include <iomanip.h>
#include <iostream.h>
#include <deque>
using namespace std;

void main (void)
{
    deque<double> aDeq;
    deque<double> aD2;

    aDeq.push_back (3.14);

    if (aDeq == aD2)
    {
        cout << "Die Deque-Objekte sind gleich" << endl;
    }
    else
    {
        cout << "Die Deque-Objekte sind nicht gleich" << endl;
    }

    aD2 = aDeq;

    if (aDeq == aD2)
    {
        cout << "Die Deque-Objekte sind gleich" << endl;
    }
    else
    {
        cout << "Die Deque-Objekte sind nicht gleich" << endl;
    }
}
}
```

#### 4.3.4 Der deque Operator !=

Der Operator prüft, ob die Inhalte des linken und rechten Operanden (**deque**-Objekte gleichen Typs) verschieden sind. Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn beide **deque**-Objekte unterschiedliche Elemente oder gleiche Elemente in unterschiedlicher Reihenfolge enthalten.

Syntax:

```
bool operator!= (const deque<T>& v) const;
```



```
//=====
// PROGRAMM: DEQUE_BSP_24
//=====

#include <iomanip.h>
#include <iostream.h>
#include <deque>
using namespace std;
```

```

void main (void)
{
    deque<double> aDeq;
    deque<double> aD2;

    aDeq.push_back (3.14);

    if (aDeq != aD2)
    {
        cout << "Die Deque-Objekte sind nicht gleich" << endl;
    }
    else
    {
        cout << "Die Deque-Objekte sind gleich" << endl;
    }

    aD2 = aDeq;

    if (aDeq != aD2)
    {
        cout << "Die Deque-Objekte sind nicht gleich" << endl;
    }
    else
    {
        cout << "Die Deque-Objekte sind gleich" << endl;
    }
}

```

#### 4.3.5 Der deque Operator <

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch kleiner ist, als der des rechten Operanden (**deque**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „<“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch kleiner ist als der rechte Operand.

Syntax:

```
bool operator< (const deque<T>& v) const;
```

```

//=====
// PROGRAMM: DEQUE_BSP_25
//=====

#include <iomanip.h>
#include <iostream.h>
#include <deque>
using namespace std;

void main (void)
{
    deque<double> aDeq;
    deque<double> aD2;

    aDeq.push_back (3.14);
    aD2.push_back (3.15);

    if (aDeq < aD2)
    {
        cout << "aDeq ist kleiner als aD2" << endl;
    }
}

```



```

else
{
    cout << " aDeq ist nicht kleiner als aD2" << endl;
}
}

```

#### 4.3.6 Der deque Operator >

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch größer ist, als der des rechten Operanden (**deque**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „>“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch größer ist als der rechte Operand.

Syntax:

```
bool operator> (const deque<T>& v) const;
```



```

//=====
// PROGRAMM: DEQUE_BSP_26
//=====

#include <iomanip.h>
#include <iostream.h>
#include <deque>
using namespace std;

void main (void)
{
    deque<double> aDeq;
    deque<double> aD2;

    aDeq.push_back (3.14);
    aD2.push_back (3.15);

    if (aDeq > aD2)
    {
        cout << "aDeq ist größer als aD2" << endl;
    }
    else
    {
        cout << " aDeq ist nicht größer als aD2" << endl;
    }
}

```

#### 4.3.7 Der deque Operator <=

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch kleiner oder gleich mit dem rechten Operanden ist (**deque**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „<=“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch kleiner oder gleich mit dem rechten Operanden ist.

Syntax:

```
bool operator<= (const deque<T>& v) const;
```

```

//=====
// PROGRAMM: DEQUE_BSP_27
//=====

#include <iomanip.h>
#include <iostream.h>
#include <deque>
using namespace std;

void main (void)
{
    deque<double> aDeq;
    deque<double> aD2;

    aDeq.push_back (3.14);
    aD2.push_back (3.15);

    if (aDeq <= aD2)
    {
        cout << "aDeq ist kleiner-gleich aD2" << endl;
    }
    else
    {
        cout << " aDeq ist nicht kleiner-gleich als aD2" <<
endl;
    }
}

```



#### 4.3.8 Der deque Operator >=

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch größer oder gleich mit dem rechten Operanden ist (**deque**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „>=“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch größer oder gleich mit dem rechten Operanden ist.

Syntax:

```
bool operator>= (const deque<T>& v) const;
```

```

//=====
// PROGRAMM: DEQUE_BSP_28
//=====

#include <iomanip.h>
#include <iostream.h>
#include <deque>
using namespace std;

void main (void)
{
    deque<double> aDeq;
    deque<double> aD2;

    aDeq.push_back (3.14);
    aD2.push_back (3.15);

    if (aDeq >= aD2)
    {
        cout << "aDeq ist größer-gleich aD2" << endl;
    }
}

```



```
    }  
    else  
    {  
        cout << " aDeq ist nicht größer-gleich als aD2" << endl;  
    }  
}
```

## 5. Die Containerklasse list

Die Containerklasse **list** realisiert eine verkettete Liste, optimiert für schnelle Einfügen von Datensätzen an beliebiger Stelle, auf Kosten des wahlfreien Zugriffs.

Ein **list**-Objekt hat eine komplett anderen Aufbau als ein **vector**- oder **deque**-Objekt.

Die dazu notwendige Speicherverwaltung wird automatisch vorgenommen und ist so effizient wie möglich realisiert. Für **list**-Objekte wird kein geschlossener Speicherbereich benötigt, da eine Vorwärts-/Rückwärtsverkettung verwendet wird. Um ein **list**-Objekt zu erweitern ist kein Umkopieren der gesamten Containerklasse notwendig, da lediglich die Verkettung geändert werden muß.

Neue Datenelemente können somit an beliebiger Stelle eines Containerobjektes eingefügt werden. Somit kann schnell eingefügt werden, Aufwand für Reallokationen fällt nicht an.

Der **operator[]** (Zugriff über Indexklammer) ist für **list** nicht definiert.

### 5.1 Die list Constructoren

Die folgende Tabelle faßt die Constructoren der Containerklasse **list** zusammen. In den nachstehenden Abschnitten werden die Constructoren einzeln behandelt und mit Beispielen erläutert.

<b>Constructoren der Containerklasse LIST</b>	
<i>Constructor</i>	<i>Bedeutung</i>
<code>list&lt;T&gt; VarName;</code>	Standard-Constructor, erzeugt ein neues, leeres <b>list</b> -Objekt
<code>list&lt;T&gt; VarName (Size n);</code>	Dieser Constructor erzeugt ein neues <b>list</b> mit <b>n</b> vordefinierten Elementen vom Typ <b>T</b> . Für jedes der <b>n</b> Elemente wird der Standard-Constructor von <b>T</b> aufgerufen.
<code>list&lt;T&gt; VarName (Size n, T x);</code>	Dieser Constructor erzeugt ein neues <b>list</b> mit <b>n</b> vordefinierten Elementen vom Typ <b>T</b> . Für jedes der <b>n</b> Elemente wird der Copy-Constructor von <b>T</b> aufgerufen und somit werden insgesamt <b>n</b> Kopien des Objektes <b>x</b> erzeugt.
<code>list&lt;T&gt; VarName (T* pFirst, T* pLast);</code>	Dieser Constructor erzeugt ein neues <b>list</b> anhand eines bereits bestehenden Arrays. Die Pointer <b>pFirst</b> und <b>pLast</b> bezeichnen dabei Anfang- und Endwert eines Bereiches. Das Element, auf welches der Zeiger <b>pLast</b> verweist gehört <b>nicht</b> mehr zum kopierten Bereich.
<code>list&lt;T&gt; VarName (const list&lt;T&gt;&amp; L);</code>	Dieser Constructor erzeugt einen neuen <b>list</b> anhand eines bereits bestehenden <b>list</b> -Objektes. Das neue

<b>Constructoren der Containerklasse LIST</b>	
<i>Constructor</i>	<i>Bedeutung</i>
	<b>list</b> -Objekt ist eine vollständige Kopie des übergebenen <b>list</b> -Objektes <b>L</b> .

Tabelle 5-1– Constructoren der Containerklasse list

### 5.1.1 Der list Standard-Constructor

Der **list** Standard-Constructor, erzeugt ein neues, leeres **list**-Objekt vom Typ **T**<sup>1</sup>.

Syntax:

```
list<T> Variablenname;
```

Das folgende Beispielprogramm zeigt den Einsatz des Standard-Constructors innerhalb eines Programms:



```
//=====
// PROGRAMM: LIST_BSP_01
//=====

#include <list>
using namespace std;

void main (void)
{
    list<int> aMyIntList;
}
```

### 5.1.2 Der list Constructor mit Größenangabe

Dieser Constructor erzeugt eine neue **list** mit **n** vordefinierten Elementen vom Typ **T**. Für jedes der **n** Elemente wird der Standard-Constructor von **T** (zwecks Initialisierung) aufgerufen.

Syntax:

```
list<T> Variablenname (size_type2 n);
```

Das folgende Beispielprogramm zeigt den Einsatz des Constructors innerhalb eines Programms:



```
//=====
// PROGRAMM: LIST_BSP_02
//=====

#include <list>
using namespace std;

void main (void)
{
    list<int> aMyIntList (5); // mit 5 Elementen beginnen
}
```

<sup>1</sup> **T** ist der Platzhalter für den im **list** verwalteten Datentyp

<sup>2</sup> Logischer Größentyp für eine Anzahl von Elementen vom Typ **T**

Diese Form des Constructors ist besonders nützlich, wenn man eine konstante Menge an zu verwaltenden Elementen verwalten möchte.

### 5.1.3 Der list Constructor mit Größenangabe und Vorlage

Dieser Constructor erzeugt ein neues **list**-Objekt mit **n** vordefinierten Elementen vom Typ **T**. Für jedes der **n** Elemente wird der Copy-Constructor von **T** (zwecks Initialisierung) aufgerufen und somit werden insgesamt **n** Kopien des Objektes **x** erzeugt.

Syntax:

```
list<T> Variablenname (size_type n, T& x);
```

Das folgende Beispielprogramm zeigt den Einsatz des Constructors innerhalb eines Programms:

```
//=====
// PROGRAMM: LIST_BSP_03
//=====

#include <list>
using namespace std;

void main (void)
{
    int Vorlage = 17;
    list<int> aMyIntList (5, Vorlage);
}
```



Auch diese Form des Constructors ist besonders nützlich, wenn man eine konstante Menge an zu verwaltenden Elementen verwalten möchte und es eine individuelle Vorlage gibt, die verwendet werden soll.

Man hat hier die Möglichkeit, eine gezielte Vorbelegung für alle zu erzeugenden Elemente zu treffen, die von der Vorbelegung durch den Standard-Constructor der zu speichernden Elemente **T** abweicht.

### 5.1.4 Der list Constructor mit Bereichsangabe

Dieser Constructor erzeugt eine neue **list** anhand eines bereits bestehenden **list**-Objektes. Die Pointer **pFirst** und **pLast** bezeichnen dabei Anfang- und Endwert eines Bereiches. Das Element, auf welches der Iterator **pLast** verweist gehört **nicht** mehr zum kopierten Bereich – d.h. **pLast** verweist auf das erste Element hinter dem zu kopierenden Bereich.

Für jedes der im Bereich befindlichen Elemente wird einmal der Copy-Constructor von **T** mit dem korrespondierenden Element des übergebenen Bereiches aufgerufen.

Syntax:

```
list<T> Variablenname (T* pFirst, T* pLast);
```

Das folgende Beispielprogramm zeigt, wie der Constructor innerhalb eines Programms verwendet werden kann:



```
//=====
// PROGRAMM: LIST_BSP_04
//=====

#include <list>
using namespace std;

int Feld [7] = {1, 2, 3, 4, 5, 6, 7};

void main (void)
{
    int Wert = 13;
    list<int> aTest (6, Wert);

    // Die gesamte list kopieren
    list<int> aMyIntList (aTest.begin(), aTest.end());
}
```

Anhand der übergebenen Zeiger kann der Constructor die benötigte Anzahl an Elementen ermitteln und eine entsprechend großen Block reservieren. Da der Copy-Constructor von **T** nur einmal für jedes Element aufgerufen wird (im Unterschied zum Constructor mit Größenangabe und Vorlage sind es hier verschiedene Objekte des Typs **T** die kopiert werden), ist diese Constructor-Form ebenso effizient wie der bereits erwähnte Constructor mit Größenangabe und Vorlage.

### 5.1.5 Der list Copy-Constructor

Dieser Constructor erzeugt eine neue **list** anhand eines bereits bestehenden **list**-Objektes. Das neue **list**-Objekt ist eine vollständige Kopie des übergebenen Objektes L.

Syntax:

```
list<T> Variablenname (const list<T>& L);
```

Das folgende Beispielprogramm zeigt, wie der Copy-Constructor verwendet werden kann:



```
//=====
// PROGRAMM: LIST_BSP_05
//=====

#include <list>
using namespace std;

void main (void)
{
    list<int> aMyIntList1 (3);

    // das list-Objekt kopieren
    list<int> aMyIntList2 (aMyIntList1);
}
```

## 5.2 Die list Methoden

Die folgende Tabelle faßt die Methoden der **list** Containerklasse zusammen. In den nachstehenden Abschnitten werden die Methoden einzeln behandelt und mit Beispielen erläutert.

<b>Methoden der Containerklasse LIST</b>	
<i>Methode</i>	<i>Bedeutung</i>
back	Gibt eine Referenz auf das letzte in der <b>list</b> enthaltene Datenelement zurück (also DatentypT&)
begin	Rückgabe eines <b>iterator</b> , der auf das erste in der <b>list</b> enthaltene Element zeigt
empty	Gibt den Wert <b>true</b> zurück, wenn die <b>list</b> keine Datenelemente enthält.
end	Rückgabe eines <b>iterator</b> , der hinter das letzte in der <b>list</b> enthaltene Element zeigt
erase	Löscht ein oder mehrere Datenelemente an der angegebenen Position aus der <b>list</b>
front	Gibt eine Referenz auf das erste in der <b>list</b> enthaltene Datenelement zurück (also DatentypT&)
insert	Fügt ein oder mehrere Datenelemente an der angegebenen Position in die <b>list</b> ein
max_size	Gibt die maximale Anzahl an Datenelementen zurück, welche die <b>list</b> enthalten kann
merge	Vereinigt zwei <b>list</b> -Objekte gleichen Typs zu einem <b>list</b> -Objekt
pop_back	Löscht das letzte Datenelement aus der <b>list</b>
pop_front	Löscht das erste Datenelement aus der <b>list</b>
push_back	Fügt ein Datenelement am Ende der <b>list</b> an
push_front	Fügt ein Datenelement am Anfang der <b>list</b> an
rbegin	Rückgabe eines <b>reverse_iterator</b> , der auf das letzte in der <b>list</b> enthaltene Element zeigt
remove	Löschen aller <b>list</b> -Einträge, die einem bestimmten Wert entsprechen
rend	Rückgabe eines <b>reverse_iterator</b> , der vor das erste in der <b>list</b> enthaltene Element zeigt
reverse	Ordnet die Datenelemente der <b>list</b> in umgekehrter Reihenfolge an
size	Gibt die Anzahl von Datensätzen zurück, die aktuell in der <b>list</b> enthalten sind
sort	Optimierte Sortierung des Inhalts der <b>list</b> mit Hilfe des Operators <.
splice	Aufsplitten und Anfügen einer anderen <b>list</b> gleichen Typs
swap	Tauscht den Inhalt zweier <b>list</b> -Objekte aus
unique	Entfernen gleicher, benachbarter Datenelemente aus der <b>list</b>

Tabelle 5-1– Methoden der Containerklasse list

### 5.2.1 Die list Methode back

Die Methode **back()** gibt eine Referenz oder einen **const**-Referenz auf das letzte in der **list** gespeicherte Datenelement zurück. Im Gegensatz zu **end()** handelt es sich um eine Objektreferenz vom Typ **T** und nicht einen **iterator** (welcher zudem hinter das letzte Element zeigen würde).

Der Aufruf von **back()** selbst verändert den Inhalt der **list** nicht.

Syntax:

```
T& back ();
const T& back () const;
```

Das nachfolgende Beispielprogramm zeigt den Unterschied zwischen **back()** und **end()** anhand einer Ausgabeanweisung:



```
//=====
// PROGRAMM: LIST_BSP_06
//=====

#include <iomanip.h>
#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
    list<double> aList;
    list<double>::const_iterator aI;

    aList.push_back (7.0);

    aI = aList.end();
    aI--;
    cout << *aI << endl;          // Ausgabe Derefer. von aI
    cout << aList.back() << endl; // Ausgabe über Referenz
}

```

### 5.2.2 Die list Methode begin

Rückgabe eines **iterator** oder **const\_iterator**, der auf das erste in der **list** enthaltene Element zeigt.

Syntax:

```
iterator begin ();
const_iterator begin () const;
```

Die wohl häufigste Anwendung der Methode **begin()** liegt in der Verarbeitung von Daten durch Schleifen.  
Der Aufruf von **begin()** verändert den Inhalt der **list** nicht.



```
//=====
// PROGRAMM: LIST_BSP_07
//=====

#include <iomanip.h>
#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
    list<double> aList;
    list<double>::iterator aI;

    aList.push_back (7.0);
    aList.push_back (1.2);
}

```

```

aList.push_back (3.14);

for (aI=aList.begin(); aI!=aList.end(); aI++)
{
    cout << "Wert: " << *aI << endl;
}
}

```

### 5.2.3 Die list Methode empty

Die Methode **empty()** prüft, ob die **list** Datenelemente enthält oder nicht. Sind Datenelemente enthalten (dies entspricht einem Rückgabewert von **size()** größer als Null), so wird der Wert **false** zurückgegeben, sind keine Datenelemente vorhanden ist der Rückgabewert **true**.

Der Aufruf von **empty()** verändert den Inhalt der **list** nicht.

Syntax:

```
bool empty () const;
```

```

//=====
// PROGRAMM: LIST_BSP_08
//=====

#include <iomanip.h>
#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
    list<double> aList;

    if (aList.empty())
    {
        cout << "List ist leer" << endl;
    }
    else
    {
        cout << "List enthält Daten" << endl;
    }

    aList.push_back (7.0);
    aList.push_back (3.14);

    if (aList.empty())
    {
        cout << "List ist leer" << endl;
    }
    else
    {
        cout << "List enthält Daten" << endl;
    }
}

```



### 5.2.4 Die list Methode end

Rückgabe eines **iterator** oder **const\_iterator**, der *hinter* das letzte in der **list** enthaltene Element zeigt.

Syntax:

```
iterator end ();
const_iterator end () const;
```

Die wohl häufigste Anwendung der Methode **end()** liegt in der Verarbeitung von Daten durch Schleifen.

Es ist zu beachten, daß **end()** hinter das letzte gültige Datenelement zeigt, alle Schleifen müssen also auf „ungleich“ (Operator **!=**) oder „echt kleiner als“ prüfen (Operator **<**) und nicht auf „kleiner oder gleich“ (Operator **<=**). Zeigt ein **iterator** auf **end()** führt jeder dereferenzierende Zugriff zu einer Exception vom Typ **out\_of\_range**. Wird die Exception nicht abgefangen (wie zweiten Beispiel), bricht das Programm unkontrolliert ab (Absturz).

Der Aufruf von **end()** verändert den Inhalt der **list** nicht.



```
//=====
// PROGRAMM: LIST_BSP_09
//=====

#include <iomanip.h>
#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
    list<double> aList;
    list<double>::iterator aI;

    aList.push_back (7.0);
    aList.push_back (1.2);
    aList.push_back (3.14);

    for (aI=aList.begin(); aI!=aList.end(); aI++)
    {
        cout << "Wert: " << *aI << endl;
    }
}
```

Ein Beispiel, welches die Auslösung einer Exception aufgrund eines illegalen Zugriffs auf **end()** zeigt, ist im Abschnitt zur Methode **end()** bei der Containerklasse **vector** aufgeführt.

### 5.2.5 Die list Methode erase

Löscht ein oder mehrere Datenelemente an der angegebenen Position aus der **list**.

Syntax:

```
void erase(iterator toDel);
void erase(iterator First, iterator Last);
```

Die Methode **erase()** kann verwendet werden, um einzelne Datensätze oder ganze Bereiche aus der **list** zu löschen.

*Im Gegensatz zum vector und dem deque werden durch das Löschen in der list nur die Iteratoren,*

*Verweise und Referenzen auf Datensätze im gelöschten Bereich ungültig.*

*Da es sich bei der List nicht um ein Array handelt (welches als geschlossener Block gespeichert werden muß), gibt es keine Notwendigkeit, das list-Objekt im Speicher zu verschieben.*

*Iteratoren, Verweise und Referenzen, die nicht auf einen durch die Methode gelöschten Bereich zeigen, bleiben daher gültig.*

Die erste Form der Methode sorgt dafür, daß ein einzelner Datensatz aus der **list** gelöscht und der Destructor der Klasse **T** für diesen Datensatz aufgerufen wird.

Bei der zweiten Syntaxform wird ein zu löschender Bereich angegeben, für jedes zu löschende Element wird der Destructor der Klasse **T** aufgerufen. Die Angabe erfolgt durch zwei Iteratoren, die auf das erste und das letzte zu löschende Element zeigen.

Es ist besonders bei den Bereichsangaben zu beachten, daß nicht versehentlich eine **out\_of\_range** Exception erzeugt wird.

Ein Beispiel, welches die Auslösung einer Exception aufgrund eines illegalen Zugriffs durch **erase()** zeigt, ist im Abschnitt zur Methode **erase()** bei der Containerklasse **vector** aufgeführt.

### 5.2.6 Die list Methode front

Die Methode **front()** gibt eine Referenz oder einen **const**-Referenz auf das erste in der **list** gespeicherte Datenelement zurück. Im Gegensatz zu **begin()** handelt es sich um eine Objektreferenz vom Typ **T** und nicht einen **iterator**.

Der Aufruf von **front()** selbst verändert den Inhalt der **list** nicht.

Syntax:

```
T& front ();
const T& front () const;
```

Das nachfolgende Beispielprogramm zeigt den Unterschied zwischen **front()** und **begin()** anhand einer Ausgabeanweisung:

```
//=====
// PROGRAMM: LIST_BSP_10
//=====

#include <iomanip.h>
#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
    list<double> aList;
    list<double>::const_iterator aI;

    aList.push_back (7.0);

    aI = aList.begin();
```



```

    cout << *aI << endl;           // Ausgabe Derefer. v. aI
    cout << aList.front() << endl; // Ausgabe über Referenz
}

```

### 5.2.7 Die list Methode insert

Fügt ein oder mehrere Datenelemente vor der angegebenen Position in der **list** ein.

Syntax:

```

iterator insert (iterator pos, T& value);
void          insert (iterator pos, size_type Anz,
                    T& value);
void          insert (iterator pos, const T *First,
                    const T *Last);

```

Die Methode **insert()** kann verwendet werden um einzelne Datensätze oder ganze Bereiche in eine **list** einzufügen.

Da eine **list** als verkettete Liste aufgebaut ist, muß (anders als bei **vector** oder **deque**) beim Einfügen von Datensätzen nichts umkopiert werden, was die Methode **insert()** für **list**-Objekte sehr effizient macht. Der Zeitaufwand für das Einfügen in einem **list**-Objekt ist daher immer konstant.

*Im Gegensatz zum vector und dem deque werden durch das Einfügen keine Iteratoren, Verweise oder Referenzen ungültig.*

Die erste Form der Methode sorgt dafür, daß ein einzelner Datensatz (**value**) vor der angegebenen Position (**pos**) in die **list** eingefügt wird.

Dazu wird der Copy-Constructor der Klasse **T** für den Parameter **value** aufgerufen und die so entstandene Kopie in die **list** eingefügt.

Die zweite Form der **insert()** Methode fügt mehrere (**Anz**) Kopien des Datensatzes **value** an der Position **pos** ein. Der Copy-Constructor der Klasse **T** wird **Anz**-mal für den Parameter **value** aufgerufen und die so entstandenen Kopien in die **list** eingefügt.

Mit der dritten Methode können Bereiche aus einem Array in die **list** kopiert werden (hier anhand eines Ausschnittes aus dem Array **fArray** dargestellt).

Der Bereich wird durch Anfangs- und Endadresse festgelegt. Es sollte darauf geachtet werden, daß die Endadresse nicht vor der Anfangsadresse liegen darf und daß beide Adressen zum gleichen Array bzw. gleichen **list**-Objekt gehören.



```

//=====
// PROGRAMM: LIST_BSP_11
//=====

#include <iomanip.h>
#include <iostream.h>
#include <list>
using namespace std;

double fArray [6] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0};

```

```

void main (void)
{
    list<double> aList;
    list<double> aL2;
    list<double>::iterator aI;

    aList.push_back (1.1);
    aList.push_back (2.2);
    aList.push_back (3.3);

    aL2.push_back (11.1);
    aL2.push_back (22.2);
    aL2.push_back (33.3);

    //-----
    // 1. Form
    //-----
    aI = aList.begin();
    aI++; // Auf 2. Element zeigen
    aList.insert (aI, 4.4); // vor 2. Element einfügen

    //-----
    // 2. Form
    //-----
    aI = aList.begin();
    aI++; // wegen möglicher Reallokation
    aList.insert (aI, 6, 5.5); // 6 Kopien einfügen

    //-----
    // 3. Form
    //-----
    aI = aList.begin();
    aI++; // wegen möglicher Reallokation
    aList.insert(aI, fArray+1, fArray+4); // Elemente aus Array

    //-----
    // 3. Form wie oben, aber mit Indexschreibweise
    //-----
    aI = aList.begin();
    aI++; // wegen möglicher Reallok.
    aList.insert(aI, &fArray[1], &fArray[4]); // Elem aus Array

    for (aI=aList.begin(); aI!=aList.end(); aI++)
    {
        cout << *aI << endl;
    }
}

```

### 5.2.8 Die list Methode max\_size

Gibt die maximale Anzahl an Datenelementen zurück, die die **list** insgesamt enthalten kann.

Syntax:

```
size_type max_size () const;
```

Der Wert ist abhängig von der Implementation und vom verwendeten Betriebssystem. Bei modernen Compilern und entsprechender Einstellung kann man davon ausgehen, dass 32-Bit Adressen verwendet werden, also theoretisch 4 Gigabytes angesprochen werden können.

Da **list**-Objekte nicht als Block gespeichert sein müssen, wird die Anzahl der Datenelemente nur durch den freien RAM-Speicher begrenzt, die Größe der zu verwaltenden Datenelemente, sowie die Segmentierung (Zersplitterung) des Speichers.

Die Methode **max\_size()** dient dazu abschätzen zu können, ob eine aufzubauende **list** überhaupt in den Speicher paßt.

Der Aufruf der Methode verändert den Inhalt der **list** nicht.



```
//=====
// PROGRAMM: LIST_BSP_12
//=====

#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
    list<double> aList;

    cout << "List kann " << aList.max_size ()
         << " Datenelemente aufnehmen";
}

```

### 5.2.9 Die list Methode merge

Vereinigt zwei **list**-Objekte gleichen Typs zu einem **list**-Objekt.

Syntax:

```
void merge (const list<T>& L);
```

Die Methode **merge()** verschiebt alle Elemente der als Parameter übergebenen **list L** in die **list** für welche die Methode **merge()** aufgerufen wurde. Die Verschiebung erfolgt so, daß die Datenelemente dabei durch den Operator „<“ geordnet sind. Voraussetzung, um sinnvolle Ergebnisse zu erzielen, ist allerdings, daß beide **list**-Objekte bereits gemäß des Operators „<“ geordnet sind (siehe auch Methode **sort()**).

Die übergebene **list** ist anschließend leer (im nachfolgenden Beispiel **aL2**)



```
//=====
// PROGRAMM: LIST_BSP_13
//=====

#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
    list<double> aList;
    list<double> aL2;
    list<double>::iterator aI;

    aList.push_back (1.2);
    aList.push_back (2.4);
    aL2.push_back (1.1);
    aL2.push_back (2.3);
}

```

```

aList.merge (aL2);

for (aI=aList.begin(); aI!=aList.end(); aI++)
{
    cout << *aI << endl;
}

for (aI=aL2.begin(); aI!=aL2.end(); aI++)
{
    cout << *aI << endl;
}
}

```

### 5.2.10 Die list Methode pop\_back

Löscht das letzte Datenelement aus der **list**.

Syntax:

```
void pop_back ();
```

Die Methode **pop\_back()** wird verwendet um einen einzelnen Datensatz vom Ende der **list** zu entfernen. Im Gegensatz zum **vector** und zum **deque** ist es für das Laufzeitverhalten des Programms unerheblich, ob die Daten mit der Methode **erase()** oder als wiederholter Aufruf von **pop\_back()** entfernt werden.

*Durch das Löschen werden daß alle Iteratoren – sowie Verweise und Referenzen auf Datensätze im gelöschten Bereich – ungültig.*

Zum Löschen wird der Destructor der Klasse **T** für das letzte Element in der **list** aufgerufen.

```

//=====
// PROGRAMM: LIST_BSP_14
//=====

#include <iomanip.h>
#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
    list<double> aList;
    list<double>::iterator aI;

    aList.push_back (1.1);
    aList.push_back (2.2);
    aList.push_back (3.3);

    aList.pop_back ();

    for (aI=aList.begin(); aI!=aList.end(); aI++)
    {
        cout << *aI << endl;
    }
}

```



### 5.2.11 Die list Methode pop\_front

Löscht das erste Datenelement aus der **list**.

Syntax:  

```
void pop_front ();
```

Die Methode **pop\_front()** wird verwendet um einen einzelnen Datensatz vom Beginn der **list** zu entfernen. Im Gegensatz zum **vector** und zum **deque** ist es für das Laufzeitverhalten des Programms unerheblich, ob die Daten mit der Methode **erase()** oder als wiederholter Aufruf von **pop\_front()** entfernt werden.

*Durch das Löschen werden daß alle Iteratoren – sowie Verweise und Referenzen auf Datensätze im gelöschten Bereich – ungültig.*

Zum Löschen wird der Destructor der Klasse **T** für das erste Element in der **list** aufgerufen.



```
//=====
// PROGRAMM: LIST_BSP_15
//=====

#include <iomanip.h>
#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
    list<double> aList;
    list<double>::iterator aI;

    aList.push_back (1.1);
    aList.push_back (2.2);
    aList.push_back (3.3);

    aList.pop_front ();

    for (aI=aList.begin(); aI!=aList.end(); aI++)
    {
        cout << *aI << endl;
    }
}
```

### 5.2.12 Die list Methode push\_back

Fügt ein Datenelement am Ende der **list** an.

Syntax:  

```
void push_back (const T& value);
```

Die Methode **push\_back()** wird verwendet um einen einzelnen Datensatz an das Ende der **list** einzufügen. Im Gegensatz zum **vector** und dem **deque** ist es unerheblich, ob die Datensätze mit **insert()** oder als wiederholter Aufruf von **push\_back()** eingefügt werden.

Zum Anfügen wird der Copy-Constructor der Klasse **T** für den Parameter **value** aufgerufen und die so entstandene Kopie an die **list** als letzter Datensatz angehängt.

```
//=====
// PROGRAMM: LIST_BSP_16
//=====

#include <iomanip.h>
#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
    list<double> aList;
    list<double>::iterator aI;

    aList.push_back (1.1);
    aList.push_back (2.2);

    for (aI=aList.begin(); aI!=aList.end(); aI++)
    {
        cout << *aI << endl;
    }
}
```



### 5.2.13 Die list Methode push\_front

Fügt ein Datenelement am Anfang der **list** an.

Syntax:

```
void push_front (const T& value);
```

Die Methode **push\_front()** wird verwendet um einen einzelnen Datensatz an den Beginn der **list** einzufügen. Im Gegensatz zum **vector** und zum **deque** ist es für das Laufzeitverhalten des Programms unerheblich, ob die Daten mit der Methode **insert()** oder als wiederholter Aufruf von **push\_front()** angefügt werden.

Zum Anfügen wird der Copy-Constructor der Klasse **T** für den Parameter **value** aufgerufen und die so entstandene Kopie an die **list** als erster Datensatz angefügt.

```
//=====
// PROGRAMM: LIST_BSP_17
//=====

#include <iomanip.h>
#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
    list<double> aList;
    list<double>::iterator aI;

    aList.push_front (1.1);
```



```

aList.push_front (2.2);

for (aI=aList.begin(); aI!=aList.end(); aI++)
{
    cout << *aI << endl;
}
}

```

### 5.2.14 Die list Methode rbegin

Rückgabe eines **reverse\_iterator** oder **const\_reverse\_iterator**, der auf das letzte in der **list** enthaltene Element zeigt.

Syntax:

```

reverse_iterator rbegin ();
const_reverse_iterator rbegin () const;

```

Die wohl häufigste Anwendung der Methode **rbegin()** liegt in der Verarbeitung von Daten durch rückwärts gerichtete Schleifen. Der Aufruf von **rbegin()** verändert den Inhalt der **list** nicht.



```

//=====
// PROGRAMM: LIST_BSP_18
//=====

#include <iomanip.h>
#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
    list<double> aList;
    list<double>::reverse_iterator aI;

    aList.push_back (7.0);
    aList.push_back (1.2);
    aList.push_back (3.14);

    for (aI=aList.rbegin(); aI!=aList.rend(); aI++)
    {
        cout << "Wert: " << *aI << endl;
    }
}

```

### 5.2.15 Die list Methode remove

Die Methode **remove()** kann dazu verwendet werden, alle Datenelemente aus der **list** zu löschen, deren Inhalt identisch mit dem übergebenen Wert **value** ist.

Syntax:

```

void remove (const T& value);

```



```

//=====
// PROGRAMM: LIST_BSP_19
//=====

```

```

#include <iomanip.h>
#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
    list<double> aList;
    list<double>::reverse_iterator aI;
    double fToDel = 1.0;

    aList.push_back (1.0);
    aList.push_back (1.2);
    aList.push_back (1.0);

    for (aI=aList.rbegin(); aI!=aList.rend(); aI++)
    {
        cout << "Wert: " << *aI << endl;
    }

    aList.remove (fToDel);

    for (aI=aList.rbegin(); aI!=aList.rend(); aI++)
    {
        cout << "Wert: " << *aI << endl;
    }
}

```

### 5.2.16 Die list Methode rend

Rückgabe eines **reverse\_iterator** oder **const\_reverse\_iterator**, der vor das erste in der **list** enthaltene Element zeigt.

Syntax:

```

reverse_iterator rend ();
const_reverse_iterator rend () const;

```

Die wohl häufigste Anwendung der Methode **rend()** liegt in der Verarbeitung von Daten durch rückwärts gerichtete Schleifen.

Es ist zu beachten, daß **rend()** vor das erste gültige Datenelement zeigt, alle Schleifen müssen also auf „ungleich“ prüfen (Operator !=). Zeigt ein **iterator** auf **rend()** führt jeder dereferenzierende Zugriff zu einer Exception vom Typ **out\_of\_range**. Wird die Exception nicht wie zweiten Beispiel abgefangen bricht das Programm unkontrolliert ab (Absturz).

Der Aufruf von **rend()** verändert den Inhalt der **list** nicht.

```

//=====
// PROGRAMM: LIST_BSP_20
//=====

#include <iomanip.h>
#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
    list<double> aList;
    list<double>::reverse_iterator aI;

```



```

aList.push_back (7.0);
aList.push_back (1.2);
aList.push_back (3.14);

for (aI=aList.rbegin(); aI!=aList.rend(); aI++)
{
    cout << "Wert: " << *aI << endl;
}
}

```

Ein Beispiel, welches die Auslösung einer Exception aufgrund eines illegalen Zugriffs auf **rend()** zeigt, ist im Abschnitt über die Methode **rend()** der Containerklasse **vector** zu finden.

### 5.2.17 Die list Methode reverse

Die Methode **reverse()** kehrt die Verkettungsstruktur der **list** um, so daß anschließend die enthaltenen Datenelemente in umgekehrter Reihenfolge vorliegen.

Syntax:  
void reverse ();



```

//=====
// PROGRAMM: LIST_BSP_21
//=====

#include <iomanip.h>
#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
    list<double> aList;
    list<double>::iterator aI;

    aList.push_back (1.2);
    aList.push_back (3.14);
    aList.push_back (6.14);

    for (aI=aList.begin(); aI!=aList.end(); aI++)
    {
        cout << "Wert: " << *aI << endl;
    }

    aList.reverse();

    for (aI=aList.begin(); aI!=aList.end(); aI++)
    {
        cout << "Wert: " << *aI << endl;
    }
}

```

### 5.2.18 Die list Methode size

Gibt die Anzahl von Datensätzen zurück, die aktuell in der **list** enthalten sind.

Syntax:

```
size_type size () const;
```

Der Aufruf von **size()** verändert den Inhalt der **list** nicht.

```
//=====
// PROGRAMM: LIST_BSP_22
//=====

#include <iomanip.h>
#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
    list<double> aList;

    aList.push_back (1.2);
    aList.push_back (3.14);

    cout << "Anzahl aktuell: " << aList.size() << endl;
}
```



### 5.2.19 Die list Methode sort

Sortiert eine **list** anhand des „Kleiner“-Operators („<“).

Syntax:

```
void sort ();
```

Containerobjekte vom Typ **list** sind die einzigen Typen in der STL, die über einen eigenen, optimierten Sortieralgorithmus verfügen. Dieser ist aufgrund der speziell für **list**-Objekte vorgenommenen Optimierungen dem verallgemeinerten Sortierverfahren der mitgelieferten Algorithmen-Bibliothek vorzuziehen (siehe Abschnitt über generische Algorithmen). Die Zeitkomplexität des verwendeten Sortierverfahrens ist  $O(N \log N)$ .

```
//=====
// PROGRAMM: LIST_BSP_23
//=====

#include <iomanip.h>
#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
    list<double> aList;
    list<double>::iterator aI;

    aList.push_back (1.2);
    aList.push_back (6.14);
    aList.push_back (16.4);
    aList.push_back (3.14);

    for (aI=aList.begin(); aI!=aList.end(); aI++)
    {
        cout << "Wert: " << *aI << endl;
    }
}
```



```

}

aList.sort();

for (aI=aList.begin(); aI!=aList.end(); aI++)
{
    cout << "Wert: " << *aI << endl;
}
}

```

### 5.2.20 Die list Methode splice

Die Methode **splice()** dient zur Teilung eines **list**-Objektes und Anfügen an ein anderes Objekt gleichen Typs.

Syntax:

```

void splice (iterator To, list<T>& L);
void splice (iterator To, list<T>& L, iterator From);
void splice (iterator To, list<T>& L,
            iterator First, iterator Last);

```

Die erste Form der Methode **splice()** löscht alle Elemente aus der Liste **L** und fügt sie ab der Position **To** in das aktuelle **list**-Objekt ein (das Objekt für das die Methode **splice()** aufgerufen wurde).



```

//=====
// PROGRAMM: LIST_BSP_24
//=====

#include <iomanip.h>
#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
    list<double> aList;
    list<double> aL2;
    list<double>::iterator aI;

    aList.push_back (1.1);
    aList.push_back (2.2);

    aL2.push_back (11.11);
    aL2.push_back (22.22);

    //-----
    // Splice-Methode Nr. 1
    //-----
    cout << "Splice 1" << endl;
    for (aI=aList.begin(); aI!=aList.end(); aI++)
    {
        cout << "Wert: " << *aI << endl;
    }

    aI = aList.begin();
    aList.splice(aI, aL2);

    for (aI=aList.begin(); aI!=aList.end(); aI++)
    {

```

```

    cout << "Wert: " << *aI << endl;
}
}

```

Die zweite Form löscht alle Elemente aus der Liste **L** ab der Position **From** und fügt sie ab der Position **To** in das aktuelle **list**-Objekt ein.

```

//=====
// PROGRAMM: LIST_BSP_25
//=====

#include <iomanip.h>
#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
    list<double> aList;
    list<double> aL2;
    list<double>::iterator aI;
    list<double>::iterator aJ;

    aList.push_back (1.1);
    aList.push_back (2.2);

    aL2.push_back (11.11);
    aL2.push_back (22.22);
    aL2.push_back (33.33);

    //-----
    // Splice-Methode Nr. 2
    //-----
    cout << "Splice 2" << endl;
    for (aI=aList.begin(); aI!=aList.end(); aI++)
    {
        cout << "Wert: " << *aI << endl;
    }

    aI = aList.begin();
    aI++;
    aJ = aL2.begin();
    aJ++;
    aList.splice(aI, aL2, aJ);

    for (aI=aList.begin(); aI!=aList.end(); aI++)
    {
        cout << "Wert: " << *aI << endl;
    }
}

```

Die dritte Form löscht alle Elemente aus der Liste **L** ab der Position **First** bis zur Position **Last** (Last selbst wird nicht mehr mit verschoben) und fügt sie ab der Position **To** in das aktuelle **list**-Objekt ein.

```

//=====
// PROGRAMM: LIST_BSP_26
//=====

#include <iomanip.h>

```

```

#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
    list<double> aList;
    list<double> aL2;
    list<double>::iterator aI;
    list<double>::iterator aJ;
    list<double>::iterator aK;

    aList.push_back (1.1);
    aList.push_back (2.2);

    aL2.push_back (11.11);
    aL2.push_back (22.22);
    aL2.push_back (33.33);
    aL2.push_back (44.44);
    aL2.push_back (55.55);
    aL2.push_back (66.66);

    //-----
    // Splice-Methode Nr. 3
    //-----
    cout << "Splice 3" << endl;
    for (aI=aList.begin(); aI!=aList.end(); aI++)
    {
        cout << "Wert: " << *aI << endl;
    }

    aI = aList.begin();
    aI++;
    aJ = aL2.begin();
    aJ++;
    aK = aL2.end();
    aK--;
    aK--;
    aList.splice(aI, aL2, aJ, aK);

    for (aI=aList.begin(); aI!=aList.end(); aI++)
    {
        cout << "Wert: " << *aI << endl;
    }
}

```

### 5.2.21 Die list Methode swap

Tauscht die Inhalte zweier **list**-Objekte aus. Die **list**-Objekte müssen von gleichem Typ sein, also gleiche Datenelementtypen enthalten.

Syntax:

```
void swap (list<T>& v);
```

*Es ist zu beachten, daß somit alle Iteratoren, Verweise und Referenzen ungültig werden, da es sich um Pointer handelt und die Datensätze durch den Tausch im Speicher verschoben werden.*

*Alle Iteratoren, Verweise und Referenzen müssen neu ermittelt werden.*

```
//=====
// PROGRAMM: LIST_BSP_27
//=====

#include <iomanip.h>
#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
    list<double> aList;
    list<double> aL2;
    list<double>::iterator aI;

    aList.push_back (1.2);
    aList.push_back (3.14);

    aL2.push_back (222.22);
    aL2.push_back (333.33);
    aL2.push_back (111.11);
    aL2.push_back (444.44);

    for (aI=aList.begin(); aI!=aList.end(); aI++)
    {
        cout << "aList Wert: " << *aI << endl;
    }

    for (aI=aL2.begin(); aI!=aL2.end(); aI++)
    {
        cout << "aL2 Wert: " << *aI << endl;
    }

    aList.swap (aL2);

    for (aI=aList.begin(); aI!=aList.end(); aI++)
    {
        cout << "aList Wert: " << *aI << endl;
    }

    for (aI=aL2.begin(); aI!=aL2.end(); aI++)
    {
        cout << "aL2 Wert: " << *aI << endl;
    }
}
```



### 5.2.22 Die list Methode unique

Die Methode **unique()** sorgt dafür, daß sich wiederholende Datenelemente entfernt werden, so daß jedes Datenelement nur einmal vorkommt.

Um alle Dubletten zu entfernen, ist es notwendig, daß die **list** sortiert vorliegt. Die Methode **unique()** prüft nur jeweils zwei aufeinanderfolgende Elemente.

Syntax:

```
void unique ();
```



```
//=====
// PROGRAMM: LIST_BSP_28
//=====

#include <iomanip.h>
#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
    list<double> aList;
    list<double> aL2;
    list<double>::iterator aI;

    aList.push_back (1.1);
    aList.push_back (2.2);
    aList.push_back (2.2);
    aList.push_back (2.2);
    aList.push_back (3.2);
    aList.push_back (2.2);

    for (aI=aList.begin(); aI!=aList.end(); aI++)
    {
        cout << "aList Wert: " << *aI << endl;
    }

    aList.unique();

    for (aI=aList.begin(); aI!=aList.end(); aI++)
    {
        cout << "aList Wert: " << *aI << endl;
    }
}
```

### 5.3 Die list Operatoren

Mit den folgenden Operatoren kann auf eine Containerklasse vom Typ **list** zugegriffen werden:

<b>Operatoren der Containerklasse LIST</b>	
<i>Operato r</i>	<i>Bedeutung</i>
=	Zuweisung zwischen zwei <b>list</b> -Objekten gleichen Typs
==	Vergleich zwischen zwei <b>list</b> -Objekten gleichen Typs
!=	Vergleich zwischen zwei <b>list</b> -Objekten gleichen Typs
<	Lexikographischer Vergleich zwischen zwei <b>list</b> -Objekten gleichen Typs
>	Lexikographischer Vergleich zwischen zwei <b>list</b> -Objekten gleichen Typs
>=	Lexikographischer Vergleich zwischen zwei <b>list</b> -Objekten gleichen Typs
<=	Lexikographischer Vergleich zwischen zwei <b>list</b> -Objekten gleichen Typs

Tabelle 5-1– Operatoren der Containerklasse list

### 5.3.1 Der list Operator =

Der Operator ersetzt den Inhalt eines **list**-Objektes durch den Inhalt des zugewiesenen **list**-Objektes gleichen Typs.

Syntax:

```
list<T>& operator= (const list<T>& v);
```

*Durch die Zuweisung werden alle Iteratoren, Verweise und Referenzen auf dieser List ungültig. Alle Iteratoren, Verweise und Referenzen müssen neu ermittelt werden.*

```
//=====
// PROGRAMM: LIST_BSP_29
//=====

#include <iomanip.h>
#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
    list<double> aList;
    list<double> aL2;
    list<double>::iterator aI;

    aList.push_back (7.0);
    aList.push_back (1.2);
    aList.push_back (3.14);

    aL2 = aList;

    for (aI=aList.begin(); aI!=aList.end(); aI++)
    {
        cout << "aList Wert: " << *aI << endl;
    }

    for (aI=aL2.begin(); aI!=aL2.end(); aI++)
    {
        cout << "aL2 Wert: " << *aI << endl;
    }
}
```



### 5.3.2 Der list Operator ==

Der Operator prüft, ob die Inhalte des linken und rechten Operanden (**list**-Objekte gleichen Typs) übereinstimmen. Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn beide **list**-Objekte die gleichen Elemente in der gleichen Reihenfolge enthalten.

Syntax:

```
bool operator== (const list<T>& v) const;
```



```
//=====
// PROGRAMM: LIST_BSP_30
//=====

#include <iomanip.h>
#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
    list<double> aList;
    list<double> aL2;

    aList.push_back (3.14);

    if (aList == aL2)
    {
        cout << "Die List-Objekte sind gleich" << endl;
    }
    else
    {
        cout << "Die List-Objekte sind nicht gleich" << endl;
    }

    aL2 = aList;

    if (aList == aL2)
    {
        cout << "Die List-Objekte sind gleich" << endl;
    }
    else
    {
        cout << "Die List-Objekte sind nicht gleich" << endl;
    }
}
}
```

### 5.3.3 Der list Operator !=

Der Operator prüft, ob die Inhalte des linken und rechten Operanden (**list**-Objekte gleichen Typs) verschieden sind. Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn beide **list**-Objekte unterschiedliche Elemente oder gleiche Elemente in unterschiedlicher Reihenfolge enthalten.

Syntax:

```
bool operator!= (const list<T>& v) const;
```



```
//=====
// PROGRAMM: LIST_BSP_31
//=====

#include <iomanip.h>
#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
```

```

list<double> aList;
list<double> aL2;

aList.push_back (3.14);

if (aList != aL2)
{
    cout << "Die List-Objekte sind nicht gleich" << endl;
}
else
{
    cout << "Die List-Objekte sind gleich" << endl;
}

aL2 = aList;

if (aList != aL2)
{
    cout << "Die List-Objekte sind nicht gleich" << endl;
}
else
{
    cout << "Die List-Objekte sind gleich" << endl;
}
}

```

### 5.3.4 Der list Operator <

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch kleiner ist, als der des rechten Operanden (**list**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „<“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch kleiner ist als der rechte Operand.

Syntax:

```
bool operator< (const list<T>& v) const;
```

```

//=====
// PROGRAMM: LIST_BSP_32
//=====

#include <iomanip.h>
#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
    list<double> aList;
    list<double> aL2;

    aList.push_back (3.14);
    aL2.push_back (3.15);

    if (aList < aL2)
    {
        cout << "aList ist kleiner als aL2" << endl;
    }
    else
    {

```



```

    cout << " aList ist nicht kleiner als aL2" << endl;
}
}

```

### 5.3.5 Der list Operator >

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch größer ist, als der des rechten Operanden (**list**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „>“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch größer ist als der rechte Operand.

Syntax:

```
bool operator> (const list<T>& v) const;
```



```

//=====
// PROGRAMM: LIST_BSP_33
//=====

#include <iomanip.h>
#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
    list<double> aList;
    list<double> aL2;

    aList.push_back (3.14);
    aL2.push_back (3.15);

    if (aList > aL2)
    {
        cout << "aList ist größer als aL2" << endl;
    }
    else
    {
        cout << " aList ist nicht größer als aL2" << endl;
    }
}

```

### 5.3.6 Der list Operator <=

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch kleiner oder gleich mit dem rechten Operanden ist (**list**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „<=“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch kleiner oder gleich mit dem rechten Operanden ist.

Syntax:

```
bool operator<= (const list<T>& v) const;
```



```

//=====
// PROGRAMM: LIST_BSP_34
//=====

```

```

#include <iomanip.h>
#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
    list<double> aList;
    list<double> aL2;

    aList.push_back (3.14);
    aL2.push_back (3.15);

    if (aList <= aL2)
    {
        cout << "aList ist kleiner-gleich aL2" << endl;
    }
    else
    {
        cout << " aList ist nicht kleiner-gleich als aL2" << endl;
    }
}

```

### 5.3.7 Der list Operator >=

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch größer oder gleich mit dem rechten Operanden ist (**list**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „>=“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch größer oder gleich mit dem rechten Operanden ist.

Syntax:

```
bool operator>= (const list<T>& v) const;
```

```

//=====
// PROGRAMM: LIST_BSP_35
//=====

#include <iomanip.h>
#include <iostream.h>
#include <list>
using namespace std;

void main (void)
{
    list<double> aList;
    list<double> aL2;

    aList.push_back (3.14);
    aL2.push_back (3.15);

    if (aList >= aL2)
    {
        cout << "aList ist größer-gleich aL2" << endl;
    }
    else
    {
        cout << " aList ist nicht größer-gleich als aL2" << endl;
    }
}

```



## 6. Die Containerklasse set

Die Containerklasse **set** realisiert einen assoziativen Behälter (auch Hashtable genannt), welcher Schlüssel beinhaltet (Keys). Der **set** ist durch eine vom Benutzer definierte Vergleichsfunktion (Compareclass) sortiert. Die Containerklasse setzt voraus, daß jeder Schlüssel nur ein einziges mal (unique) in einem **set** vorkommen kann.

Die interne Struktur eines **set** (Hashtable) ermöglicht es, einen Key sehr effizient zu ermitteln<sup>1</sup>.

Eine häufige Form der Verwendung von **set** ist z.B. die Markierung von durchlaufenen Programmknoten oder die Ordnung von Datenlisten, bei denen jede Einstellung bzw. jedes Datum nur einmal vorkommen kann.

Ein **set**-Objekt hat intern einen anderen Aufbau (Baumstruktur) als ein **vector**-, **deque**- oder **list**-Objekt, um eine effiziente Suche zu ermöglichen.

Die notwendige Speicherverwaltung wird automatisch vorgenommen und ist so effizient wie möglich realisiert. Wie für **list**-Objekte wird auch für einen **set** kein geschlossener Speicherbereich benötigt. Um ein **set**-Objekt zu erweitern ist daher kein Umkopieren der gesamten Containerklasse notwendig, da lediglich die Verkettung innerhalb der Baumstruktur geändert werden muß.

Ein wahlfreier Zugriff auf **set**-Objekte ist nicht möglich, der **operator[]** (Zugriff über Indexklammer) ist daher nicht definiert.

In einem Punkt weicht **set** von den bisher beschriebenen Containerklassen **vector**, **deque** und **list** ab. Während diese lediglich eine Klassenangabe erwarten, um die Containerklasse zu definieren, also lediglich die zu verwaltende Klasse benötigen, braucht ein **set**-Objekt insgesamt zwei Klassenangaben. Die erste erforderliche Angabe verweist auf die Klasse, welche den Key repräsentiert, die zweite definiert die Vergleichsklasse für die Sortierung.

Die Deklaration eines **set**-Objektyps kann durch eine **typedef**-Anweisung weiter vereinfacht werden und hat folgenden grundsätzlichen Aufbau:

```
set<Datentyp, Ordnungsobjekt<Datentyp> > Variablenname;
typedef set<Datentyp, Ordnungsobjekt<Datentyp> > Datentypname;
```

**ACHTUNG:** Das Leerzeichen zwischen den beiden schließenden, spitzen Klammern „> >“ ist zwingend notwendig, da der Compiler zusammenhängende spitze Klammern „>>“ versucht als den entsprechenden Shift-Operator zu verwenden.

### 6.1 Die set Constructoren

Die folgende Tabelle faßt die Constructoren der Containerklasse **set** zusammen. In den nachstehenden Abschnitten werden die Constructoren einzeln behandelt und mit Beispielen erläutert.

<b>Constructoren der Containerklasse SET</b>	
<i>Constructor</i>	<i>Bedeutung</i>
set< K, C > VarName;	Standard-Constructor, erzeugt ein

<sup>1</sup> Die übliche Form der Realisierung ist ein binärer Baum, der in jedem Knoten der Baumstruktur genau einen Key speichert.

<b>Constructoren der Containerklasse SET</b>	
<i>Constructor</i>	<i>Bedeutung</i>
	neues, leeres <b>set</b> -Objekt
<code>set&lt; K, C &gt; VarName (const Compare&amp; newcompare);</code>	Dieser Constructor erzeugt einen neuen, leeren <b>set</b> , welcher die enthaltenen Keys intern anhand der Vergleichsfunktion <code>newcompare</code> ordnet, anstelle der ursprünglich definierten <code>compare</code> -Funktion <b>C</b> .
<code>set&lt; K, C &gt; VarName (K* pFirst, K* pLast);</code>	Dieser Constructor erzeugt einen neuen <b>set</b> anhand eines bereits bestehenden <b>set</b> -Objektes. Die Pointer <b>pFirst</b> und <b>pLast</b> bezeichnen dabei Anfang- und Endwert eines Bereiches. Das Element, auf welches der Zeiger <b>pLast</b> verweist gehört <b>nicht</b> mehr zum kopierten Bereich.
<code>set&lt;K, C&gt; VarName (K* pFirst, K* pLast, const Compare&amp; newcompare);</code>	Dieser Constructor erzeugt einen neuen <b>set</b> anhand eines bereits bestehenden <b>set</b> -Objektes. Die Pointer <b>pFirst</b> und <b>pLast</b> bezeichnen dabei Anfang- und Endwert eines Bereiches. Das Element, auf welches der Zeiger <b>pLast</b> verweist gehört <b>nicht</b> mehr zum kopierten Bereich. Der <b>set</b> verwendet die übergebene <code>compare</code> -Funktion zur internen Ordnung der Keys.
<code>set&lt; K, C &gt; VarName (const set &lt;Key, Compare&gt;&amp; s);</code>	Dieser Constructor erzeugt einen neuen <b>set</b> anhand eines bereits bestehenden <b>set</b> -Objektes. Das neue <b>set</b> -Objekt ist eine vollständige Kopie des übergebenen <b>set</b> -Objektes <b>s</b> .

Tabelle 6-1– Constructoren der Containerklasse set

### 6.1.1 Der set Standard-Constructor

Der **set** Standard-Constructor, erzeugt ein neues, leeres **set**-Objekt vom Typ **K, C**<sup>2</sup>.

Syntax:  
`set<K, C> Variablenname;`

Das folgende Beispielprogramm zeigt den Einsatz des Standard-Constructors innerhalb eines Programms:



```
//=====
// PROGRAMM: SET_BSP_01
//=====
```

<sup>2</sup> **K** ist der Platzhalter für den im **set** verwalteten Schlüssel (Key) und **C** ist der Platzhalter für die im **set** verwendeten Ordnungsklasse.

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

typedef set<char, less<char> > MySetType;

void main (void)
{
    MySetType aSet;
    set<int, less<int> > aSet2;
}
```

### 6.1.2 Der set Constructor mit alternativer Ordnungsfunktion

Dieser Constructor erzeugt einen neuen **set** mit der Ordnungsfunktion **C2** anstelle der vordefinierten Ordnungsfunktion **C**.

Syntax:

```
set<K, C> Variablenname (Compare& C2);
```

Das folgende Beispielprogramm zeigt den Einsatz des Constructors innerhalb eines Programms:

```
//=====
// PROGRAMM: SET_BSP_02
//=====

#include <set>
using namespace std;

typedef set<char, less<char> > MySetType;

void main (void)
{
    MySetType aSet(greater<char>);
}
```



### 6.1.3 Der set Constructor mit Bereichsangabe

Dieser Constructor erzeugt einen neuen **set** anhand eines bereits bestehenden **set**-Objektes. Die Pointer **pFirst** und **pLast** bezeichnen dabei Anfang- und Endwert eines Bereiches. Das Element, auf welches der Iterator **pLast** verweist gehört **nicht** mehr zum kopierten Bereich – d.h. **pLast** verweist auf das erste Element hinter dem zu kopierenden Bereich.

Für jedes der im Bereich befindlichen Elemente wird einmal der Copy-Constructor von **K** mit dem korrespondierenden Element des übergebenen Bereiches aufgerufen. Zur Sortierung der Elemente im **set** wird die Ordnungsfunktion **C** herangezogen.

Syntax:

```
set<K, C> Variablenname (K* pFirst, K* pLast);
```

Anhand der übergebenen Zeiger kann der Constructor die benötigte Anzahl an Elementen ermitteln und eine entsprechend großen Block reservieren.

### 6.1.4 Der set Constructor mit Bereichsangabe und alternativer Ordnungsfunktion

Dieser Constructor erzeugt einen neuen **set** anhand eines bereits bestehenden **set**-Objektes. Die Pointer **pFirst** und **pLast** bezeichnen dabei Anfang- und Endwert eines Bereiches. Das Element, auf welches der Iterator **pLast** verweist gehört *nicht* mehr zum kopierten Bereich – d.h. **pLast** verweist auf das erste Element hinter dem zu kopierenden Bereich. Der Constructor erzeugt einen neuen **set**, der jedoch die Ordnungsfunktion **C2** anstelle der vordefinierten Ordnungsfunktion **C** nutzt, d.h. der kopierte Bereich ist in der **set**-Kopie anders sortiert als im Original.

Für jedes der im Bereich befindlichen Elemente wird einmal der Copy-Constructor von **K** mit dem korrespondierenden Element des übergebenen Bereiches aufgerufen.

Syntax:

```
set<K, C> Variablenname (K* pFirst, K* pLast,
                        Compare& C2);
```

Anhand der übergebenen Zeiger kann der Constructor die benötigte Anzahl an Elementen ermitteln und eine entsprechend großen Block reservieren.

### 6.1.5 Der set Copy-Constructor

Dieser Constructor erzeugt einen neuen **set** anhand eines bereits bestehenden **set**-Objektes. Das neue **set**-Objekt ist eine vollständige Kopie des übergebenen Objektes **S**.

Syntax:

```
set<K, C> Variablenname (const set<K, C>& S);
```

Das folgende Beispielprogramm zeigt, wie der Copy-Constructor verwendet werden kann:



```
//=====
// PROGRAMM: SET_BSP_03
//=====

#include <set>
#include <iostream.h>
using namespace std;

void main (void)
{
    set<char, less<char> > aMySet;

    aMySet.insert('A');
    aMySet.insert('B');
    aMySet.insert('C');
    aMySet.insert('D');

    // den Set kopieren
    set<char, less<char> > aMySet2 (aMySet);
}
```

## 6.2 Die set Methoden

Die folgende Tabelle faßt die Methoden der **set** Containerklasse zusammen. In den nachstehenden Abschnitten werden die Methoden einzeln behandelt und mit Beispielen erläutert.

<b>Methoden der Containerklasse SET</b>	
<i>Methode</i>	<i>Bedeutung</i>
begin	Rückgabe eines <b>iterator</b> , der auf den ersten im <b>set</b> enthaltenen Key zeigt
count	Gibt die Anzahl aller Elemente (Keys) zurück, die mit dem übergebenen Key übereinstimmen. Da <b>set</b> nur 1:1 Beziehungen speichern kann, ist das Ergebnis entweder Eins oder Null
empty	Gibt den Wert <b>true</b> zurück, wenn der <b>set</b> keine Datenelemente enthält
end	Rückgabe eines <b>iterator</b> , der hinter den letzten Key in im <b>set</b> zeigt
erase	Löscht einen Key aus dem <b>set</b>
find	Gibt, falls ein Key zum übergebenen Key existiert, einen Iterator auf dieses Element zurück, sonst einen Iterator der auf <b>end()</b> zeigt
insert	Fügt die Kopie eines Keys oder eines Key-Bereiches in den <b>set</b> ein, falls im <b>set</b> noch kein entsprechender Key existierte.
key_comp	Gibt das Vergleichsobjekt zurück, welches zum Vergleich der Schlüssel verwendet wird.
lower_bound	Gibt einen Iterator zurück, der auf die erste Position im <b>set</b> zeigt, an der ein Key eingefügt werden kann, ohne daß die Ordnung der Klasse <b>C</b> verletzt wird. Der Iterator zeigt auf <b>end()</b> wenn eine solche Position nicht gefunden werden konnte.
max_size	Gibt die maximale Anzahl an Datenelementen zurück, welche der <b>set</b> enthalten kann
rbegin	Rückgabe eines <b>reverse_iterator</b> , der auf den letzten im <b>set</b> enthaltenen Key zeigt
rend	Rückgabe eines <b>reverse_iterator</b> , der vor den ersten im <b>set</b> enthaltenen Key zeigt
size	Gibt die Anzahl von Keys zurück, die aktuell im <b>set</b> enthalten sind
swap	Tauscht den Inhalt zweier <b>set</b> -Objekte aus
upper_bound	Gibt einen Iterator zurück, der auf die letzte Position im <b>set</b> zeigt, an der ein Key eingefügt werden kann, ohne daß die Ordnung der Klasse <b>C</b> verletzt wird. Der Iterator zeigt auf <b>end()</b> wenn eine solche Position nicht gefunden werden konnte.
value_comp	Gibt das Vergleichsobjekt zurück, welches zum Vergleich der Schlüssels verwendet wird.

Tabelle 6-1– Methoden der Containerklasse set

### 6.2.1 Die set Methode begin

Rückgabe eines **iterator**, der auf das erste im **set** enthaltene Element zeigt.

Syntax:

```
iterator begin ();
```

Die wohl häufigste Anwendung der Methode **begin()** liegt in der Verarbeitung von Daten durch Schleifen.

Der Aufruf von **begin()** verändert den Inhalt des **set** nicht.



```
//=====
// PROGRAMM: SET_BSP_04
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;

void main (void)
{
    set<int, less<int> > aSet;
    set<int, less<int> >::const_iterator aI;

    aSet.insert(3);
    aSet.insert(8);
    aSet.insert(12);

    for (aI=aSet.begin(); aI!=aSet.end(); aI++)
    {
        cout << "Wert: " << *aI << endl;
    }
}
```

### 6.2.2 Die set Methode count

Rückgabe der Anzahl der Keys, die mit einem übergebenen Schlüssel übereinstimmen.

Syntax:

```
size_type count (const K& key) const;
```

Die wichtigste Bedeutung der Methode **count()** liegt beim einfachen **set** darin festzustellen, ob ein bestimmter Schlüssel existiert oder nicht. Dies kann zwar auch über **find()** festgestellt werden, da **find()** aber einen Iterator zurückgibt, muß dieser erst wieder mit einem weiteren Vergleich überprüft werden.



```
//=====
// PROGRAMM: SET_BSP_05
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;

void main (void)
```

```

{
    set<int, less<int> > aSet;
    set<int, less<int> >::const_iterator aI;

    aSet.insert(3);
    aSet.insert(8);
    aSet.insert(12);

    if (aSet.count(7))
    {
        cout << "Key zu 7 existiert" << endl;
    }
    else
    {
        cout << "Key zu 7 existiert nicht" << endl;
    }

    if (aSet.count(8))
    {
        cout << "Key zu 8 existiert" << endl;
    }
    else
    {
        cout << "Key zu 8 existiert nicht" << endl;
    }
}

```

### 6.2.3 Die set Methode empty

Die Methode **empty()** prüft, ob der **set** Datenelemente enthält oder nicht. Sind Datenelemente enthalten (dies entspricht einem Rückgabewert von **size()** größer als Null), so wird der Wert **false** zurückgegeben, sind keine Datenelemente vorhanden ist der Rückgabewert **true**.

Der Aufruf von **empty()** verändert den Inhalt des **set**-Objektes nicht.

Syntax:

```
bool empty () const;
```

```

//=====
// PROGRAMM: SET_BSP_06
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;

void main (void)
{
    set<int, less<int> > aSet;

    if (aSet.empty())
    {
        cout << "Set ist leer" << endl;
    }
    else
    {
        cout << "Set enthält Keys" << endl;
    }

    aSet.insert(3);

```



```

aSet.insert(6);
aSet.insert(7);

if (aSet.empty())
{
    cout << "Set ist leer" << endl;
}
else
{
    cout << "Set enthält Keys" << endl;
}
}

```

### 6.2.4 Die set Methode end

Rückgabe eines **iterator**, der *hinter* den letzten im **set** enthaltenen Key zeigt.

Syntax:

```
iterator end ();
```

Die wohl häufigste Anwendung der Methode **end()** liegt in der Verarbeitung von Daten durch Schleifen.

Es ist zu beachten, daß **end()** hinter den letzten gültigen Key zeigt, alle Schleifen müssen also auf „ungleich“ (Operator !=) oder „echt kleiner als“ prüfen (Operator <) und nicht auf „kleiner oder gleich“ (Operator <=). Zeigt ein **iterator** auf **end()** führt jeder dereferenzierende Zugriff zu einer Exception vom Typ **out\_of\_range**. Wird die Exception nicht abgefangen (wie zweiten Beispiel), bricht das Programm unkontrolliert ab (Absturz).

Der Aufruf von **end()** verändert den Inhalt des **set**-Objektes nicht.



```

//=====
// PROGRAMM: SET_BSP_07
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;

void main (void)
{
    set<int, less<int> > aSet;
    set<int, less<int> >::const_iterator aI;

    aSet.insert(4);
    aSet.insert(5);
    aSet.insert(6);

    for (aI=aSet.begin(); aI!=aSet.end(); aI++)
    {
        cout << "Wert: " << *aI << endl;
    }
}

```

Ein Beispiel, welches die Auslösung einer Exception aufgrund eines illegalen Zugriffs auf **end()** zeigt, ist im Abschnitt zur Methode **end()** bei der Containerklasse **vector** aufgeführt.

### 6.2.5 Die set Methode erase

Löscht ein oder mehrere Datenelemente an der angegebenen Position aus dem **set**.

Syntax:

```
void erase(iterator toDel);  
void erase(iterator First, iterator Last);  
size_type erase (const K& key);
```

Die Methode **erase()** kann verwendet werden, um einzelne Keys oder ganze Bereiche aus dem **set** zu löschen.

*Im Gegensatz zum vector und dem deque werden durch das Löschen im set nur die Iteratoren, Verweise und Referenzen auf Datensätze im gelöschten Bereich ungültig.*

*Da es sich beim set nicht um ein Array handelt (welches als geschlossener Block gespeichert werden muß), gibt es keine Notwendigkeit, das set-Objekt im Speicher zu verschieben.*

*Iteratoren, Verweise und Referenzen, die nicht auf einen durch die Methode gelöschten Bereich zeigen, bleiben daher gültig.*

Die erste Form der Methode sorgt dafür, daß ein einzelner Key aus dem **set** gelöscht und der Destructor des zu löschenden Keys aufgerufen wird.

Bei der zweiten Syntaxform wird ein zu löschender Bereich angegeben, für jeden zu löschenden Key wird der Destructor aufgerufen. Die Angabe erfolgt durch zwei Iteratoren, die auf das erste und das letzte zu löschende Element zeigen.

Es ist besonders bei den Bereichsangaben zu beachten, daß nicht versehentlich eine **out\_of\_range** Exception erzeugt wird.

Ein Beispiel, welches die Auslösung einer Exception aufgrund eines illegalen Zugriffs durch **erase()** zeigt, ist im Abschnitt zur Methode **erase()** bei der Containerklasse **vector** aufgeführt.

Die dritte Form der **erase**-Methode löscht alle Keys, die mit einem angegebenen Key übereinstimmen. Da es sich beim **set** um eine 1:1-Beziehung handelt ist die Anzahl der gelöschten Keys dementsprechend entweder Eins oder Null.

### 6.2.6 Die set Methode find

Die Methode **find()** gibt einen Iterator auf einen im **set** gespeicherten Key zurück. Ist ein entsprechender Key im **set** nicht enthalten, so wird ein Iterator auf **end()** zurückgegeben, der hinter den letzten gültigen Eintrag zeigt.

Der Aufruf von **find()** selbst verändert den Inhalt des **set**-Objektes nicht.

Syntax:

```
iterator find (const K& key) const;
```

Das nachfolgende Beispielprogramm zeigt den Einsatz der **find()**-Methode in einem Programm:



```
//=====
// PROGRAMM: SET_BSP_08
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;

void main (void)
{
    set<int, less<int> > aSet;
    set<int, less<int> >::iterator aI;

    aSet.insert(1);
    aSet.insert(3);
    aSet.insert(4);

    aI = aSet.find(7);
    if (aI != aSet.end())
    {
        cout << "Key zu 7 existiert" << endl;
    }
    else
    {
        cout << "Key zu 7 existiert nicht" << endl;
    }

    aI = aSet.find(4);
    if (aI != aSet.end())
    {
        cout << "Key zu 4 existiert" << endl;
    }
    else
    {
        cout << "Key zu 4 existiert nicht" << endl;
    }
}
}
```

### 6.2.7 Die set Methode insert

Fügt einen oder mehrere Keys vor der angegebenen Position im **set** ein.

Syntax:

```
pair<iterator, bool> insert (const K& key);
void                insert (const K* first, const K* last);
iterator insert (iterator pos, const K& key);
```

Die Methode **insert()** kann verwendet werden um einzelne Keys oder ganze Bereiche in einen **set** einzufügen.

Da ein **set** als Baumstruktur aufgebaut ist, muß (anders als bei **vector** oder **deque**) beim Einfügen von Keys nichts umkopiert werden, was die Methode **insert()** für **set**-Objekte sehr effizient macht. Der Zeitaufwand für das Einfügen in einem **set**-Objekt ist daher immer konstant.

*Im Gegensatz zum `vector` und dem `deque` werden durch das Einfügen keine Iteratoren, Verweise oder Referenzen ungültig.*

Die erste Form der Methode sorgt dafür, daß ein einzelner Key in den `set` eingefügt wird. Eingefügt wird die Kopie des Keys, falls im `set` noch kein Key zu diesem Schlüssel existiert hat. Rückgabewert der Methode ist ein `pair`, dessen erstes Element der Iterator auf das eingefügte Element im `set` ist und dessen zweites Element `true` (Key existierte noch nicht und wurde eingefügt) oder `false` (Key existierte bereits und wurde daher nicht eingefügt) ist.

Mit der zweiten Form der `insert()`-Methode können Bereiche aus einem anderen `set` in den `set` kopiert werden. Der Bereich wird durch Anfangs- und Endadresse festgelegt. Es sollte darauf geachtet werden, daß die Endadresse nicht vor der Anfangsadresse liegen darf und daß beide Adressen zum gleichen `set`-Objekt gehören.

Die dritte `insert()`-Form fügt, wie die erste, einen Schlüssel ein, sofern noch nicht vorhanden. Beginnt jedoch mit der Suche nach der richtigen Position zum Einfügen an der angegebenen Position des Iterators `pos`.

```
//=====
// PROGRAMM: SET_BSP_09
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;

void main (void)
{
    typedef set<int, less<int> > mysettype;
    mysettype aSet;

    aSet.insert(4);
    aSet.insert(5);
}
```



### 6.2.8 Die set Methode `key_comp`

Gibt das Vergleichsobjekt zurück, welches zum Vergleich der Schlüssel herangezogen wird.

Syntax:

```
Compare key_comp () const;
```

### 6.2.9 Die set Methode `lower_bound`

Die Methode gibt einen Iterator zurück, welcher auf die erste Position zeigt, an der zu einem gegebenen Schlüssel eingefügt werden kann, ohne daß die durch das Compare-Objekt vorgegebenen Ordnungskriterien verletzt werden. Der Iterator zeigt auf die Position `end()`, wenn keine gültige Position gefunden werden kann.

Syntax:

```
iterator lower_bound (const K& key);
```

### 6.2.10 Die set Methode max\_size

Die Methode **max\_size()** gibt die maximale Anzahl an Keys zurück, die in einem **set** gespeichert werden kann. Der Wert von **max\_size()** ist u.a. vom Typ des Schlüssels abhängig.

Syntax:

```
size_type max_size () const;
```

Der Wert ist zudem abhängig von der Implementation und vom verwendeten Betriebssystem. Bei modernen Compilern und entsprechender Einstellung kann man davon ausgehen, dass 32-Bit Adressen verwendet werden, also theoretisch 4 Gigabytes angesprochen werden können.

Da **set**-Objekte nicht als Block gespeichert sein müssen, wird die Anzahl der Datenelemente sonst nur durch den freien RAM-Speicher begrenzt, die Größe der zu verwaltenden Datenelemente, sowie die Segmentierung (Zersplitterung) des Speichers.

Der Aufruf der Methode verändert den Inhalt des **set**-Objektes nicht.



```
//=====
// PROGRAMM: SET_BSP_10
//=====

#include <iostream.h>
#include <set>
using namespace std;

void main (void)
{
    set<char, less<char> > aSet;

    cout << "Set kann " << aSet.max_size ()
         << " Datenelemente aufnehmen";
}

```

### 6.2.11 Die set Methode rbegin

Rückgabe eines **reverse\_iterator** oder **const\_reverse\_iterator**, der auf den letzten im **set** enthaltenen Key zeigt.

Syntax:

```
reverse_iterator rbegin ();
```

Die wohl häufigste Anwendung der Methode **rbegin()** liegt in der Verarbeitung von Daten durch rückwärts gerichtete Schleifen.

Der Aufruf von **rbegin()** verändert den Inhalt des **set** nicht.



```
//=====
// PROGRAMM: SET_BSP_11
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>

```

```

using namespace std;

void main (void)
{
    set<int, less<int> > aSet;
    set<int, less<int> >::reverse_iterator aI;

    aSet.insert(1);
    aSet.insert(3);
    aSet.insert(4);

    for (aI=aSet.rbegin(); aI!=aSet.rend(); aI++)
    {
        cout << "Wert: " << *aI << endl;
    }
}

```

### 6.2.12 Die set Methode rend

Rückgabe eines **reverse\_iterator** oder **const\_reverse\_iterator**, der *vor* den ersten im **set** enthaltene Key zeigt.

Syntax:

```
reverse_iterator rend ();
```

Die wohl häufigste Anwendung der Methode **rend()** liegt in der Verarbeitung von Daten durch rückwärts gerichtete Schleifen.

Es ist zu beachten, daß **rend()** vor den ersten gültigen Key zeigt, alle Schleifen müssen also auf „ungleich“ prüfen (Operator !=). Zeigt ein **iterator** auf **rend()** führt jeder dereferenzierende Zugriff zu einer Exception vom Typ **out\_of\_range**. Wird die Exception nicht wie zweiten Beispiel abgefangen bricht das Programm unkontrolliert ab (Absturz).

Der Aufruf von **rend()** verändert den Inhalt des **set** nicht.

Ein Anwendungsbeispiel ist unter **rbegin()** zu finden. Ein Beispiel, welches die Auslösung einer Exception aufgrund eines illegalen Zugriffs auf **rend()** zeigt, ist im Abschnitt über die Methode **rend()** der Containerklasse **vector** zu finden.

### 6.2.13 Die set Methode size

Gibt die Anzahl von Keys zurück, die aktuell im **set** enthalten sind.

Syntax:

```
size_type size () const;
```

Der Aufruf von **size()** verändert den Inhalt des **set** nicht.

```

//=====
// PROGRAMM: SET_BSP_12
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;

void main (void)

```



```

{
    set<int, less<int> > aSet;
    set<int, less<int> >::const_reverse_iterator aI;

    aSet.insert(1);
    aSet.insert(3);
    aSet.insert(4);

    cout << "Anzahl aktuell: " << aSet.size() << endl;
}

```

### 6.2.14 Die set Methode swap

Tauscht die Inhalte zweier **set**-Objekte aus. Die **set**-Objekte müssen von gleichem Typ sein, also gleiche Key- und Compare-Typen enthalten.

Syntax:

```
void swap (set<K, C>& s);
```

*Es ist zu beachten, daß somit alle Iteratoren, Verweise und Referenzen ungültig werden, da es sich um Pointer handelt und die Datensätze durch den Tausch im Speicher verschoben werden.*

*Alle Iteratoren, Verweise und Referenzen müssen neu ermittelt werden.*



```

//=====
// PROGRAMM: SET_BSP_13
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;

void main (void)
{
    typedef set<int, less<int> > mysettype;
    mysettype aSet1;
    mysettype aSet2;

    aSet1.insert(1);
    aSet1.insert(3);
    aSet1.insert(4);

    aSet2.insert(11);
    aSet2.insert(33);
    aSet2.insert(44);

    aSet1.swap (aSet2);
}

```

### 6.2.15 Die set Methode upper\_bound

Die Methode gibt einen Iterator zurück, welcher auf die letzte Position zeigt, an der zu einem gegebenen Schlüssel eingefügt werden kann, ohne daß die durch das Compare-Objekt vorgegebenen Ordnungskriterien verletzt werden.

Der Iterator zeigt auf die Position **end()**, wenn keine gültige Position gefunden werden kann.

Syntax:

```
iterator upper_bound (const K& key);
```

### 6.2.16 Die set Methode value\_comp

Gibt das Vergleichsobjekt zurück, welches zum Vergleich der Schlüssel herangezogen wird.

Syntax:

```
set_compare<K,C> value_comp () const;
```

## 6.3 Die set Operatoren

Mit den folgenden Operatoren kann auf eine Containerklasse vom Typ **set** zugegriffen werden:

<b>Operatoren der Containerklasse SET</b>	
<i>Operator</i>	<i>Bedeutung</i>
=	Zuweisung zwischen zwei <b>set</b> -Objekten gleichen Typs
==	Vergleich zwischen zwei <b>set</b> -Objekten gleichen Typs
!=	Vergleich zwischen zwei <b>set</b> -Objekten gleichen Typs
<	Lexikographischer Vergleich zwischen zwei <b>set</b> -Objekten gleichen Typs
>	Lexikographischer Vergleich zwischen zwei <b>set</b> -Objekten gleichen Typs
>=	Lexikographischer Vergleich zwischen zwei <b>set</b> -Objekten gleichen Typs
<=	Lexikographischer Vergleich zwischen zwei <b>set</b> -Objekten gleichen Typs

Tabelle 6-1– Operatoren der Containerklasse set

### 6.3.1 Der set Operator =

Der Operator ersetzt den Inhalt eines **set**-Objektes durch den Inhalt des zugewiesenen **set**-Objektes gleichen Typs.

Syntax:

```
set<K,C>& operator= (const set<K,C>& s);
```

*Durch die Zuweisung werden alle Iteratoren, Verweise und Referenzen auf dieser Set ungültig. Alle Iteratoren, Verweise und Referenzen müssen neu ermittelt werden.*

```
//=====
// PROGRAMM: SET_BSP_14
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
```



```

using namespace std;

void main (void)
{
    typedef set<int, less<int> > mysettype;
    mysettype aSet1;
    mysettype aSet2;
    mysettype::iterator aI;

    aSet1.insert(1);
    aSet1.insert(2);
    aSet1.insert(3);

    aSet2 = aSet1;

    for (aI=aSet1.begin(); aI!=aSet1.end(); aI++)
    {
        cout << "aSet1 Wert: " << *aI << endl;
    }

    for (aI=aSet2.begin(); aI!=aSet2.end(); aI++)
    {
        cout << "aSet2 Wert: " << *aI << endl;
    }
}

```

### 6.3.2 Der set Operator ==

Der Operator prüft, ob die Inhalte des linken und rechten Operanden (**set**-Objekte gleichen Typs) übereinstimmen. Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn beide **set**-Objekte die gleichen Keys in der gleichen Reihenfolge enthalten.

Syntax:

```
bool operator== (const set<K,C>& s) const;
```



```

//=====
// PROGRAMM: SET_BSP_15
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;

void main (void)
{
    typedef set<int, less<int> > mysettype;
    mysettype aSet1;
    mysettype aSet2;
    mysettype::iterator aI;

    aSet1.insert(1);

    if (aSet1 == aSet2)
    {
        cout << "Die set-Objekte sind gleich" << endl;
    }
    else

```

```

{
    cout << "Die Set-Objekte sind nicht gleich" << endl;
}

aSet2.insert(1);

if (aSet1 == aSet2)
{
    cout << "Die Set-Objekte sind gleich" << endl;
}
else
{
    cout << "Die Set-Objekte sind nicht gleich" << endl;
}
}

```

### 6.3.3 Der set Operator !=

Der Operator prüft, ob die Inhalte des linken und rechten Operanden (**set**-Objekte gleichen Typs) verschieden sind. Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn beide **set**-Objekte unterschiedliche Key oder gleiche Keys in unterschiedlicher Reihenfolge enthalten.

Syntax:

```
bool operator!= (const set<K,C>& s) const;
```

```

//=====
// PROGRAMM: SET_BSP_16
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;

void main (void)
{
    typedef set<int, less<int> > mysettype;
    mysettype aSet1;
    mysettype aSet2;
    mysettype::iterator aI;

    aSet1.insert(1);

    if (aSet1 != aSet2)
    {
        cout << "Die set-Objekte sind ungleich" << endl;
    }
    else
    {
        cout << "Die Set-Objekte sind nicht ungleich" << endl;
    }

    aSet2.insert(1);

    if (aSet1 != aSet2)
    {
        cout << "Die Set-Objekte sind ungleich" << endl;
    }
}

```



```

else
{
    cout << "Die Set-Objekte sind nicht ungleich" << endl;
}
}

```

### 6.3.4 Der set Operator <

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch kleiner ist, als der des rechten Operanden (**set**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „<“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch kleiner ist als der rechte Operand.

Syntax:

```
bool operator< (const set<K,C>& s) const;
```



```

//=====
// PROGRAMM: SET_BSP_17
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;

void main (void)
{
    set<int, less<int> > aSet1;
    set<int, less<int> > aSet2;

    aSet1.insert(1);
    aSet2.insert(2);

    if (aSet1 < aSet2)
    {
        cout << "aSet1 ist kleiner als aSet2" << endl;
    }
    else
    {
        cout << " aSet1 ist nicht kleiner als aSet2" << endl;
    }
}

```

### 6.3.5 Der set Operator >

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch größer ist, als der des rechten Operanden (**set**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „>“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch größer ist als der rechte Operand.

Syntax:

```
bool operator> (const set<K,C>& s) const;
```

```
//=====
// PROGRAMM: SET_BSP_18
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;

void main (void)
{
    set<int, less<int> > aSet1;
    set<int, less<int> > aSet2;

    aSet1.insert(1);
    aSet2.insert(2);

    if (aSet1 > aSet2)
    {
        cout << "aSet1 ist größer als aSet2" << endl;
    }
    else
    {
        cout << " aSet1 ist nicht größer als aSet2" << endl;
    }
}

```



### 6.3.6 Der set Operator <=

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch kleiner oder gleich mit dem rechten Operanden ist (**set**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „<=“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch kleiner oder gleich mit dem rechten Operanden ist.

Syntax:

```
bool operator<= (const set<K,C>& s) const;
```

```
//=====
// PROGRAMM: SET_BSP_19
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;

void main (void)
{
    set<int, less<int> > aSet1;
    set<int, less<int> > aSet2;

    aSet1.insert(1);
    aSet2.insert(2);

    if (aSet1 <= aSet2)
    {
        cout << "aSet1 ist kleiner gleich aSet2" << endl;
    }
}

```



```

else
{
    cout << " aSet1 ist nicht kleiner gleich aSet2" << endl;
}
}

```

### 6.3.7 Der set Operator >=

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch größer oder gleich mit dem rechten Operanden ist (**set**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „>=“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch größer oder gleich mit dem rechten Operanden ist.

Syntax:

```
bool operator>= (const set<K,C>& s) const;
```



```

//=====
// PROGRAMM: SET_BSP_20
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;

void main (void)
{
    set<int, less<int> > aSet1;
    set<int, less<int> > aSet2;

    aSet1.insert(1);
    aSet2.insert(2);

    if (aSet1 >= aSet2)
    {
        cout << "aSet1 ist größer gleich aSet2" << endl;
    }
    else
    {
        cout << " aSet1 ist nicht größer gleich aSet2" << endl;
    }
}

```

## 7. Die Containerklasse multiset

Auch die Containerklasse **multiset** realisiert einen assoziativen Behälter (Hashtable), welcher Schlüssel beinhaltet (Keys). Wie **set** ist auch der **multiset** durch eine vom Benutzer definierte Vergleichsfunktion (Compareclass) sortiert. Die Containerklasse setzt jedoch nicht voraus, daß jeder Schlüssel nur ein einziges mal (unique) in vorkommen kann.

Die interne Struktur eines **multiset** (Hashtable) ermöglicht es, einen Key sehr effizient zu ermitteln<sup>1</sup>.

Eine häufige Form der Verwendung von **multiset** ist z.B. die Markierung von durchlaufenen Programmknoten oder die Ordnung von Datenlisten, bei denen jede Einstellung bzw. jedes Datum mehrfach vorkommen kann.

Ein **multiset**-Objekt hat intern einen anderen Aufbau (Baumstruktur) als ein **vector**-, **deque**- oder **list**-Objekt, um eine effiziente Suche zu ermöglichen.

Die notwendige Speicherverwaltung wird automatisch vorgenommen und ist so effizient wie möglich realisiert. Wie für **list**- und **set**-Objekte wird auch für einen **multiset** kein geschlossener Speicherbereich benötigt. Um ein **multiset**-Objekt zu erweitern ist daher kein Umkopieren der gesamten Containerklasse notwendig, da lediglich die Verkettung innerhalb der Baumstruktur geändert werden muß.

Ein wahlfreier Zugriff auf **multiset**-Objekte ist nicht möglich, der **operator[]** (Zugriff über Indexklammer) ist daher nicht definiert.

In einem Punkt weicht **multiset**, wie auch der **set**, von den Containerklassen **vector**, **deque** und **list** ab. Während diese lediglich eine Klassenangabe erwarten, um die Containerklasse zu definieren, also lediglich die zu verwaltende Klasse benötigen, braucht ein **multiset**-Objekt insgesamt zwei Klassenangaben. Die erste erforderliche Angabe verweist auf die Klasse, welche den Key repräsentiert, die zweite definiert die Vergleichsklasse für die Sortierung:

Die Deklaration eines **multiset**-Objektyps kann durch eine **typedef**-Anweisung weiter vereinfacht werden und hat folgenden grundsätzlichen Aufbau:

```
multiset<Datentyp, Ordnungsobjekt<Datentyp> > Variablenname;
typedef multiset<Datentyp, Ordnungsobjekt<Datentyp> > Datentypname;
```

**ACHTUNG:** Das Leerzeichen zwischen den beiden schließenden, spitzen Klammern „>“ ist zwingend notwendig, da der Compiler zusammenhängende spitze Klammern „>>“ versucht als den entsprechenden Shift-Operator zu verwenden.

### 7.1 Die multiset Constructoren

Die folgende Tabelle faßt die Constructoren der Containerklasse **multiset** zusammen. In den nachstehenden Abschnitten werden die Constructoren einzeln behandelt und mit Beispielen erläutert.

<b>Constructoren der Containerklasse MULTiset</b>	
<i>Constructor</i>	<i>Bedeutung</i>

<sup>1</sup> Die übliche Form der Realisierung ist ein binärer Baum, der in jedem Knoten der Baumstruktur genau einen Key speichert.

<b>Constructoren der Containerklasse <i>MULTISET</i></b>	
<i>Constructor</i>	<i>Bedeutung</i>
<code>multiset&lt; K, C &gt; VarName;</code>	Standard-Constructor, erzeugt ein neues, leeres <b>multiset</b> -Objekt
<code>multiset&lt; K, C &gt; VarName (const   Compare&amp; newcompare);</code>	Dieser Constructor erzeugt einen neuen, leeren <b>multiset</b> , welcher die enthaltenen Keys intern anhand der Vergleichsfunktion <code>newcompare</code> ordnet, anstelle der ursprünglich definierten <code>compare</code> -Funktion <b>C</b> .
<code>multiset&lt; K, C &gt; VarName (K* pFirst, K* pLast);</code>	Dieser Constructor erzeugt einen neuen <b>multiset</b> anhand eines bereits bestehenden <b>multiset</b> -Objektes. Die Pointer <b>pFirst</b> und <b>pLast</b> bezeichnen dabei Anfang- und Endwert eines Bereiches. Das Element, auf welches der Zeiger <b>pLast</b> verweist gehört <b>nicht</b> mehr zum kopierten Bereich.
<code>multiset&lt;K, C&gt; VarName (K* pFirst, K* pLast, const   Compare&amp; newcompare);</code>	Dieser Constructor erzeugt einen neuen <b>multiset</b> anhand eines bereits bestehenden <b>multiset</b> -Objektes. Die Pointer <b>pFirst</b> und <b>pLast</b> bezeichnen dabei Anfang- und Endwert eines Bereiches. Das Element, auf welches der Zeiger <b>pLast</b> verweist gehört <b>nicht</b> mehr zum kopierten Bereich. Der <b>multiset</b> verwendet die übergebene <code>compare</code> -Funktion zur internen Ordnung der Keys.
<code>multiset&lt; K, C &gt; VarName (const multiset &lt;Key,   Compare&gt;&amp; s);</code>	Dieser Constructor erzeugt einen neuen <b>multiset</b> anhand eines bereits bestehenden <b>multiset</b> -Objektes. Das neue <b>multiset</b> -Objekt ist eine vollständige Kopie des übergebenen <b>multiset</b> -Objektes <b>s</b> .

 Tabelle 7-1– Constructoren der Containerklasse *multiset*

### 7.1.1 Der **multiset** Standard-Constructor

Der **multiset** Standard-Constructor, erzeugt ein neues, leeres **multiset**-Objekt vom Typ **K, C**<sup>2</sup>.

Syntax:

```
multiset<K, C> Variablenname;
```

Das folgende Beispielprogramm zeigt den Einsatz des Standard-Constructors innerhalb eines Programms:

<sup>2</sup> **K** ist der Platzhalter für den im **set** verwalteten Schlüssel (Key) und **C** ist der Platzhalter für die im **set** verwendeten Ordnungsklasse.

```
//=====
// PROGRAMM: MULTISSET_BSP_01
//=====

#include <set>
using namespace std;

typedef multiset<char, less<char> > MySetType;

void main (void)
{
    MySetType aSet;
    multiset<int, less<int> > aSet2;
}
```



### 7.1.2 Der multiset Constructor mit alternativer Ordnungsfunktion

Dieser Constructor erzeugt einen neuen **multiset** mit der Ordnungsfunktion **C2** anstelle der vordefinierten Ordnungsfunktion **C**.

Syntax:

```
multiset<K, C> Variablenname (Compare& C2);
```

Das folgende Beispielprogramm zeigt den Einsatz des Constructors innerhalb eines Programms:

```
//=====
// PROGRAMM: MULTISSET_BSP_02
//=====

#include <set>
using namespace std;

typedef multiset<char, less<char> > MySetType;

void main (void)
{
    MySetType aSet(greater<char>);
}
```



### 7.1.3 Der multiset Constructor mit Bereichsangabe

Dieser Constructor erzeugt einen neuen **multiset** anhand eines bereits bestehenden **multiset**-Objektes. Die Pointer **pFirst** und **pLast** bezeichnen dabei Anfang- und Endwert eines Bereiches. Das Element, auf welches der Iterator **pLast** verweist gehört *nicht* mehr zum kopierten Bereich – d.h. **pLast** verweist auf das erste Element hinter dem zu kopierenden Bereich. Für jedes der im Bereich befindlichen Elemente wird einmal der Copy-Constructor von **K** mit dem korrespondierenden Element des übergebenen Bereiches aufgerufen. Zur Sortierung der Elemente im **multiset** wird die Ordnungsfunktion **C** herangezogen.

Syntax:

```
multiset<K, C> Variablenname (K* pFirst, K* pLast);
```

Anhand der übergebenen Zeiger kann der Constructor die benötigte Anzahl an Elementen ermitteln und eine entsprechend großen Block reservieren.

### 7.1.4 Der multiset Constructor mit Bereichsangabe und alternativer Ordnungsfunktion

Dieser Constructor erzeugt einen neuen **multiset** anhand eines bereits bestehenden **multiset**-Objektes. Die Pointer **pFirst** und **pLast** bezeichnen dabei Anfang- und Endwert eines Bereiches. Das Element, auf welches der Iterator **pLast** verweist gehört *nicht* mehr zum kopierten Bereich – d.h. **pLast** verweist auf das erste Element hinter dem zu kopierenden Bereich. Der Constructor erzeugt einen neuen **multiset**, der jedoch die Ordnungsfunktion **C2** anstelle der vordefinierten Ordnungsfunktion **C** nutzt, d.h. der kopierte Bereich ist in der **multiset**-Kopie anders sortiert als im Original.

Für jedes der im Bereich befindlichen Elemente wird einmal der Copy-Constructor von **K** mit dem korrespondierenden Element des übergebenen Bereiches aufgerufen.

Syntax:

```
multiset<K, C> Variablenname (K* pFirst, K* pLast,
                             Compare& C2);
```

Anhand der übergebenen Zeiger kann der Constructor die benötigte Anzahl an Elementen ermitteln und eine entsprechend großen Block reservieren.

### 7.1.5 Der multiset Copy-Constructor

Dieser Constructor erzeugt einen neuen **multiset** anhand eines bereits bestehenden **multiset**-Objektes. Das neue **multiset**-Objekt ist eine vollständige Kopie des übergebenen Objektes **S**.

Syntax:

```
multiset<K,C> Variablenname (const multiset<K,C>& S);
```

Das folgende Beispielprogramm zeigt, wie der Copy-Constructor verwendet werden kann:



```
//=====
// PROGRAMM: MULTISSET_BSP_03
//=====

#include <set>
#include <iostream.h>
using namespace std;

void main (void)
{
    multiset<char, less<char> > aMySet;

    aMySet.insert('A');
    aMySet.insert('B');
    aMySet.insert('B');
    aMySet.insert('C');
    aMySet.insert('D');

    // den Multiset kopieren
    multiset<char, less<char> > aMySet2 (aMySet);
}
```

## 7.2 Die multiset Methoden

Die folgende Tabelle faßt die Methoden der **multiset** Containerklasse zusammen. In den nachstehenden Abschnitten werden die Methoden einzeln behandelt und mit Beispielen erläutert.

<b>Methoden der Containerklasse MULTiset</b>	
<i>Methode</i>	<i>Bedeutung</i>
begin	Rückgabe eines <b>iterator</b> , der auf den ersten im <b>multiset</b> enthaltenen Key zeigt
count	Gibt die Anzahl aller Elemente (Keys) zurück, die mit dem übergebenen Key übereinstimmen.
empty	Gibt den Wert <b>true</b> zurück, wenn der <b>multiset</b> keine Datenelemente enthält
end	Rückgabe eines <b>iterator</b> , der hinter den letzten Key in im <b>multiset</b> zeigt
equal_range	Diese Methode ermittelt ein Iteratoren-Paar, dessen erstes Element dem <b>lower_bound()</b> und dessen zweites Element dem <b>upper_bound</b> entspricht
erase	Löscht einen Key aus dem <b>multiset</b>
find	Gibt, falls ein Key zum übergebenen Key existiert, einen Iterator auf dieses Element zurück, sonst einen Iterator der auf <b>end()</b> zeigt
insert	Fügt die Kopie eines Keys oder eines Key-Bereiches in den <b>multiset</b> ein.
key_comp	Gibt das Vergleichsobjekt zurück, welches zum Vergleich der Schlüssel verwendet wird.
lower_bound	Gibt einen Iterator zurück, der auf die erste Position im <b>multiset</b> zeigt, an der ein Key eingefügt werden kann, ohne daß die Ordnung der Klasse <b>C</b> verletzt wird. Der Iterator zeigt auf <b>end()</b> wenn eine solche Position nicht gefunden werden konnte.
max_size	Gibt die maximale Anzahl an Datenelementen zurück, welche der <b>multiset</b> enthalten kann
rbegin	Rückgabe eines <b>reverse_iterator</b> , der auf den letzten im <b>multiset</b> enthaltenen Key zeigt
rend	Rückgabe eines <b>reverse_iterator</b> , der vor den ersten im <b>multiset</b> enthaltenen Key zeigt
size	Gibt die Anzahl von Keys zurück, die aktuell im <b>multiset</b> enthalten sind
swap	Tauscht den Inhalt zweier <b>multiset</b> -Objekte aus
upper_bound	Gibt einen Iterator zurück, der auf die letzte Position im <b>multiset</b> zeigt, an der ein Key eingefügt werden kann, ohne daß die Ordnung der Klasse <b>C</b> verletzt wird. Der Iterator zeigt auf <b>end()</b> wenn eine solche Position nicht gefunden werden konnte.
value_comp	Gibt das Vergleichsobjekt zurück, welches zum Vergleich der Schlüssels verwendet wird.

Tabelle 7-1– Methoden der Containerklasse multiset

### 7.2.1 Die multiset Methode begin

Rückgabe eines **iterator**, der auf das erste im **multiset** enthaltene Element zeigt.

Syntax:

```
iterator begin ();
```

Die wohl häufigste Anwendung der Methode **begin()** liegt in der Verarbeitung von Daten durch Schleifen.

Der Aufruf von **begin()** verändert den Inhalt des **multiset** nicht.



```
//=====
// PROGRAMM: MULTISSET_BSP_04
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;

void main (void)
{
    multiset<int, less<int> > aSet;
    multiset<int, less<int> >::const_iterator aI;

    aSet.insert(3);
    aSet.insert(3);
    aSet.insert(8);
    aSet.insert(8);
    aSet.insert(12);

    for (aI=aSet.begin(); aI!=aSet.end(); aI++)
    {
        cout << "Wert: " << *aI << endl;
    }
}
```

### 7.2.2 Die multiset Methode count

Rückgabe der Anzahl der Keys, die mit einem übergebenen Schlüssel übereinstimmen.

Syntax:

```
size_type count (const K& key) const;
```

Die wichtigste Bedeutung der Methode **count()** liegt beim einfachen **multiset** darin festzustellen, ob ein bestimmter Schlüssel existiert oder nicht. Dies kann zwar auch über **find()** festgestellt werden, da **find()** aber einen Iterator zurückgibt, muß dieser erst wieder mit einem weiteren Vergleich überprüft werden.



```
//=====
// PROGRAMM: MULTISSET_BSP_05
//=====

#include <iomanip.h>
#include <iostream.h>
```

```

#include <set>
using namespace std;

void main (void)
{
    multiset<int, less<int> > aSet;
    multiset<int, less<int> >::const_iterator aI;

    aSet.insert(3);
    aSet.insert(3);
    aSet.insert(8);

    cout << "Count zu Key 3: " << aSet.count(3) << endl;
    cout << "Count zu Key 7: " << aSet.count(7) << endl;
}

```

### 7.2.3 Die multiset Methode empty

Die Methode **empty()** prüft, ob der **multiset** Datenelemente enthält oder nicht. Sind Datenelemente enthalten (dies entspricht einem Rückgabewert von **size()** größer als Null), so wird der Wert **false** zurückgegeben, sind keine Datenelemente vorhanden ist der Rückgabewert **true**.

Der Aufruf von **empty()** verändert den Inhalt des **multiset**-Objektes nicht.

Syntax:

```
bool empty () const;
```

```

//=====
// PROGRAMM: MULTISSET_BSP_06
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;

void main (void)
{
    multiset<int, less<int> > aSet;

    if (aSet.empty())
    {
        cout << "Multiset ist leer" << endl;
    }
    else
    {
        cout << "Multiset enthält Keys" << endl;
    }

    aSet.insert(3);
    aSet.insert(6);
    aSet.insert(7);

    if (aSet.empty())
    {
        cout << "Multiset ist leer" << endl;
    }
    else
    {
        cout << "Multiset enthält Keys" << endl;
    }
}

```



}

### 7.2.4 Die multiset Methode end

Rückgabe eines **iterator**, der *hinter* den letzten im **multiset** enthaltenen Key zeigt.

Syntax:

```
iterator end ();
```

Die wohl häufigste Anwendung der Methode **end()** liegt in der Verarbeitung von Daten durch Schleifen.

Es ist zu beachten, daß **end()** hinter den letzten gültigen Key zeigt, alle Schleifen müssen also auf „ungleich“ (Operator **!=**) oder „echt kleiner als“ prüfen (Operator **<**) und nicht auf „kleiner oder gleich“ (Operator **<=**). Zeigt ein **iterator** auf **end()** führt jeder dereferenzierende Zugriff zu einer Exception vom Typ **out\_of\_range**. Wird die Exception nicht abgefangen (wie zweiten Beispiel), bricht das Programm unkontrolliert ab (Absturz).

Der Aufruf von **end()** verändert den Inhalt des **multiset**-Objektes nicht.



```
//=====
// PROGRAMM: MULTISSET_BSP_07
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;

void main (void)
{
    multiset<int, less<int> > aSet;
    multiset<int, less<int> >::const_iterator aI;

    aSet.insert(4);
    aSet.insert(4);
    aSet.insert(6);

    for (aI=aSet.begin(); aI!=aSet.end(); aI++)
    {
        cout << "Wert: " << *aI << endl;
    }
}
```

Ein Beispiel, welches die Auslösung einer Exception aufgrund eines illegalen Zugriffs auf **end()** zeigt, ist im Abschnitt zur Methode **end()** bei der Containerklasse **vector** aufgeführt.

### 7.2.5 Die multiset Methode equal\_range

Gibt ein Iteratoren-Paar zurück, dessen erstes Element der Rückgabe von **lower\_bound()** (erste Stelle an der ein angegebener Key eingefügt werden kann) und dessen zweites Element der Rückgabe von **upper\_bound()** (letzte Stelle, an der ein angegebener Key eingefügt werden kann) entspricht.

Syntax:

```
pair<const_iterator, const_iterator>  
equal_range(const K& key) const;
```

### 7.2.6 Die multiset Methode erase

Löscht ein oder mehrere Datenelemente an der angegebenen Position aus dem **multiset**.

Syntax:

```
void erase(iterator toDel);  
void erase(iterator First, iterator Last);  
size_type erase (const K& key);
```

Die Methode **erase()** kann verwendet werden, um einzelne Keys oder ganze Bereiche aus dem **multiset** zu löschen.

*Im Gegensatz zum vector und dem deque werden durch das Löschen im multiset nur die Iteratoren, Verweise und Referenzen auf Datensätze im gelöschten Bereich ungültig.*

*Da es sich beim multiset nicht um ein Array handelt (welches als geschlossener Block gespeichert werden muß), gibt es keine Notwendigkeit, das multiset-Objekt im Speicher zu verschieben.*

*Iteratoren, Verweise und Referenzen, die nicht auf einen durch die Methode gelöschten Bereich zeigen, bleiben daher gültig.*

Die erste Form der Methode sorgt dafür, daß ein einzelner Key aus dem **multiset** gelöscht und der Destructor des zu löschenden Keys aufgerufen wird.

Bei der zweiten Syntaxform wird ein zu löschender Bereich angegeben, für jeden zu löschenden Key wird der Destructor aufgerufen. Die Angabe erfolgt durch zwei Iteratoren, die auf das erste und das letzte zu löschende Element zeigen.

Es ist besonders bei den Bereichsangaben zu beachten, daß nicht versehentlich eine **out\_of\_range** Exception erzeugt wird.

Ein Beispiel, welches die Auslösung einer Exception aufgrund eines illegalen Zugriffs durch **erase()** zeigt, ist im Abschnitt zur Methode **erase()** bei der Containerklasse **vector** aufgeführt.

Die dritte Form der **erase**-Methode löscht alle Keys, die mit einem angegebenen Key übereinstimmen. Da es sich beim **multiset** um eine 1:1-Beziehung handelt ist die Anzahl der gelöschten Keys dementsprechend entweder Eins oder Null.

### 7.2.7 Die multiset Methode find

Die Methode **find()** gibt einen Iterator auf einen im **multiset** gespeicherten Key zurück. Ist ein entsprechender Key im **multiset** nicht enthalten, so wird ein Iterator auf **end()** zurückgegeben, der hinter den letzten gültigen Eintrag zeigt.

Der Aufruf von **find()** selbst verändert den Inhalt des **multiset**-Objektes nicht.

Syntax:

```
iterator find (const K& key) const;
```

Das nachfolgende Beispielprogramm zeigt den Einsatz der **find()**-Methode in einem Programm:



```
//=====
// PROGRAMM: MULTISSET_BSP_08
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;

void main (void)
{
    multiset<int, less<int> > aSet;
    multiset<int, less<int> >::iterator aI;

    aSet.insert(1);
    aSet.insert(3);
    aSet.insert(4);

    aI = aSet.find(7);
    if (aI != aSet.end())
    {
        cout << "Key zu 7 existiert" << endl;
    }
    else
    {
        cout << "Key zu 7 existiert nicht" << endl;
    }

    aI = aSet.find(4);
    if (aI != aSet.end())
    {
        cout << "Key zu 4 existiert" << endl;
    }
    else
    {
        cout << "Key zu 4 existiert nicht" << endl;
    }
}
```

### 7.2.8 Die multiset Methode insert

Fügt einen oder mehrere Keys vor der angegebenen Position im **multiset** ein.

Syntax:

```
iterator insert (const K& key);
void          insert (const K* first, const K* last);
iterator insert (iterator pos, const K& key);
```

Die Methode **insert()** kann verwendet werden um einzelne Keys oder ganze Bereiche in einen **multiset** einzufügen.

Da ein **multiset** als Baumstruktur aufgebaut ist, muß (anders als bei **vector** oder **deque**) beim Einfügen von Keys nichts umkopiert werden, was die Methode **insert()** für **multiset**-Objekte sehr effizient macht. Der Zeitaufwand für das Einfügen in einem **multiset**-Objekt ist daher immer konstant.

*Im Gegensatz zum **vector** und dem **deque** werden durch das Einfügen keine Iteratoren, Verweise oder Referenzen ungültig.*

Die erste Form der Methode sorgt dafür, daß ein einzelner Key in den **multiset** eingefügt wird. Eingefügt wird die Kopie des Keys. Rückgabewert der Methode ist ein **iterator**, der auf das eingefügte Element im **multiset** zeigt.

Mit der zweiten Form der **insert()**-Methode können Bereiche aus einem anderen **multiset** in den **multiset** kopiert werden. Der Bereich wird durch Anfangs- und Endadresse festgelegt. Es sollte darauf geachtet werden, daß die Endadresse nicht vor der Anfangsadresse liegen darf und daß beide Adressen zum gleichen **multiset**-Objekt gehören.

Die dritte **insert()**-Form fügt, wie die erste, einen Schlüssel ein, sofern noch nicht vorhanden. Beginnt jedoch mit der Suche nach der richtigen Position zum Einfügen an der angegebenen Position des Iterators **pos**.

```
//=====
// PROGRAMM: MULTISSET_BSP_09
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;

void main (void)
{
    typedef multiset<int, less<int> > mysettype;
    mysettype aSet;

    aSet.insert(4);
    aSet.insert(5);
}
```



### 7.2.9 Die multiset Methode key\_comp

Gibt das Vergleichsobjekt zurück, welches zum Vergleich der Schlüssel herangezogen wird.

Syntax:

```
Compare key_comp () const;
```

### 7.2.10 Die multiset Methode lower\_bound

Die Methode gibt einen Iterator zurück, welcher auf die erste Position zeigt, an der zu einem gegebenen Schlüssel eingefügt werden kann, ohne daß die durch das Compare-Objekt vorgegebenen Ordnungskriterien verletzt werden. Der Iterator zeigt auf die Position **end()**, wenn keine gültige Position gefunden werden kann.

Syntax:

```
iterator lower_bound (const K& key);
```

### 7.2.11 Die multiset Methode max\_size

Die Methode **max\_size()** gibt die maximale Anzahl an Keys zurück, die in einem **multiset** gespeichert werden kann. Der Wert von **max\_size()** ist u.a. vom Typ des Schlüssels abhängig.

Syntax:

```
size_type max_size () const;
```

Der Wert ist zudem abhängig von der Implementation und vom verwendeten Betriebssystem. Bei modernen Compilern und entsprechender Einstellung kann man davon ausgehen, das 32-Bit Adressen verwendet werden, also theoretisch 4 Gigabytes angesprochen werden können.

Da **multiset**-Objekte nicht als Block gespeichert sein müssen, wird die Anzahl der Datenelemente sonst nur durch den freien RAM-Speicher begrenzt, die Größe der zu verwaltenden Datenelemente, sowie die Segmentierung (Zersplitterung) des Speichers.

Der Aufruf der Methode verändert den Inhalt des **multiset**-Objektes nicht.



```
//=====
// PROGRAMM: MULTISSET_BSP_10
//=====

#include <iostream.h>
#include <set>
using namespace std;

void main (void)
{
    multiset<char, less<char> > aSet;

    cout << "Multiset kann " << aSet.max_size ()
         << " Datenelemente aufnehmen";
}

```

### 7.2.12 Die multiset Methode rbegin

Rückgabe eines **reverse\_iterator** oder **const\_reverse\_iterator**, der auf den letzten im **multiset** enthaltenen Key zeigt.

Syntax:

```
reverse_iterator rbegin ();
```

Die wohl häufigste Anwendung der Methode **rbegin()** liegt in der Verarbeitung von Daten durch rückwärts gerichtete Schleifen.

Der Aufruf von **rbegin()** verändert den Inhalt des **multiset** nicht.

```
//=====
// PROGRAMM: MULTISSET_BSP_11
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;

void main (void)
{
    multiset<int, less<int> > aSet;
    multiset<int, less<int> >::reverse_iterator aI;

    aSet.insert(1);
    aSet.insert(3);
    aSet.insert(4);

    for (aI=aSet.rbegin(); aI!=aSet.rend(); aI++)
    {
        cout << "Wert: " << *aI << endl;
    }
}
```



### 7.2.13 Die multiset Methode rend

Rückgabe eines **reverse\_iterator** oder **const\_reverse\_iterator**, der vor den ersten im **multiset** enthaltene Key zeigt.

Syntax:

```
reverse_iterator rend ();
```

Die wohl häufigste Anwendung der Methode **rend()** liegt in der Verarbeitung von Daten durch rückwärts gerichtete Schleifen.

Es ist zu beachten, daß **rend()** vor den ersten gültigen Key zeigt, alle Schleifen müssen also auf „ungleich“ prüfen (Operator !=). Zeigt ein **iterator** auf **rend()** führt jeder dereferenzierende Zugriff zu einer Exception vom Typ **out\_of\_range**. Wird die Exception nicht wie zweiten Beispiel abgefangen bricht das Programm unkontrolliert ab (Absturz).

Der Aufruf von **rend()** verändert den Inhalt des **multiset** nicht.

Ein Anwendungsbeispiel ist unter **rbegin()** zu finden. Ein Beispiel, welches die Auslösung einer Exception aufgrund eines illegalen Zugriffs auf **rend()** zeigt, ist im Abschnitt über die Methode **rend()** der Containerklasse **vector** zu finden.

### 7.2.14 Die multiset Methode size

Gibt die Anzahl von Keys zurück, die aktuell im **multiset** enthalten sind.

Syntax:

```
size_type size () const;
```

Der Aufruf von **size()** verändert den Inhalt des **multiset** nicht.



```
//=====
// PROGRAMM: MULTISSET_BSP_12
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;

void main (void)
{
    multiset<int, less<int> > aSet;
    multiset<int, less<int> >::const_reverse_iterator aI;

    aSet.insert(1);
    aSet.insert(3);
    aSet.insert(4);

    cout << "Anzahl aktuell: " << aSet.size() << endl;
}

```

### 7.2.15 Die multiset Methode swap

Tauscht die Inhalte zweier **multiset**-Objekte aus. Die **multiset**-Objekte müssen von gleichem Typ sein, also gleiche Key- und Compare-Typen enthalten.

Syntax:

```
void swap (multiset<K, C>& s);
```

*Es ist zu beachten, daß somit alle Iteratoren, Verweise und Referenzen ungültig werden, da es sich um Pointer handelt und die Datensätze durch den Tausch im Speicher verschoben werden. Alle Iteratoren, Verweise und Referenzen müssen neu ermittelt werden.*



```
//=====
// PROGRAMM: MULTISSET_BSP_13
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;

void main (void)
{
    typedef multiset<int, less<int> > mysettype;
    mysettype aSet1;
    mysettype aSet2;

    aSet1.insert(1);
    aSet1.insert(3);
    aSet1.insert(4);

    aSet2.insert(11);
    aSet2.insert(33);
    aSet2.insert(44);
}

```

```
aSet1.swap (aSet2);
}
```

### 7.2.16 Die multiset Methode upper\_bound

Die Methode gibt einen Iterator zurück, welcher auf die letzte Position zeigt, an der zu einem gegebenen Schlüssel eingefügt werden kann, ohne daß die durch das Compare-Objekt vorgegebenen Ordnungskriterien verletzt werden. Der Iterator zeigt auf die Position **end()**, wenn keine gültige Position gefunden werden kann.

Syntax:  

```
iterator upper_bound (const K& key);
```

### 7.2.17 Die multiset Methode value\_comp

Gibt das Vergleichsobjekt zurück, welches zum Vergleich der Schlüssel herangezogen wird.

Syntax:  

```
set_compare<K,C> value_comp () const;
```

## 7.3 Die multiset Operatoren

Mit den folgenden Operatoren kann auf eine Containerklasse vom Typ **multiset** zugegriffen werden:

<b>Operatoren der Containerklasse MULTiset</b>	
<i>Operator</i>	<i>Bedeutung</i>
=	Zuweisung zwischen zwei <b>multiset</b> -Objekten gleichen Typs
==	Vergleich zwischen zwei <b>multiset</b> -Objekten gleichen Typs
!=	Vergleich zwischen zwei <b>multiset</b> -Objekten gleichen Typs
<	Lexikographischer Vergleich zwischen zwei <b>multiset</b> -Objekten gleichen Typs
>	Lexikographischer Vergleich zwischen zwei <b>multiset</b> -Objekten gleichen Typs
>=	Lexikographischer Vergleich zwischen zwei <b>multiset</b> -Objekten gleichen Typs
<=	Lexikographischer Vergleich zwischen zwei <b>multiset</b> -Objekten gleichen Typs

Tabelle 7-1– Operatoren der Containerklasse multiset

### 7.3.1 Der multiset Operator =

Der Operator ersetzt den Inhalt eines **multiset**-Objektes durch den Inhalt des zugewiesenen **multiset**-Objektes gleichen Typs.

Syntax:  

```
multiset<K,C>& operator= (const multiset<K,C>& s);
```

*Durch die Zuweisung werden alle Iteratoren, Verweise und Referenzen auf dieser Multiset ungültig.*

*Alle Iteratoren, Verweise und Referenzen müssen neu ermittelt werden.*



```
//=====
// PROGRAMM: MULTISSET_BSP_14
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;

void main (void)
{
    typedef multiset<int, less<int> > mysettype;
    mysettype aSet1;
    mysettype aSet2;
    mysettype::iterator aI;

    aSet1.insert(1);
    aSet1.insert(2);
    aSet1.insert(3);

    aSet2 = aSet1;

    for (aI=aSet1.begin(); aI!=aSet1.end(); aI++)
    {
        cout << "aSet1 Wert: " << *aI << endl;
    }

    for (aI=aSet2.begin(); aI!=aSet2.end(); aI++)
    {
        cout << "aSet2 Wert: " << *aI << endl;
    }
}
```

### 7.3.2 Der multiset Operator ==

Der Operator prüft, ob die Inhalte des linken und rechten Operanden (**multiset**-Objekte gleichen Typs) übereinstimmen. Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn beide **multiset**-Objekte die gleichen Keys in der gleichen Reihenfolge enthalten.

Syntax:

```
bool operator== (const multiset<K,C>& s) const;
```



```
//=====
// PROGRAMM: MULTISSET_BSP_15
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;

void main (void)
{
    typedef multiset<int, less<int> > mysettype;
```

```

mysettype aSet1;
mysettype aSet2;
mysettype::iterator aI;

aSet1.insert(1);

if (aSet1 == aSet2)
{
    cout << "Die Multiset-Objekte sind gleich" << endl;
}
else
{
    cout << "Die Multiset-Objekte sind nicht gleich" << endl;
}

aSet2.insert(1);

if (aSet1 == aSet2)
{
    cout << "Die Multiset-Objekte sind gleich" << endl;
}
else
{
    cout << "Die Multiset-Objekte sind nicht gleich" << endl;
}
}

```

### 7.3.3 Der multiset Operator !=

Der Operator prüft, ob die Inhalte des linken und rechten Operanden (**multiset**-Objekte gleichen Typs) verschieden sind. Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn beide **multiset**-Objekte unterschiedliche Key oder gleiche Keys in unterschiedlicher Reihenfolge enthalten.

Syntax:

```
bool operator!= (const multiset<K,C>& s) const;
```

```

//=====
// PROGRAMM: MULTISSET_BSP_16
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;

void main (void)
{
    typedef multiset<int, less<int> > mysettype;
    mysettype aSet1;
    mysettype aSet2;
    mysettype::iterator aI;

    aSet1.insert(1);

    if (aSet1 != aSet2)
    {

```



```

        cout << "Die Multiset-Objekte sind ungleich" << endl;
    }
    else
    {
        cout << "Die Multiset-Objekte sind nicht ungleich" << endl;
    }

    aSet2.insert(1);

    if (aSet1 != aSet2)
    {
        cout << "Die Multiset-Objekte sind ungleich" << endl;
    }
    else
    {
        cout << "Die Multiset-Objekte sind nicht ungleich" << endl;
    }
}

```

### 7.3.4 Der multiset Operator <

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch kleiner ist, als der des rechten Operanden (**multiset**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „<“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch kleiner ist als der rechte Operand.

Syntax:

```
bool operator< (const multiset<K,C>& s) const;
```



```

//=====
// PROGRAMM: MULTISSET_BSP_17
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;

void main (void)
{
    multiset<int, less<int> > aSet1;
    multiset<int, less<int> > aSet2;

    aSet1.insert(1);
    aSet2.insert(2);

    if (aSet1 < aSet2)
    {
        cout << "aSet1 ist kleiner als aSet2" << endl;
    }
    else
    {
        cout << " aSet1 ist nicht kleiner als aSet2" << endl;
    }
}

```

### 7.3.5 Der multiset Operator >

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch größer ist, als der des rechten Operanden (**multiset**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „>“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch größer ist als der rechte Operand.

Syntax:

```
bool operator> (const multiset<K,C>& s) const;
```

```
//=====
// PROGRAMM: MULTISSET_BSP_18
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;

void main (void)
{
    multiset<int, less<int> > aSet1;
    multiset<int, less<int> > aSet2;

    aSet1.insert(1);
    aSet2.insert(2);

    if (aSet1 > aSet2)
    {
        cout << "aSet1 ist größer als aSet2" << endl;
    }
    else
    {
        cout << " aSet1 ist nicht größer als aSet2" << endl;
    }
}
```



### 7.3.6 Der multiset Operator <=

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch kleiner oder gleich mit dem rechten Operanden ist (**multiset**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „<=“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch kleiner oder gleich mit dem rechten Operanden ist.

Syntax:

```
bool operator<= (const multiset<K,C>& s) const;
```

```
//=====
// PROGRAMM: MULTISSET_BSP_19
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;
```



```

void main (void)
{
    multiset<int, less<int> > aSet1;
    multiset<int, less<int> > aSet2;

    aSet1.insert(1);
    aSet2.insert(2);

    if (aSet1 <= aSet2)
    {
        cout << "aSet1 ist kleiner gleich aSet2" << endl;
    }
    else
    {
        cout << " aSet1 ist nicht kleiner gleich aSet2" << endl;
    }
}

```

### 7.3.7 Der multiset Operator >=

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch größer oder gleich mit dem rechten Operanden ist (**multiset**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „>=“ aufgerufen wird. Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch größer oder gleich mit dem rechten Operanden ist.

Syntax:

```
bool operator>= (const multiset<K,C>& s) const;
```



```

//=====
// PROGRAMM: MULTISSET_BSP_20
//=====

#include <iomanip.h>
#include <iostream.h>
#include <set>
using namespace std;

void main (void)
{
    multiset<int, less<int> > aSet1;
    multiset<int, less<int> > aSet2;

    aSet1.insert(1);
    aSet2.insert(2);

    if (aSet1 >= aSet2)
    {
        cout << "aSet1 ist größer gleich aSet2" << endl;
    }
    else
    {
        cout << " aSet1 ist nicht größer gleich aSet2" << endl;
    }
}

```

## 8. Die Containerklasse map

Die Containerklasse **map** realisiert, wie **set** einen assoziativen Behälter (Hashtable). Im Gegensatz zu einem **set**, der nur einen Schlüsselwert speichert, werden in einer **map** jedoch Schlüssel/Wertpaare abgelegt (Key/Value-Paare), zu jedem Key eines **set** kommt so noch ein zweiter Wert hinzu. Die **map** ist durch eine vom Benutzer definierte Vergleichsfunktion (compare-Klasse) sortiert. Die Key/Value-Beziehung setzt voraus, daß jeder Schlüssel nur ein einziges mal (unique) in der **map** vorkommen kann und daß es für jeden Key genau einen zugehörigen Value gibt. Values dürfen wiederholt zu unterschiedlichen Schlüsseln vorkommen, sind also nicht unique.

Die interne Struktur der **map** (Hashtable) ermöglicht es, ein Key/Value-Paar sehr effizient zu ermitteln<sup>1</sup>.

Eine häufige Form der Verwendung von **map**-Objekten ist z.B. die Speicherung von Programmeinstellungen oder langen Datenlisten, bei denen jede Einstellung bzw. jedes Datum nur einmal vorkommen kann.

Ein **map**-Objekt hat intern einen anderen Aufbau (Baumstruktur) als ein **vector**-, **deque**- oder **list**-Objekt, um eine effiziente Suche zu ermöglichen.

Die notwendige Speicherverwaltung wird automatisch vorgenommen und ist so effizient wie möglich realisiert. Wie für **list**-Objekte wird auch für eine **map** kein geschlossener Speicherbereich benötigt. Um ein **map**-Objekt zu erweitern ist daher kein Umkopieren der gesamten Containerklasse notwendig, da lediglich die Verkettung innerhalb der Baumstruktur geändert werden muß.

Ein wahlfreier Zugriff auf **map**-Objekte ist, da es sich um eine 1:1 Zuordnung handelt, möglich, der **operator[]** (Zugriff über Indexklammer) ist daher für den schnellen Zugriff auf die **map**-Inhalte definiert.

In einem Punkt weicht **map** von den bisher beschriebenen Container-Klassen **vector**, **deque** und **list** ab. Während diese lediglich eine Klassenangabe erwarten, um die Containerklasse zu definieren, also lediglich die zu verwaltende Klasse benötigen, braucht ein **map**-Objekt insgesamt drei Klassenangaben. Die erste erforderliche Angabe verweist auf die Klasse, welche den Key repräsentiert, die zweite auf die Value-Klasse und die dritte Klasse definiert die Vergleichsklasse für die Sortierung:

Die Deklaration eines **map**-Objektyps kann durch eine **typedef**-Anweisung weiter vereinfacht werden und hat folgenden grundsätzlichen Aufbau:

```
map<Keytyp, Valuetyp, Ordnungsobjekt<Keytyp> > Variablenname;
typedef map<Keytyp, Valuetyp, Ordnungsobjekt<Keytyp> > Datentypname;
```

**ACHTUNG:** Das Leerzeichen zwischen den beiden schließenden, spitzen Klammern „> >“ ist zwingend notwendig, da der Compiler zusammenhängende spitze Klammern „>>“ versucht als den entsprechenden Shift-Operator zu verwenden.

<sup>1</sup> Die übliche Form der Realisierung ist ein binärer Baum, der in jedem Knoten der Baumstruktur genau ein Key/Value-Paar speichert.

## 8.1 Die map Constructoren

Die folgende Tabelle faßt die Constructoren der Containerklasse **map** zusammen. In den nachstehenden Abschnitten werden die Constructoren einzeln behandelt und mit Beispielen erläutert.

<b>Constructoren der Containerklasse MAP</b>	
<i>Constructor</i>	<i>Bedeutung</i>
map< K, V, C > VarName;	Standard-Constructor, erzeugt ein neues, leeres <b>map</b> -Objekt
map< K, V, C > VarName (const Compare& newcompare);	Dieser Constructor erzeugt eine neue, leere <b>map</b> , welche die enthaltenen Key/Value-Paare intern anhand der Vergleichsfunktion newcompare ordnet, anstelle der ursprünglich definierten compare-Funktion.
map< K, V, C > VarName (V* pFirst, V* pLast);	Dieser Constructor erzeugt eine neue <b>map</b> anhand einer bereits bestehenden <b>map</b> . Die Pointer <b>pFirst</b> und <b>pLast</b> bezeichnen dabei Anfang- und Endwert eines Bereiches. Das Element, auf welches der Zeiger <b>pLast</b> verweist gehört <i>nicht</i> mehr zum kopierten Bereich.
map<K, V, C> VarName (V* pFirst, V* pLast, const Compare& newcompare);	Dieser Constructor erzeugt eine neue <b>map</b> anhand einer bereits bestehenden <b>map</b> . Die Pointer <b>pFirst</b> und <b>pLast</b> bezeichnen dabei Anfang- und Endwert eines Bereiches. Das Element, auf welches der Zeiger <b>pLast</b> verweist gehört <i>nicht</i> mehr zum kopierten Bereich. Verwendet die übergebene compare-Funktion zur internen Ordnung der Key/Value-Paare.
map< K, V, C > VarName (const map <Key, Value, Compare>& m);	Dieser Constructor erzeugt eine neue <b>map</b> anhand eines bereits bestehenden <b>map</b> -Objektes. Das neue <b>map</b> -Objekt ist eine vollständige Kopie des übergebenen <b>map</b> -Objektes <b>m</b> .

Tabelle 8-1– Constructoren der Containerklasse map

### 8.1.1 Der map Standard-Constructor

Der **map** Standard-Constructor, erzeugt ein neues, leeres **map**-Objekt vom Typ **K,V,C**<sup>2</sup>.

Syntax:

<sup>2</sup> **K** ist der Platzhalter für den in der **map** verwalteten Schlüssel (Key), **V** ist der Platzhalter für den in der **map** verwendeten Wert (Value) und **C** ist der Platzhalter für die in der **map** verwendete Ordnungsklasse.

```
map<K, V, C> Variablenname;
```

Das folgende Beispielprogramm zeigt den Einsatz des Standard-Constructors innerhalb eines Programms:

```
//=====
// PROGRAMM: MAP_BSP_01
//=====

#include <map>
using namespace std;

typedef map<char, int, less<char> > MyMapType;

void main (void)
{
    MyMapType aMap;
    map<int, long, less<int> > aMap2;
}
```



### 8.1.2 Der map Constructor mit alternativer Ordnungsfunktion

Dieser Constructor erzeugt eine neue **map** mit der Ordnungsfunktion **C2** anstelle der vordefinierten Ordnungsfunktion **C**.

Syntax:

```
map<K, V, C> Variablenname (Compare& C2);
```

Das folgende Beispielprogramm zeigt den Einsatz des Constructors innerhalb eines Programms:

```
//=====
// PROGRAMM: MAP_BSP_02
//=====

#include <map>
using namespace std;

typedef map<char, int, less<char> > MyMapType;

void main (void)
{
    MyMapType aMap(greater<char>);
}
```



### 8.1.3 Der map Constructor mit Bereichsangabe

Dieser Constructor erzeugt eine neue **map** anhand eines bereits bestehenden **map**-Objektes. Die Pointer **pFirst** und **pLast** bezeichnen dabei Anfang- und Endwert eines Bereiches. Das Element, auf welches der Iterator **pLast** verweist gehört **nicht** mehr zum kopierten Bereich – d.h. **pLast** verweist auf das erste Element hinter dem zu kopierenden Bereich.

Für jedes der im Bereich befindlichen Elemente wird einmal der Copy-Constructor von **K** und der Copy-Constructor von **V** mit den korrespondierenden Elementen des übergebenen Bereiches aufgerufen. Zur Sortierung der Elemente in der **map** wird die Ordnungsfunktion **C** herangezogen.

Syntax:

```
map<K, V, C> Variablenname (V* pFirst, V* pLast);
```

Anhand der übergebenen Zeiger kann der Constructor die benötigte Anzahl an Elementen ermitteln und eine entsprechend großen Block reservieren.

#### 8.1.4 Der map Constructor mit Bereichsangabe und alternativer Ordnungsfunktion

Dieser Constructor erzeugt eine neue **map** anhand eines bereits bestehenden **map**-Objektes. Die Pointer **pFirst** und **pLast** bezeichnen dabei Anfang- und Endwert eines Bereiches. Das Element, auf welches der Iterator **pLast** verweist gehört *nicht* mehr zum kopierten Bereich – d.h. **pLast** verweist auf das erste Element hinter dem zu kopierenden Bereich. Der Constructor erzeugt eine neue **map**, die jedoch die Ordnungsfunktion **C2** anstelle der vordefinierten Ordnungsfunktion **C** nutzt, d.h. der kopierte Bereich ist in der **map**-Kopie anders sortiert als im Original.

Für jedes der im Bereich befindlichen Elemente wird einmal der Copy-Constructor von **K** und der Copy-Constructor von **V** mit den korrespondierenden Elementen des übergebenen Bereiches aufgerufen.

Syntax:

```
map<K, V, C> Variablenname (V* pFirst, V* pLast,
                          Compare& C2);
```

Anhand der übergebenen Zeiger kann der Constructor die benötigte Anzahl an Elementen ermitteln und eine entsprechend großen Block reservieren.

#### 8.1.5 Der map Copy-Constructor

Dieser Constructor erzeugt eine neue **map** anhand eines bereits bestehenden **map**-Objektes. Das neue **map**-Objekt ist eine vollständige Kopie des übergebenen Objektes M.

Syntax:

```
map<K, V, C> Variablenname (const map<K, V, C>& M);
```

Das folgende Beispielprogramm zeigt, wie der Copy-Constructor verwendet werden kann:



```
//=====
// PROGRAMM: MAP_BSP_03
//=====

#include <map>
#include <iostream.h>
using namespace std;

void main (void)
{
    map<char, int, less<char> > aMyMap;

    aMyMap['A'] = 256;
    aMyMap['B'] = 17;
```

```

aMyMap['C'] = 3;
aMyMap['D'] = 170;

// die Map kopieren
map<char, int, less<char> > aMyMap2 (aMyMap);

cout << "aMyMap ['A'] = " << aMyMap ['A'] << endl;
cout << "aMyMap2['A'] = " << aMyMap2['A'] << endl;
}

```

## 8.2 Die map Methoden

Die folgende Tabelle faßt die Methoden der **map** Containerklasse zusammen. In den nachstehenden Abschnitten werden die Methoden einzeln behandelt und mit Beispielen erläutert.

<b>Methoden der Containerklasse MAP</b>	
<i>Methoden</i>	<i>Bedeutung</i>
begin	Rückgabe eines <b>iterator</b> , der auf das erste in der <b>map</b> enthaltene Element (Key/Value-Paar) zeigt
count	Gibt die Anzahl aller Key/Value-Paare zurück, deren Schlüssel mit einem übergebenen Key übereinstimmt. Da <b>map</b> nur 1:1 Beziehungen speichern kann, ist das Ergebnis entweder Eins oder Null
empty	Gibt den Wert <b>true</b> zurück, wenn die <b>map</b> keine Datenelemente enthält
end	Rückgabe eines <b>iterator</b> , der hinter das letzte Key/Value-Paar in der <b>map</b> zeigt
equal_range	Diese Methode ermittelt ein Iteratoren-Paar, dessen erstes Element dem <b>lower_bound()</b> und dessen zweites Element dem <b>upper_bound</b> entspricht
erase	Löscht ein Key/Value-Paar an der angegebenen Position aus der <b>map</b>
find	Gibt, falls ein Key/Value-Paar zum übergebenen Key existiert, einen Iterator auf dieses Element zurück, sonst einen Iterator der auf <b>end()</b> zeigt
insert	Fügt die Kopie eines Key/Value-Paares oder eines Key/Value-Bereiches in die <b>map</b> ein, falls in der <b>map</b> noch kein Key/Value-Paar zu diesem Schlüssel existiert.
key_comp	Gibt das Vergleichsobjekt zurück, welches zum Vergleich der Schlüssel verwendet wird.
lower_bound	Gibt einen Iterator zurück, der auf die erste Position in der <b>map</b> zeigt, an der ein Key/Value-Paar eingefügt werden kann, ohne daß die Ordnung der Klasse <b>C</b> verletzt wird. Der Iterator zeigt auf <b>end()</b> wenn eine solche Position nicht gefunden werden konnte.
max_size	Gibt die maximale Anzahl an Datenelementen zurück, welche die <b>map</b> enthalten kann
rbegin	Rückgabe eines <b>reverse_iterator</b> , der auf das letzte in der <b>map</b> enthaltene Key/value-Paar zeigt
rend	Rückgabe eines <b>reverse_iterator</b> , der vor das erste in der <b>map</b> enthaltene Key/Value-Paar zeigt

<b>Methoden der Containerklasse MAP</b>	
<i>Methode</i>	<i>Bedeutung</i>
size	Gibt die Anzahl von Key/Value-Paaren zurück, die aktuell in der <b>map</b> enthalten sind
swap	Tauscht den Inhalt zweier <b>map</b> -Objekte aus
upper_bound	Gibt einen Iterator zurück, der auf die letzte Position in der <b>map</b> zeigt, an der ein Key/Value-Paar eingefügt werden kann, ohne daß die Ordnung der Klasse <b>C</b> verletzt wird. Der Iterator zeigt auf <b>end()</b> wenn eine solche Position nicht gefunden werden konnte.
value_comp	Gibt das Vergleichsobjekt zurück, welches zum Vergleich der Schlüssel/Wert-Paare verwendet wird.

Tabelle 8-1– Methoden der Containerklasse map

### 8.2.1 Die map Methode begin

Rückgabe eines **iterator** oder **const\_iterator**, der auf das erste in der **map** enthaltene Element zeigt.

Syntax:

```
iterator begin ();
const_iterator begin () const;
```

Die wohl häufigste Anwendung der Methode **begin()** liegt in der Verarbeitung von Daten durch Schleifen.

Der Aufruf von **begin()** verändert den Inhalt der **map** nicht.



```
//=====
// PROGRAMM: MAP_BSP_04
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    map<int, double, less<int> > aMap;
    map<int, double, less<int> >::const_iterator aI;

    aMap[3] = 7.0;
    aMap[8] = 1.2;
    aMap[12] = 3.14;

    for (aI=aMap.begin(); aI!=aMap.end(); aI++)
    {
        cout << "Wert: " << (*aI).second << endl;
    }
}
```

### 8.2.2 Die map Methode count

Rückgabe der Anzahl der Key/Value-Paare, deren Schlüssel mit einem übergebenen Schlüssel übereinstimmt.

Syntax:

```
size_type count (const K& key) const;
```

Die wichtigste Bedeutung der Methode `count()` liegt bei der einfachen **map** darin festzustellen, ob ein Key/Value-Paar zu einem bestimmten Schlüssel existiert oder nicht. Dies kann zwar auch über **find()** festgestellt werden, da **find()** aber einen Iterator zurückgibt, muß dieser erst wieder mit einem weiteren Vergleich überprüft werden.

```
//=====
// PROGRAMM: MAP_BSP_05
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    map<int, double, less<int> > aMap;
    map<int, double, less<int> >::const_iterator aI;

    aMap[3] = 7.0;
    aMap[8] = 1.2;
    aMap[12] = 3.14;

    if (aMap.count(7))
    {
        cout << "Key/Value-Paar zu 7 existiert" << endl;
    }
    else
    {
        cout << "Key/Value-Paar zu 7 existiert nicht" << endl;
    }

    if (aMap.count(8))
    {
        cout << "Key/Value-Paar zu 8 existiert" << endl;
    }
    else
    {
        cout << "Key/Value-Paar zu 8 existiert nicht" << endl;
    }
}
}
```



### 8.2.3 Die map Methode empty

Die Methode **empty()** prüft, ob die **map** Datenelemente enthält oder nicht. Sind Datenelemente enthalten (dies entspricht einem Rückgabewert von **size()** größer als Null), so wird der Wert **false** zurückgegeben, sind keine Datenelemente vorhanden ist der Rückgabewert **true**.

Der Aufruf von **empty()** verändert den Inhalt der **map** nicht.

```
Syntax:
bool empty () const;
```

```
//=====
// PROGRAMM: MAP_BSP_06
//=====
```



```

#include <iomanip.h>
#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    map<int, double, less<int> > aMap;

    if (aMap.empty())
    {
        cout << "Map ist leer" << endl;
    }
    else
    {
        cout << "Map enthält Key/Value-Paare" << endl;
    }

    aMap[3] = 7.0;
    aMap[8] = 1.2;
    aMap[12] = 3.14;

    if (aMap.empty())
    {
        cout << "Map ist leer" << endl;
    }
    else
    {
        cout << "Map enthält Key/Value-Paare" << endl;
    }
}

```

### 8.2.4 Die map Methode end

Rückgabe eines **iterator** oder **const\_iterator**, der *hinter* das letzte in der **map** enthaltene Key/Value-Paar zeigt.

Syntax:

```

iterator end ();
const_iterator end () const;

```

Die wohl häufigste Anwendung der Methode **end()** liegt in der Verarbeitung von Daten durch Schleifen.

Es ist zu beachten, daß **end()** hinter das letzte gültige Key/value-Paar zeigt, alle Schleifen müssen also auf „ungleich“ (Operator !=) oder „echt kleiner als“ prüfen (Operator <) und nicht auf „kleiner oder gleich“ (Operator <=). Zeigt ein **iterator** auf **end()** führt jeder dereferenzierende Zugriff zu einer Exception vom Typ **out\_of\_range**. Wird die Exception nicht abgefangen (wie zweiten Beispiel), bricht das Programm unkontrolliert ab (Absturz).

Der Aufruf von **end()** verändert den Inhalt der **map** nicht.



```

//=====
// PROGRAMM: MAP_BSP_07
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>

```

```

using namespace std;

void main (void)
{
    map<int, double, less<int> > aMap;
    map<int, double, less<int> >::const_iterator aI;

    aMap[3] = 7.0;
    aMap[8] = 1.2;
    aMap[12] = 3.14;

    for (aI=aMap.begin(); aI!=aMap.end(); aI++)
    {
        cout << "Wert: " << (*aI).second << endl;
    }
}

```

Ein Beispiel, welches die Auslösung einer Exception aufgrund eines illegalen Zugriffs auf **end()** zeigt, ist im Abschnitt zur Methode **end()** bei der Containerklasse **vector** aufgeführt.

### 8.2.5 Die map Methode equal\_range

Gibt ein Iteratoren-Paar zurück, dessen erstes Element der Rückgabe von **lower\_bound()** (erste Stelle an der ein Key/Value-Paar mit angegebenem Key eingefügt werden kann) und dessen zweites Element der Rückgabe von **upper\_bound()** (letzte Stelle, an der ein Key/Value-Paar mit angegebenem Key eingefügt werden kann) entspricht.

Syntax:

```

pair<iterator, iterator> equal_range(const K& key);
pair<const_iterator, const_iterator>
    equal_range(const K& key) const;

```

### 8.2.6 Die map Methode erase

Löscht ein oder mehrere Datenelemente an der angegebenen Position aus der **map**.

Syntax:

```

void erase(iterator toDel);
void erase(iterator First, iterator Last);
size_type erase (const K& key);

```

Die Methode **erase()** kann verwendet werden, um Datensätze oder ganze Bereiche aus der **map** zu löschen.

*Im Gegensatz zum vector und dem deque werden durch das Löschen in der map nur die Iteratoren, Verweise und Referenzen auf Datensätze im gelöschten Bereich ungültig.*

*Da es sich bei der map nicht um ein Array handelt (welches als geschlossener Block gespeichert werden muß), gibt es keine Notwendigkeit, das map-Objekt im Speicher zu verschieben.*

*Iteratoren, Verweise und Referenzen, die nicht auf einen durch die Methode gelöschten Bereich zeigen, bleiben daher gültig.*

Die erste Form der Methode sorgt dafür, daß ein einzelner Datensatz aus der **map** gelöscht und der Destructor des Key/Value-Paares für diesen Datensatz aufgerufen wird.

Bei der zweiten Syntaxform wird ein zu löschender Bereich angegeben, für jedes zu löschende Key/Value-Paar wird der Destructor aufgerufen. Die Angabe erfolgt durch zwei Iteratoren, die auf das erste und das letzte zu löschende Element zeigen.

Es ist besonders bei den Bereichsangaben zu beachten, daß nicht versehentlich eine **out\_of\_range** Exception erzeugt wird.

Ein Beispiel, welches die Auslösung einer Exception aufgrund eines illegalen Zugriffs durch **erase()** zeigt, ist im Abschnitt zur Methode **erase()** bei der Containerklasse **vector** aufgeführt.

Die dritte Form der **erase**-Methode löscht alle Key/Value-Paare, deren Schlüssel mit einem angegebenen Key übereinstimmen. Da es sich bei der **map** um eine 1:1-Beziehung handelt ist die Anzahl der gelöschten Key/Value-Paare dementsprechend entweder Eins oder Null.

### 8.2.7 Die map Methode find

Die Methode **find()** gibt einen Iterator auf ein in der **map** gespeichertes Key/Value-Paar zu einem vorgegebenem Schlüssel zurück. Ist ein entsprechendes Key/Value-Paar in der **map** nicht enthalten, so wird ein Iterator auf **end()** zurückgegeben, der hinter den letzten gültigen Eintrag zeigt.

Der Aufruf von **find()** selbst verändert den Inhalt der **map** nicht.

Syntax:

```
iterator find (const K& key) const;
```

Das nachfolgende Beispielprogramm zeigt den Einsatz der **find()**-Methode in einem Programm:



```
//=====
// PROGRAMM: MAP_BSP_08
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    map<int, double, less<int> > aMap;
    map<int, double, less<int> >::iterator aI;

    aMap[3] = 7.0;
    aMap[8] = 1.2;
    aMap[12] = 3.14;

    aI = aMap.find(7);
```

```

if (aI != aMap.end())
{
    cout << "Key/Value-Paar zu 7 existiert" << endl;
}
else
{
    cout << "Key/Value-Paar zu 7 existiert nicht" << endl;
}

aI = aMap.find(8);
if (aI != aMap.end())
{
    cout << "Key/Value-Paar zu 8 existiert" << endl;
}
else
{
    cout << "Key/Value-Paar zu 8 existiert nicht" << endl;
}
}

```

### 8.2.8 Die map Methode insert

Fügt ein oder mehrere Datenelemente vor der angegebenen Position in der **map** ein.

Syntax:

```

pair<iterator, bool> insert (const V& vpair);
void                insert (const V* first, const V* last);
iterator            insert (iterator pos, const V& vpair);

```

Die Methode **insert()** kann verwendet werden um einzelne Key/Value-Paare oder ganze Bereiche in eine **map** einzufügen.

Da eine **map** als Baumstruktur aufgebaut ist, muß (anders als bei **vector** oder **deque**) beim Einfügen von Key/Value-Paaren nichts umkopiert werden, was die Methode **insert()** für **map**-Objekte sehr effizient macht. Der Zeitaufwand für das Einfügen in einem **map**-Objekt ist daher immer konstant.

*Im Gegensatz zum vector und dem deque werden durch das Einfügen keine Iteratoren, Verweise oder Referenzen ungültig.*

Die erste Form der Methode sorgt dafür, daß ein einzelnes Key/Value-Paar in die **map** eingefügt wird. Eingefügt wird die Kopie des Key/Value-Paares, falls in der **map** noch kein Key/Value-Paar zu diesem Schlüssel existiert. Rückgabewert der Methode ist ein **pair**, dessen erstes Element der Iterator auf das eingefügte Element in der **map** ist und dessen zweites Element **true** (Key/Value existierte noch nicht und wurde eingefügt) oder **false** (Key/Value existierte bereits und wurde daher nicht eingefügt) ist.

Mit der zweiten Form der **insert()**-Methode können Bereiche aus einer anderen **map** in die **map** kopiert werden. Der Bereich wird durch Anfangs- und Endadresse festgelegt. Es sollte darauf geachtet werden, daß die Endadresse nicht vor der Anfangsadresse liegen darf und daß beide Adressen zum gleichen **map**-Objekt gehören.

Die dritte **insert()**-Form fügt, wie die erste, einen Schlüssel ein, sofern noch nicht vorhanden. Beginnt jedoch mit der Suche nach der richtigen Position zum Einfügen an der angegebenen Position des Iterators **pos**.



```
//=====
// PROGRAMM: MAP_BSP_09
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    typedef map<int, double, less<int> > mymaptype;
    mymaptype aMap;

    pair<mymaptype::iterator, bool> aRPair;

    aMap [4] = 4.44;
    aMap [5] = 5.55;

    //-----
    // insert-Aufruf
    //-----
    aRPair = aMap.insert (mymaptype::value_type (3, 3.33));
    if (aRPair.second == true)
    {
        cout << "Pair wurde eingefügt" << endl;
    }
    else
    {
        cout << "Pair wurde nicht eingefügt" << endl;
    }

    aRPair = aMap.insert (mymaptype::value_type (3, 33.33));
    if (aRPair.second == true)
    {
        cout << "Pair wurde eingefügt" << endl;
    }
    else
    {
        cout << "Pair wurde nicht eingefügt" << endl;
    }
}
}
```

### 8.2.9 Die map Methode key\_comp

Gibt das Vergleichsobjekt zurück, welches zum Vergleich der Schlüssel herangezogen wird.

Syntax:

```
Compare key_comp () const;
```

### 8.2.10 Die map Methode lower\_bound

Die Methode gibt einen Iterator zurück, welcher auf die erste Position zeigt, an der zu einem gegebenen Schlüssel eingefügt werden kann, ohne daß die durch das Compare-Objekt vorgegebenen Ordnungskriterien verletzt werden.

Der Iterator zeigt auf die Position **end()**, wenn keine gültige Position gefunden werden kann.

Syntax:

```
iterator lower_bound (const K& key);
const_iterator lower_bound (const K& key) const;
```

### 8.2.11 Die map Methode max\_size

Die Methode **max\_size()** gibt die maximale Anzahl an Key/Value-Paaren zurück, die in einer **map** gespeichert werden kann. Der Wert von **max\_size()** ist u.a. vom Typ des Schlüssels abhängig.

Syntax:

```
size_type max_size () const;
```

Der Wert ist zudem abhängig von der Implementation und vom verwendeten Betriebssystem. Bei modernen Compilern und entsprechender Einstellung kann man davon ausgehen, dass 32-Bit Adressen verwendet werden, also theoretisch 4 Gigabytes angesprochen werden können.

Da **map**-Objekte nicht als Block gespeichert sein müssen, wird die Anzahl der Datenelemente sonst nur durch den freien RAM-Speicher begrenzt, die Größe der zu verwaltenden Datenelemente, sowie die Segmentierung (Zersplitterung) des Speichers.

Der Aufruf der Methode verändert den Inhalt der **map** nicht.

```
//=====
// PROGRAMM: MAP_BSP_10
//=====

#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    map<char, double, less<char> > aMap;

    cout << "Map kann " << aMap.max_size ()
         << " Datenelemente aufnehmen";
}

```



### 8.2.12 Die map Methode rbegin

Rückgabe eines **reverse\_iterator** oder **const\_reverse\_iterator**, der auf das letzte in der **map** enthaltene Key/Value-Paar zeigt.

Syntax:

```
reverse_iterator rbegin ();
const_reverse_iterator rbegin () const;
```

Die wohl häufigste Anwendung der Methode **rbegin()** liegt in der Verarbeitung von Daten durch rückwärts gerichtete Schleifen.

Der Aufruf von **rbegin()** verändert den Inhalt der **map** nicht.



```
//=====
// PROGRAMM: MAP_BSP_11
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    map<int, double, less<int> > aMap;
    map<int, double, less<int> >::reverse_iterator aI;

    aMap[3] = 7.0;
    aMap[8] = 1.2;
    aMap[12] = 3.14;

    for (aI=aMap.rbegin(); aI!=aMap.rend(); aI++)
    {
        cout << "Wert: " << (*aI).second << endl;
    }
}
```

### 8.2.13 Die map Methode rend

Rückgabe eines **reverse\_iterator** oder **const\_reverse\_iterator**, der vor das erste in der **map** enthaltene Key/Value-Paar zeigt.

Syntax:

```
reverse_iterator rend ();
const_reverse_iterator rend () const;
```

Die wohl häufigste Anwendung der Methode **rend()** liegt in der Verarbeitung von Daten durch rückwärts gerichtete Schleifen.

Es ist zu beachten, daß **rend()** vor das erste gültige Key/Value-Paar zeigt, alle Schleifen müssen also auf „ungleich“ prüfen (Operator !=). Zeigt ein **iterator** auf **rend()** führt jeder dereferenzierende Zugriff zu einer Exception vom Typ **out\_of\_range**. Wird die Exception nicht wie zweiten Beispiel abgefangen bricht das Programm unkontrolliert ab (Absturz).

Der Aufruf von **rend()** verändert den Inhalt der **map** nicht.

Ein Anwendungsbeispiel ist unter **rbegin()** zu finden. Ein Beispiel, welches die Auslösung einer Exception aufgrund eines illegalen Zugriffs auf **rend()** zeigt, ist im Abschnitt über die Methode **rend()** der Containerklasse **vector** zu finden.

### 8.2.14 Die map Methode size

Gibt die Anzahl von Key/Value-Paaren zurück, die aktuell in der **map** enthalten sind.

Syntax:

```
size_type size () const;
```

Der Aufruf von **size()** verändert den Inhalt der **map** nicht.

```
//=====
// PROGRAMM: MAP_BSP_12
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    map<int, double, less<int> > aMap;
    map<int, double, less<int> >::const_reverse_iterator aI;

    aMap[3] = 7.0;
    aMap[8] = 1.2;
    aMap[12] = 3.14;

    cout << "Anzahl aktuell: " << aMap.size() << endl;
}

```



### 8.2.15 Die map Methode swap

Tauscht die Inhalte zweier **map**-Objekte aus. Die **map**-Objekte müssen von gleichem Typ sein, also gleiche Key/Value- und Compare-Typen enthalten.

Syntax:

```
void swap (map<K, V, C>& m);
```

*Es ist zu beachten, daß somit alle Iteratoren, Verweise und Referenzen ungültig werden, da es sich um Pointer handelt und die Datensätze durch den Tausch im Speicher verschoben werden.*

*Alle Iteratoren, Verweise und Referenzen müssen neu ermittelt werden.*

```
//=====
// PROGRAMM: MAP_BSP_13
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    typedef map<int, double, less<int> > mymaptype;
    mymaptype aMap1;
    mymaptype aMap2;

    aMap1 [1] = 1.1;
    aMap1 [2] = 2.2;
    aMap1 [3] = 3.3;

    aMap2 [11] = 11.11;
    aMap2 [22] = 22.22;
    aMap2 [33] = 33.33;
}

```



```
aMap1.swap (aMap2);

cout << aMap1[11] << endl;
cout << aMap2[1] << endl;
}
```

### 8.2.16 Die map Methode upper\_bound

Die Methode gibt einen Iterator zurück, welcher auf die letzte Position zeigt, an der zu einem gegebenen Schlüssel eingefügt werden kann, ohne daß die durch das Compare-Objekt vorgegebenen Ordnungskriterien verletzt werden. Der Iterator zeigt auf die Position **end()**, wenn keine gültige Position gefunden werden kann.

```
Syntax:
iterator upper_bound (const K& key);
const_iterator upper_bound (const K& key) const;
```

### 8.2.17 Die map Methode value\_comp

Gibt das Vergleichsobjekt zurück, welches zum Vergleich der Schlüssel/Wert-Paare herangezogen wird.

```
Syntax:
map_compare<K, V, C> value_comp () const;
```

## 8.3 Die map Operatoren

Mit den folgenden Operatoren kann auf eine Containerklasse vom Typ **map** zugegriffen werden:

<b>Operatoren der Containerklasse MAP</b>	
<i>Operator</i>	<i>Bedeutung</i>
=	Zuweisung zwischen zwei <b>map</b> -Objekten gleichen Typs
==	Vergleich zwischen zwei <b>map</b> -Objekten gleichen Typs
!=	Vergleich zwischen zwei <b>map</b> -Objekten gleichen Typs
<	Lexikographischer Vergleich zwischen zwei <b>map</b> -Objekten gleichen Typs
>	Lexikographischer Vergleich zwischen zwei <b>map</b> -Objekten gleichen Typs
>=	Lexikographischer Vergleich zwischen zwei <b>map</b> -Objekten gleichen Typs
<=	Lexikographischer Vergleich zwischen zwei <b>map</b> -Objekten gleichen Typs
[]	Zugriff auf ein Key/Value-Paar mit in eckigen Klammern angegebenem Schlüssel

Tabelle 8-1– Operatoren der Containerklasse map

### 8.3.1 Der map Operator =

Der Operator ersetzt den Inhalt eines **map**-Objektes durch den Inhalt des zugewiesenen **map**-Objektes gleichen Typs.

```
Syntax:
```

```
map<K,V,C>& operator= (const map<K,V,C>& m);
```

*Durch die Zuweisung werden alle Iteratoren, Verweise und Referenzen auf dieser Map ungültig. Alle Iteratoren, Verweise und Referenzen müssen neu ermittelt werden.*

```
//=====
// PROGRAMM: MAP_BSP_14
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    typedef map<int, double, less<int> > mymaptype;
    mymaptype aMap1;
    mymaptype aMap2;
    mymaptype::iterator aI;

    aMap1 [1] = 7.0;
    aMap1 [2] = 1.2;
    aMap1 [3] = 3.14;

    aMap2 = aMap1;

    for (aI=aMap1.begin(); aI!=aMap1.end(); aI++)
    {
        cout << "aMap1 Wert: " << (*aI).second << endl;
    }

    for (aI=aMap2.begin(); aI!=aMap2.end(); aI++)
    {
        cout << "aMap2 Wert: " << (*aI).second << endl;
    }
}
```



### 8.3.2 Der map Operator ==

Der Operator prüft, ob die Inhalte des linken und rechten Operanden (**map**-Objekte gleichen Typs) übereinstimmen. Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn beide **map**-Objekte die gleichen Key/Value-Paare in der gleichen Reihenfolge enthalten.

Syntax:

```
bool operator== (const map<K, V, C>& m) const;
```

```
//=====
// PROGRAMM: MAP_BSP_15
//=====

#include <iomanip.h>
#include <iostream.h>
```



```

#include <map>
using namespace std;

void main (void)
{
    typedef map<int, double, less<int> > mymaptype;
    mymaptype aMap1;
    mymaptype aMap2;
    mymaptype::iterator aI;

    aMap1 [1] = 7.0;

    if (aMap1 == aMap2)
    {
        cout << "Die map-Objekte sind gleich" << endl;
    }
    else
    {
        cout << "Die Map-Objekte sind nicht gleich" << endl;
    }

    aMap2 [1] = 7.0;

    if (aMap1 == aMap2)
    {
        cout << "Die Map-Objekte sind gleich" << endl;
    }
    else
    {
        cout << "Die Map-Objekte sind nicht gleich" << endl;
    }
}

```

### 8.3.3 Der map Operator !=

Der Operator prüft, ob die Inhalte des linken und rechten Operanden (**map**-Objekte gleichen Typs) verschieden sind. Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn beide **map**-Objekte unterschiedliche key/Value-Paare oder gleiche Key/Value-Paare in unterschiedlicher Reihenfolge enthalten.

Syntax:

```
bool operator!= (const map<K, V, C>& m) const;
```



```

//=====
// PROGRAMM: MAP_BSP_16
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    typedef map<int, double, less<int> > mymaptype;
    mymaptype aMap1;
    mymaptype aMap2;

```

```

mymaptype::iterator aI;

aMap1 [1] = 7.0;

if (aMap1 != aMap2)
{
    cout << "Die map-Objekte sind ungleich" << endl;
}
else
{
    cout << "Die Map-Objekte sind nicht ungleich" << endl;
}

aMap2 [1] = 7.0;

if (aMap1 != aMap2)
{
    cout << "Die Map-Objekte sind ungleich" << endl;
}
else
{
    cout << "Die Map-Objekte sind nicht ungleich" << endl;
}
}

```

### 8.3.4 Der map Operator <

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch kleiner ist, als der des rechten Operanden (**map**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „<“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch kleiner ist als der rechte Operand.

Syntax:

```
bool operator< (const map<K, V, C>& m) const;
```

```

//=====
// PROGRAMM: MAP_BSP_17
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    map<int, double, less<int> > aMap1;
    map<int, double, less<int> > aMap2;

    aMap1 [1] = 3.14;
    aMap2 [1] = 3.15;

    if (aMap1 < aMap2)
    {
        cout << "aMap1 ist kleiner als aMap2" << endl;
    }
    else
    {
        cout << " aMap1 ist nicht kleiner als aMap2" << endl;
    }
}

```



```
}
}
```

### 8.3.5 Der map Operator >

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch größer ist, als der des rechten Operanden (**map**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „>“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch größer ist als der rechte Operand.

Syntax:

```
bool operator> (const map<K, V, C>& m) const;
```



```
//=====
// PROGRAMM: MAP_BSP_18
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    map<int, double, less<int> > aMap1;
    map<int, double, less<int> > aMap2;

    aMap1 [1] = 3.14;
    aMap2 [1] = 3.15;

    if (aMap1 > aMap2)
    {
        cout << "aMap1 ist größer als aMap2" << endl;
    }
    else
    {
        cout << " aMap1 ist nicht größer als aMap2" << endl;
    }
}
```

### 8.3.6 Der map Operator <=

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch kleiner oder gleich mit dem rechten Operanden ist (**map**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „<=“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch kleiner oder gleich mit dem rechten Operanden ist.

Syntax:

```
bool operator<= (const map<K, V, C>& m) const;
```



```
//=====
// PROGRAMM: MAP_BSP_19
//=====

#include <iomanip.h>
```

```

#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    map<int, double, less<int> > aMap1;
    map<int, double, less<int> > aMap2;

    aMap1 [1] = 3.14;
    aMap2 [1] = 3.15;

    if (aMap1 <= aMap2)
    {
        cout << "aMap1 ist kleiner gleich aMap2" << endl;
    }
    else
    {
        cout << " aMap1 ist nicht kleiner gleich aMap2" << endl;
    }
}

```

### 8.3.7 Der map Operator >=

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch größer oder gleich mit dem rechten Operanden ist (**map**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „>=“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch größer oder gleich mit dem rechten Operanden ist.

Syntax:

```
bool operator>= (const map<K, V, C>& m) const;
```

```

//=====
// PROGRAMM: MAP_BSP_20
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    map<int, double, less<int> > aMap1;
    map<int, double, less<int> > aMap2;

    aMap1 [1] = 3.14;
    aMap2 [1] = 3.15;

    if (aMap1 >= aMap2)
    {
        cout << "aMap1 ist größer gleich aMap2" << endl;
    }
    else
    {
        cout << " aMap1 ist nicht größer gleich aMap2" << endl;
    }
}

```



### 8.3.8 Der map Operator []

Der Operator [] ermöglicht einen schnellen Zugriff auf ein in der **map** gespeichertes Key/Value-Paar. Dazu wird in den Zugriffsklammern [] der gewünschte Key angegeben. Ist ein entsprechendes key/Value-Paar zum Schlüssel vorhanden, so wird eine Referenz auf den gespeicherten Wert (Value) zurückgegeben. Ist ein entsprechendes Key/value-Paar nicht vorhanden, so wird der angegebene Schlüssel mit einem Standardwert verknüpft und dieser zurückgegeben.

Syntax:

```
Value& operator[] (const K& key);
```



```
//=====
// PROGRAMM: MAP_BSP_21
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    map<int, double, less<int> > aMap1;

    aMap1 [1] = 3.14;

    cout << aMap1[1] << endl; // wird gefunden
    cout << aMap1[2] << endl; // wird nicht gefunden
}
```

## 9. Die Containerklasse multimap

Die Containerklasse **multimap** realisiert, wie die **map**, einen assoziativen Behälter (auch Hashtable genannt), welcher Schlüssel/Wertpaare beinhaltet (Key/Value-Paare). Die **multimap** ist durch eine vom Benutzer definierte Vergleichsfunktion (compare-Klasse) sortiert. Im Gegensatz zur einfachen **map** setzt die **multimap** nicht voraus, daß jeder Schlüssel nur ein einziges mal (unique) vorkommen kann.

Die interne Struktur der **multimap** (Hashtable) ermöglicht es, ein gesuchtes Key/Value-Paar sehr effizient zu ermitteln<sup>1</sup>.

Die notwendige Speicherverwaltung wird automatisch vorgenommen und ist so effizient wie möglich realisiert. Wie für **map**-Objekte wird auch für eine **multimap** kein geschlossener Speicherbereich benötigt. Um ein **multimap**-Objekt zu erweitern ist daher kein Umkopieren der gesamten Containerklasse notwendig, da lediglich die Verkettung innerhalb der Baumstruktur geändert werden muß.

Ein wahlfreier Zugriff auf **multimap**-Objekte ist, da es sich um eine 1:n anstatt einer 1:1 Zuordnung handelt, nicht möglich. Der **operator[]** (Zugriff über Indexklammer) ist daher nicht definiert.

Wie **map** weicht **multimap** von den einfachen Container-Klassen **vector**, **deque** und **list** ab. Während diese lediglich eine Klassenangabe erwarten, um die Containerklasse zu definieren, also lediglich die zu verwaltende Klasse benötigen, braucht ein **multimap**-Objekt insgesamt drei Klassenangaben. Die erste erforderliche Angabe verweist auf die Klasse, welche den Key repräsentiert, die zweite auf die Value-Klasse und die dritte Klasse definiert die Vergleichsklasse für die Sortierung:

Die Deklaration eines **multimap**-Objektyps kann durch eine **typedef**-Anweisung weiter vereinfacht werden und hat folgenden grundsätzlichen Aufbau:

```
multimap<Keytyp, Valuetyp, Ordnungsobjekt<Keytyp> > Variablenname;
typedef multimap<Keytyp, Valuetyp, Ordnungsobjekt<Keytyp> > Datentypname;
```

**ACHTUNG:** Das Leerzeichen zwischen den beiden schließenden, spitzen Klammern „> >“ ist zwingend notwendig, da der Compiler zusammenhängende spitze Klammern „>>“ versucht als den entsprechenden Shift-Operator zu verwenden.

### 9.1 Die multimap Constructoren

Die folgende Tabelle faßt die Constructoren der Containerklasse **multimap** zusammen. In den nachstehenden Abschnitten werden die Constructoren einzeln behandelt und mit Beispielen erläutert.

<b>Constructoren der Containerklasse MULTIMAP</b>	
<i>Constructor</i>	<i>Bedeutung</i>
multimap< K, V, C > VarName;	Standard-Constructor, erzeugt ein neues, leeres <b>multimap</b> -Objekt
multimap< K, V, C >	Dieser Constructor erzeugt eine neue,

<sup>1</sup> Die übliche Form der Realisierung ist ein binärer Baum, der in jedem Knoten der Baumstruktur genau ein Key/Value-Paar speichert.

<b>Constructoren der Containerklasse MULTIMAP</b>	
<i>Constructor</i>	<i>Bedeutung</i>
VarName (const Compare& newcompare);	leere <b>multimap</b> , welche die enthaltenen Key/Value-Paare intern anhand der Vergleichsfunktion newcompare ordnet, anstelle der ursprünglich definierten compare-Funktion.
multimap< K, V, C > VarName (V* pFirst, V* pLast);	Dieser Constructor erzeugt eine neue <b>multimap</b> anhand einer bereits bestehenden <b>multimap</b> . Die Pointer <b>pFirst</b> und <b>pLast</b> bezeichnen dabei Anfang- und Endwert eines Bereiches. Das Element, auf welches der Zeiger <b>pLast</b> verweist gehört <i>nicht</i> mehr zum kopierten Bereich.
multimap<K, V, C> VarName (V* pFirst, V* pLast, const Compare& newcompare);	Dieser Constructor erzeugt eine neue <b>multimap</b> anhand einer bereits bestehenden <b>multimap</b> . Die Pointer <b>pFirst</b> und <b>pLast</b> bezeichnen dabei Anfang- und Endwert eines Bereiches. Das Element, auf welches der Zeiger <b>pLast</b> verweist gehört <i>nicht</i> mehr zum kopierten Bereich. Verwendet die übergebene compare-Funktion zur internen Ordnung der Key/Value-Paare.
Multimap< K, V, C > VarName (const multimap <Key, Value, Compare>& m);	Dieser Constructor erzeugt eine neue <b>multimap</b> anhand eines bereits bestehenden <b>multimap</b> -Objektes. Das neue <b>multimap</b> -Objekt ist eine vollständige Kopie des übergebenen <b>multimap</b> -Objektes <b>m</b> .

Tabelle 9-1– Constructoren der Containerklasse multimap

### 9.1.1 Der multimap Standard-Constructor

Der **multimap** Standard-Constructor, erzeugt ein neues, leeres **multimap**-Objekt vom Typ **K, V, C**<sup>2</sup>.

Syntax:  
`multimap<K, V, C> Variablenname;`

Das folgende Beispielprogramm zeigt den Einsatz des Standard-Constructors innerhalb eines Programms:



```
//=====
// PROGRAMM: MULTIMAP_BSP_01
//=====
```

<sup>2</sup> **K** ist der Platzhalter für den in der **multimap** verwalteten Schlüssel (Key), **V** ist der Platzhalter für den in der **multimap** verwendeten Wert (Value) und **C** ist der Platzhalter für die in der **multimap** verwendeten Ordnungsklasse.

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

typedef multimap<char, int, less<char> > MyMapType;

void main (void)
{
    MyMapType aMap;
    multimap<int, long, less<int> > aMap2;
}
```

### 9.1.2 Der multimap Constructor mit alternativer Ordnungsfunktion

Dieser Constructor erzeugt eine neue **multimap** mit der Ordnungsfunktion **C2** anstelle der vordefinierten Ordnungsfunktion **C**.

Syntax:

```
multimap<K, V, C> Variablenname (Compare& C2);
```

Das folgende Beispielprogramm zeigt den Einsatz des Constructors innerhalb eines Programms:

```
//=====
// PROGRAMM: MULTIMAP_BSP_02
//=====

#include <map>
using namespace std;

typedef multimap<char, int, less<char> > MyMapType;

void main (void)
{
    MyMapType aMap(greater<char>);
}
```



### 9.1.3 Der multimap Constructor mit Bereichsangabe

Dieser Constructor erzeugt eine neue **multimap** anhand eines bereits bestehenden **multimap**-Objektes. Die Pointer **pFirst** und **pLast** bezeichnen dabei Anfang- und Endwert eines Bereiches. Das Element, auf welches der Iterator **pLast** verweist gehört *nicht* mehr zum kopierten Bereich – d.h. **pLast** verweist auf das erste Element hinter dem zu kopierenden Bereich. Für jedes der im Bereich befindlichen Elemente wird einmal der Copy-Constructor von **K** und der Copy-Constructor von **V** mit den korrespondierenden Elementen des übergebenen Bereiches aufgerufen. Zur Sortierung der Elemente in der **multimap** wird die Ordnungsfunktion **C** herangezogen.

Syntax:

```
multimap<K, V, C> Variablenname (V* pFirst,
                                V* pLast);
```

Anhand der übergebenen Zeiger kann der Constructor die benötigte Anzahl an Elementen ermitteln und eine entsprechend großen Block reservieren.

### 9.1.4 Der multimap Constructor mit Bereichsangabe und alternativer Ordnungsfunktion

Dieser Constructor erzeugt eine neue **multimap** anhand eines bereits bestehenden **multimap**-Objektes. Die Pointer **pFirst** und **pLast** bezeichnen dabei Anfang- und Endwert eines Bereiches. Das Element, auf welches der Iterator **pLast** verweist gehört *nicht* mehr zum kopierten Bereich – d.h. **pLast** verweist auf das erste Element hinter dem zu kopierenden Bereich. Der Constructor erzeugt eine neue **multimap**, die jedoch die Ordnungsfunktion **C2** anstelle der vordefinierten Ordnungsfunktion **C** nutzt, d.h. der kopierte Bereich ist in der **multimap**-Kopie anders sortiert als im Original.

Für jedes der im Bereich befindlichen Elemente wird einmal der Copy-Constructor von **K** und der Copy-Constructor von **V** mit den korrespondierenden Elementen des übergebenen Bereiches aufgerufen.

Syntax:

```
multimap<K, V, C> Variablenname (V* pFirst, V* pLast,
                                Compare& C2);
```

Anhand der übergebenen Zeiger kann der Constructor die benötigte Anzahl an Elementen ermitteln und eine entsprechend großen Block reservieren.

### 9.1.5 Der multimap Copy-Constructor

Dieser Constructor erzeugt eine neue **multimap** anhand eines bereits bestehenden **multimap**-Objektes. Das neue **multimap**-Objekt ist eine vollständige Kopie des übergebenen Objektes M.

Syntax:

```
multimap<K, V, C> Variablenname
    (const multimap<K, V, C>& M);
```

Das folgende Beispielprogramm zeigt, wie der Copy-Constructor verwendet werden kann:



```
//=====
// PROGRAMM: MULTIMAP_BSP_03
//=====

#include <map>
#include <iostream.h>
using namespace std;

void main (void)
{
    multimap<char, int, less<char> > aMyMap;
    typedef pair<char, int> aValue_typ;

    aValue_typ aEins ('A', 256);
    aValue_typ aZwei ('B', 17);
    aValue_typ aDrei ('A', 255);
    aValue_typ aVier ('C', 3);
    aValue_typ aFuenf('D', 170);

    aMyMap.insert(aEins);
```

```

aMyMap.insert(aZwei);
aMyMap.insert(aDrei);
aMyMap.insert(aVier);
aMyMap.insert(aFuenf);

// die Multimap kopieren
multimap<char, int, less<char> > aMyMap2 (aMyMap);

cout << "aMyMap.begin() = " << (*aMyMap.begin()).second
      << endl;
cout << "aMyMap2.begin() = " << (*aMyMap2.begin()).second
      << endl;
}

```

## 9.2 Die multimap Methoden

Die folgende Tabelle faßt die Methoden der **multimap** Containerklasse zusammen. In den nachstehenden Abschnitten werden die Methoden einzeln behandelt und mit Beispielen erläutert.

<b>Methoden der Containerklasse MULTIMAP</b>	
<i> Methode </i>	<i> Bedeutung </i>
begin	Rückgabe eines <b>iterator</b> , der auf das erste in der <b>multimap</b> enthaltene Element (Key/Value-Paar) zeigt
count	Gibt die Anzahl aller Key/Value-Paare zurück, deren Schlüssel mit einem übergebenen Key übereinstimmt
empty	Gibt den Wert <b>true</b> zurück, wenn die <b>multimap</b> keine Datenelemente enthält
end	Rückgabe eines <b>iterator</b> , der hinter das letzte Key/Value-Paar in der <b>multimap</b> zeigt
equal_range	Diese Methode ermittelt ein Iteratoren-Paar, dessen erstes Element dem <b>lower_bound()</b> und dessen zweites Element dem <b>upper_bound</b> entspricht
erase	Löscht ein Key/Value-Paar an der angegebenen Position aus der <b>multimap</b>
find	Gibt, falls ein Key/Value-Paar zum übergebenen Key existiert, einen Iterator auf dieses Element zurück, sonst einen Iterator der auf <b>end()</b> zeigt
insert	Fügt die Kopie eines Key/Value-Paares oder eines Key/Value-Bereiches in die <b>multimap</b> ein.
key_comp	Gibt das Vergleichsobjekt zurück, welches zum Vergleich der Schlüssel verwendet wird.
lower_bound	Gibt einen Iterator zurück, der auf die erste Position in der <b>multimap</b> zeigt, an der ein Key/Value-Paar eingefügt werden kann, ohne daß die Ordnung der Klasse <b>C</b> verletzt wird. Der Iterator zeigt auf <b>end()</b> wenn eine solche Position nicht gefunden werden konnte.
max_size	Gibt die maximale Anzahl an Datenelementen zurück, welche die <b>multimap</b> enthalten kann
rbegin	Rückgabe eines <b>reverse_iterator</b> , der auf das letzte in der <b>multimap</b> enthaltene Key/value-Paar zeigt
rend	Rückgabe eines <b>reverse_iterator</b> , der vor das erste in der <b>multimap</b> enthaltene Key/Value-Paar zeigt

<b>Methoden der Containerklasse MULTIMAP</b>	
<i>Methode</i>	<i>Bedeutung</i>
size	Gibt die Anzahl von Key/Value-Paaren zurück, die aktuell in der <b>multimap</b> enthalten sind
swap	Tauscht den Inhalt zweier <b>multimap</b> -Objekte aus
upper_bound	Gibt einen Iterator zurück, der auf die letzte Position in der <b>multimap</b> zeigt, an der ein Key/Value-Paar eingefügt werden kann, ohne daß die Ordnung der Klasse <b>C</b> verletzt wird. Der Iterator zeigt auf <b>end()</b> wenn eine solche Position nicht gefunden werden konnte.
value_comp	Gibt das Vergleichsobjekt zurück, welches zum Vergleich der Schlüssel/Wert-Paare verwendet wird.

Tabelle 9-1– Methoden der Containerklasse multimap

### 9.2.1 Die multimap Methode begin

Rückgabe eines **iterator** oder **const\_iterator**, der auf das erste in der **multimap** enthaltene Element zeigt.

Syntax:

```
iterator begin ();
const_iterator begin () const;
```

Die wohl häufigste Anwendung der Methode **begin()** liegt in der Verarbeitung von Daten durch Schleifen.

Der Aufruf von **begin()** verändert den Inhalt der **multimap** nicht.



```
//=====
// PROGRAMM: MULTIMAP_BSP_04
//=====

#include <map>
#include <iostream.h>
using namespace std;

void main (void)
{
    multimap<char, int, less<char> > aMyMap;
    multimap<char, int, less<char> >::iterator aI;
    typedef pair<char, int> aValue_typ;

    aValue_typ aEins ('A', 256);
    aValue_typ aZwei ('B', 17);
    aValue_typ aDrei ('A', 255);

    aMyMap.insert(aEins);
    aMyMap.insert(aZwei);
    aMyMap.insert(aDrei);

    // die Multimap kopieren
    multimap<char, int, less<char> > aMyMap2 (aMyMap);

    cout << "aMyMap.begin() = " << (*aMyMap.begin()).second
         << endl;
    cout << "aMyMap2.begin() = " << (*aMyMap2.begin()).second
         << endl;
}
```

## 9.2.2 Die multimap Methode count

Rückgabe der Anzahl der Key/Value-Paare, deren Schlüssel mit einem übergebenen Schlüssel übereinstimmt.

Syntax:

```
size_type count (const K& key) const;
```

Die wichtigste Bedeutung der Methode count() liegt bei der einfachen **multimap** darin festzustellen, ob ein oder mehrere Key/Value-Paare zu einem bestimmten Schlüssel existiert oder nicht. Die Tatsache ob Key/Value-Paare existieren kann zwar auch über **find()** festgestellt werden, da **find()** aber einen Iterator zurückgibt, muß dieser erst wieder mit einem weiteren Vergleich überprüft werden.

Gegenüber der einfachen **map**-Containerklasse wird in der **multimap** die Anzahl der Key/Value-Paare ist natürlich auch für die Verarbeitung der zu einem Schlüssel gespeicherten Werte benötigt, die nur über Schleifen verarbeitet werden können.

```
//=====
// PROGRAMM: MULTIMAP_BSP_05
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    multimap<int, double, less<int> > aMap;
    multimap<int, double, less<int> >::iterator aI;
    typedef pair<int, double> aValue_typ;

    aValue_typ aEins ( 3, 7.0);
    aValue_typ aZwei ( 8, 1.2);
    aValue_typ aDrei (12, 3.14);

    aMap.insert(aEins);
    aMap.insert(aZwei);
    aMap.insert(aDrei);

    if (aMap.count(7))
    {
        cout << "Key/Value-Paar zu 7 existiert" << endl;
    }
    else
    {
        cout << "Key/Value-Paar zu 7 existiert nicht" << endl;
    }

    if (aMap.count(8))
    {
        cout << "Key/Value-Paar zu 8 existiert" << endl;
    }
    else
    {
        cout << "Key/Value-Paar zu 8 existiert nicht" << endl;
    }
}
```



}

### 9.2.3 Die multimap Methode empty

Die Methode **empty()** prüft, ob die **multimap** Datenelemente enthält oder nicht. Sind Datenelemente enthalten (dies entspricht einem Rückgabewert von **size()** größer als Null), so wird der Wert **false** zurückgegeben, sind keine Datenelemente vorhanden ist der Rückgabewert **true**.

Der Aufruf von **empty()** verändert den Inhalt der **multimap** nicht.

Syntax:

```
bool empty () const;
```



```
//=====
// PROGRAMM: MULTIMAP_BSP_06
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    multimap<int, double, less<int> > aMap;
    typedef pair<int, double> aValue_typ;

    if (aMap.empty())
    {
        cout << "Multimap ist leer" << endl;
    }
    else
    {
        cout << "Multimap enthält Key/Value-Paare" << endl;
    }

    aValue_typ aEins ( 3, 7.0);
    aValue_typ aZwei ( 8, 1.2);
    aValue_typ aDrei (12, 3.14);

    aMap.insert(aEins);
    aMap.insert(aZwei);
    aMap.insert(aDrei);

    if (aMap.empty())
    {
        cout << "Multimap ist leer" << endl;
    }
    else
    {
        cout << "Multimap enthält Key/Value-Paare" << endl;
    }
}
```

### 9.2.4 Die multimap Methode end

Rückgabe eines **iterator** oder **const\_iterator**, der *hint* das letzte in der **multimap** enthaltene Key/Value-Paar zeigt.

Syntax:

```
iterator end ();
const_iterator end () const;
```

Die wohl häufigste Anwendung der Methode **end()** liegt in der Verarbeitung von Daten durch Schleifen.

Es ist zu beachten, daß **end()** hinter das letzte gültige Key/value-Paar zeigt, alle Schleifen müssen also auf „ungleich“ (Operator **!=**) oder „echt kleiner als“ prüfen (Operator **<**) und nicht auf „kleiner oder gleich“ (Operator **<=**). Zeigt ein **iterator** auf **end()** führt jeder dereferenzierende Zugriff zu einer Exception vom Typ **out\_of\_range**. Wird die Exception nicht abgefangen (wie zweiten Beispiel), bricht das Programm unkontrolliert ab (Absturz).

Der Aufruf von **end()** verändert den Inhalt der **multimap** nicht.

```
//=====
// PROGRAMM: MULTIMAP_BSP_07
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    multimap<int, double, less<int> > aMap;
    multimap<int, double, less<int> >::iterator aI;
    typedef pair<int, double> aValue_typ;

    aValue_typ aEins ( 3, 7.0);
    aValue_typ aZwei ( 8, 1.2);
    aValue_typ aDrei (12, 3.14);

    aMap.insert(aEins);
    aMap.insert(aZwei);
    aMap.insert(aDrei);

    for (aI=aMap.begin(); aI!=aMap.end(); aI++)
    {
        cout << "Wert: " << (*aI).second << endl;
    }
}
```



Ein Beispiel, welches die Auslösung einer Exception aufgrund eines illegalen Zugriffs auf **end()** zeigt, ist im Abschnitt zur Methode **end()** bei der Containerklasse **vector** aufgeführt.

### 9.2.5 Die multimap Methode **equal\_range**

Gibt ein Iteratoren-Paar zurück, dessen erstes Element der Rückgabe von **lower\_bound()** (erste Stelle an der ein Key/Value-Paar mit angegebenem Key eingefügt werden kann) und dessen zweites Element der Rückgabe von **upper\_bound()** (letzte Stelle, an der ein Key/Value-Paar mit angegebenem Key eingefügt werden kann) entspricht.

Syntax:

```
pair<iterator, iterator> equal_range(const K& key);
```

```
pair<const_iterator, const_iterator>  
equal_range(const K& key) const;
```

### 9.2.6 Die multimap Methode erase

Löscht ein oder mehrere Datenelemente an der angegebenen Position aus der **multimap**.

Syntax:

```
void erase(iterator toDel);  
void erase(iterator First, iterator Last);  
size_type erase (const K& key);
```

Die Methode **erase()** kann verwendet werden, um Datensätze oder ganze Bereiche aus der **multimap** zu löschen.

*Im Gegensatz zum vector und dem deque werden durch das Löschen in der multimap nur die Iteratoren, Verweise und Referenzen auf Datensätze im gelöschten Bereich ungültig.*

*Da es sich bei der multimap nicht um ein Array handelt (welches als geschlossener Block gespeichert werden muß), gibt es keine Notwendigkeit, das multimap-Objekt im Speicher zu verschieben.*

*Iteratoren, Verweise und Referenzen, die nicht auf einen durch die Methode gelöschten Bereich zeigen, bleiben daher gültig.*

Die erste Form der Methode sorgt dafür, daß ein einzelner Datensatz aus der **multimap** gelöscht und der Destructor des Key/Value-Paares für diesen Datensatz aufgerufen wird.

Bei der zweiten Syntaxform wird ein zu löschender Bereich angegeben, für jedes zu löschende Key/Value-Paar wird der Destructor aufgerufen. Die Angabe erfolgt durch zwei Iteratoren, die auf das erste und das letzte zu löschende Element zeigen.

Es ist besonders bei den Bereichsangaben zu beachten, daß nicht versehentlich eine **out\_of\_range** Exception erzeugt wird.

Ein Beispiel, welches die Auslösung einer Exception aufgrund eines illegalen Zugriffs durch **erase()** zeigt, ist im Abschnitt zur Methode **erase()** bei der Containerklasse **vector** aufgeführt.

Die dritte Form der **erase**-Methode löscht alle Key/Value-Paare, deren Schlüssel mit einem angegebenen Key übereinstimmen und gibt die Anzahl der gelöschten Datensätze zurück.

### 9.2.7 Die multimap Methode find

Die Methode **find()** gibt einen Iterator auf ein in der **multimap** gespeichertes Key/Value-Paar zu einem vorgegebenem Schlüssel zurück. Ist ein entsprechendes Key/Value-Paar in der **multimap** nicht enthalten, so wird ein Iterator auf **end()** zurückgegeben, der hinter den letzten gültigen Eintrag zeigt.

Der Aufruf von **find()** selbst verändert den Inhalt der **multimap** nicht.

Syntax:

```
iterator find (const K& key) const;
```

Das nachfolgende Beispielprogramm zeigt den Einsatz der **find()**-Methode in einem Programm:

```
//=====
// PROGRAMM: MULTIMAP_BSP_08
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    multimap<int, double, less<int> > aMap;
    multimap<int, double, less<int> >::iterator aI;
    typedef pair<int, double> aValue_typ;

    aValue_typ aEins ( 3, 7.0);
    aValue_typ aZwei ( 8, 1.2);
    aValue_typ aDrei (12, 3.14);

    aMap.insert(aEins);
    aMap.insert(aZwei);
    aMap.insert(aDrei);

    aI = aMap.find(7);
    if (aI != aMap.end())
    {
        cout << "Key/Value-Paar zu 7 existiert" << endl;
    }
    else
    {
        cout << "Key/Value-Paar zu 7 existiert nicht" << endl;
    }

    aI = aMap.find(8);
    if (aI != aMap.end())
    {
        cout << "Key/Value-Paar zu 8 existiert" << endl;
    }
    else
    {
        cout << "Key/Value-Paar zu 8 existiert nicht" << endl;
    }
}
```



### 9.2.8 Die multimap Methode insert

Fügt ein oder mehrere Datenelemente vor der angegebenen Position in der **multimap** ein.

Syntax:

```
pair<iterator, bool> insert (const V& vpair);
void                insert (const V* first, const V* last);
iterator            insert (iterator pos, const V& vpair);
```

Die Methode **insert()** kann verwendet werden um einzelne Key/Value-Paare oder ganze Bereiche in eine **multimap** einzufügen.

Da eine **multimap** als Baumstruktur aufgebaut ist, muß (anders als bei **vector** oder **deque**) beim Einfügen von Key/Value-Paaren nichts umkopiert werden, was die Methode **insert()** für **multimap**-Objekte sehr effizient macht. Der Zeitaufwand für das Einfügen in einem **multimap**-Objekt ist daher immer konstant.

*Im Gegensatz zum vector und dem deque werden durch das Einfügen keine Iteratoren, Verweise oder Referenzen ungültig.*

Die erste Form der Methode sorgt dafür, daß ein einzelnes Key/Value-Paar in die **multimap** eingefügt wird. Eingefügt wird die Kopie des Key/Value-Paares. Rückgabewert der Methode ist ein **Iterator**, der auf das eingefügte Element in der **multimap** zeigt.

Mit der zweiten Form der **insert()**-Methode können Bereiche aus einer anderen **multimap** in die **multimap** kopiert werden. Der Bereich wird durch Anfangs- und Endadresse festgelegt. Es sollte darauf geachtet werden, daß die Endadresse nicht vor der Anfangsadresse liegen darf und daß beide Adressen zum gleichen **multimap**-Objekt gehören.

Die dritte **insert()**-Form fügt, wie die erste, einen Schlüssel ein. Beginnt jedoch mit der Suche nach der richtigen Position zum Einfügen an der angegebenen Position des Iterators **pos**.



```
//=====
// PROGRAMM: MULTIMAP_BSP_09
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    typedef multimap<int, double, less<int> > mymaptype;
    multimap<int, double, less<int> >::iterator aI;
    typedef pair<int, double> aValue_typ;
    mymaptype aMap;

    aValue_typ a1 (4, 4.44);
    aValue_typ a2 (5, 5.55);
    aValue_typ a3 (6, 6.66);

    aMap.insert(a1);
    aMap.insert(a1);
    aMap.insert(a1);
    aMap.insert(a2);
    aMap.insert(a3);

    for (aI=aMap.begin(); aI!=aMap.end(); aI++)
    {
```

```

    cout << "Wert: " << (*aI).second << endl;
}
}

```

### 9.2.9 Die multimap Methode key\_comp

Gibt das Vergleichsobjekt zurück, welches zum Vergleich der Schlüssel herangezogen wird.

Syntax:

```
Compare key_comp () const;
```

### 9.2.10 Die multimap Methode lower\_bound

Die Methode gibt einen Iterator zurück, welcher auf die erste Position zeigt, an der zu einem gegebenen Schlüssel eingefügt werden kann, ohne daß die durch das Compare-Objekt vorgegebenen Ordnungskriterien verletzt werden. Der Iterator zeigt auf die Position **end()**, wenn keine gültige Position gefunden werden kann.

Syntax:

```
iterator lower_bound (const K& key);
const_iterator lower_bound (const K& key) const;
```

### 9.2.11 Die multimap Methode max\_size

Die Methode **max\_size()** gibt die maximale Anzahl an Key/Value-Paaren zurück, die in einer **multimap** gespeichert werden kann. Der Wert von **max\_size()** ist u.a. vom Typ des Schlüssels abhängig.

Syntax:

```
size_type max_size () const;
```

Der Wert ist zudem abhängig von der Implementation und vom verwendeten Betriebssystem. Bei modernen Compilern und entsprechender Einstellung kann man davon ausgehen, das 32-Bit Adressen verwendet werden, also theoretisch 4 Gigabytes angesprochen werden können.

Da **multimap**-Objekte nicht als Block gespeichert sein müssen, wird die Anzahl der Datenelemente sonst nur durch den freien RAM-Speicher begrenzt, die Größe der zu verwaltenden Datenelemente, sowie die Segmentierung (Zersplitterung) des Speichers.

Der Aufruf der Methode verändert den Inhalt der **multimap** nicht.

```

//=====
// PROGRAMM: MULTIMAP_BSP_10
//=====

#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    multimap<char, double, less<char> > aMap;

    cout << "Multimap kann " << aMap.max_size ()
         << " Datenelemente aufnehmen";
}

```



}

### 9.2.12 Die multimap Methode rbegin

Rückgabe eines **reverse\_iterator** oder **const\_reverse\_iterator**, der auf das letzte in der **multimap** enthaltene Key/Value-Paar zeigt.

Syntax:

```
reverse_iterator rbegin ();
const_reverse_iterator rbegin () const;
```

Die wohl häufigste Anwendung der Methode **rbegin()** liegt in der Verarbeitung von Daten durch rückwärts gerichtete Schleifen.

Der Aufruf von **rbegin()** verändert den Inhalt der **multimap** nicht.



```
//=====
// PROGRAMM: MULTIMAP_BSP_11
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    multimap<int, double, less<int> > aMap;
    multimap<int, double, less<int> >::reverse_iterator aI;
    typedef pair<int, double> aValue_typ;

    aValue_typ aEins ( 3, 7.0);
    aValue_typ aZwei ( 8, 1.2);
    aValue_typ aDrei (12, 3.14);

    aMap.insert(aEins);
    aMap.insert(aZwei);
    aMap.insert(aDrei);

    for (aI=aMap.rbegin(); aI!=aMap.rend(); aI++)
    {
        cout << "Key: " << (*aI).first
              << " - Wert: " << (*aI).second << endl;
    }
}
```

### 9.2.13 Die multimap Methode rend

Rückgabe eines **reverse\_iterator** oder **const\_reverse\_iterator**, der vor das erste in der **multimap** enthaltene Key/Value-Paar zeigt.

Syntax:

```
reverse_iterator rend ();
const_reverse_iterator rend () const;
```

Die wohl häufigste Anwendung der Methode **rend()** liegt in der Verarbeitung von Daten durch rückwärts gerichtete Schleifen.

Es ist zu beachten, daß **rend()** vor das erste gültige Key/Value-Paar zeigt, alle Schleifen müssen also auf „ungleich“ prüfen (Operator !=). Zeigt ein

**iterator** auf **rend()** führt jeder dereferenzierende Zugriff zu einer Exception vom Typ **out\_of\_range**. Wird die Exception nicht wie zweiten Beispiel abgefangen bricht das Programm unkontrolliert ab (Absturz).

Der Aufruf von **rend()** verändert den Inhalt der **multimap** nicht.

Ein Anwendungsbeispiel ist unter **rbegin()** zu finden. Ein Beispiel, welches die Auslösung einer Exception aufgrund eines illegalen Zugriffs auf **rend()** zeigt, ist im Abschnitt über die Methode **rend()** der Containerklasse **vector** zu finden.

### 9.2.14 Die multimap Methode size

Gibt die Anzahl von Key/Value-Paaren zurück, die aktuell in der **multimap** enthalten sind.

Syntax:

```
size_type size () const;
```

Der Aufruf von **size()** verändert den Inhalt der **multimap** nicht.

```
//=====
// PROGRAMM: MULTIMAP_BSP_12
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    multimap<int, double, less<int> > aMap;
    multimap<int, double, less<int> >::const_reverse_iterator aI;
    typedef pair<int, double> aValue_typ;

    aValue_typ aEins ( 3, 7.0);
    aValue_typ aZwei ( 8, 1.2);
    aValue_typ aDrei (12, 3.14);

    aMap.insert(aEins);
    aMap.insert(aZwei);
    aMap.insert(aDrei);

    cout << "Anzahl aktuell: " << aMap.size() << endl;
}
```



### 9.2.15 Die multimap Methode swap

Tauscht die Inhalte zweier **multimap**-Objekte aus. Die **multimap**-Objekte müssen von gleichem Typ sein, also gleiche Key/Value- und Compare-Typen enthalten.

Syntax:

```
void swap (multimap<K, V, C>& m);
```

*Es ist zu beachten, daß somit alle Iteratoren, Verweise und Referenzen ungültig werden, da es sich um Pointer*

*handelt und die Datensätze durch den Tausch im Speicher verschoben werden.  
Alle Iteratoren, Verweise und Referenzen müssen neu ermittelt werden.*



```
//=====
// PROGRAMM: MULTIMAP_BSP_13
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    typedef multimap<int, double, less<int> > mymaptype;
    multimap<int, double, less<int> >::iterator aI;
    typedef pair<int, double> aValue_typ;

    mymaptype aMap1;
    mymaptype aMap2;

    aValue_typ a1 ( 1, 1.1);
    aValue_typ a2 ( 2, 2.2);
    aValue_typ a3 ( 3, 3.3);
    aValue_typ a4 (11, 11.11);
    aValue_typ a5 (22, 22.22);
    aValue_typ a6 (33, 33.33);

    aMap1.insert(a1);
    aMap1.insert(a2);
    aMap1.insert(a3);

    aMap2.insert(a4);
    aMap2.insert(a5);
    aMap2.insert(a6);

    aMap1.swap (aMap2);

    for (aI=aMap1.begin(); aI!=aMap1.end(); aI++)
    {
        cout << "aMap1 - Key: " << (*aI).first
              << " - Wert: " << (*aI).second << endl;
    }

    for (aI=aMap2.begin(); aI!=aMap2.end(); aI++)
    {
        cout << "aMap2 - Key: " << (*aI).first
              << " - Wert: " << (*aI).second << endl;
    }
}
```

### 9.2.16 Die multimap Methode upper\_bound

Die Methode gibt einen Iterator zurück, welcher auf die letzte Position zeigt, an der zu einem gegebenen Schlüssel eingefügt werden kann, ohne daß die durch das Compare-Objekt vorgegebenen Ordnungskriterien verletzt werden. Der Iterator zeigt auf die Position **end()**, wenn keine gültige Position gefunden werden kann.

Syntax:

```
iterator upper_bound (const K& key);
const_iterator upper_bound (const K& key) const;
```

### 9.2.17 Die multimap Methode value\_comp

Gibt das Vergleichsobjekt zurück, welches zum Vergleich der Schlüssel/Wert-Paare herangezogen wird.

Syntax:

```
map_compare<K, V, C> value_comp () const;
```

## 9.3 Die multimap Operatoren

Mit den folgenden Operatoren kann auf eine Containerklasse vom Typ **multimap** zugegriffen werden:

<b>Operatoren der Containerklasse MULTIMAP</b>	
<i>Operator</i>	<i>Bedeutung</i>
=	Zuweisung zwischen zwei <b>multimap</b> -Objekten gleichen Typs
==	Vergleich zwischen zwei <b>multimap</b> -Objekten gleichen Typs
!=	Vergleich zwischen zwei <b>multimap</b> -Objekten gleichen Typs
<	Lexikographischer Vergleich zwischen zwei <b>multimap</b> -Objekten gleichen Typs
>	Lexikographischer Vergleich zwischen zwei <b>multimap</b> -Objekten gleichen Typs
>=	Lexikographischer Vergleich zwischen zwei <b>multimap</b> -Objekten gleichen Typs
<=	Lexikographischer Vergleich zwischen zwei <b>multimap</b> -Objekten gleichen Typs

Tabelle 9-1– Operatoren der Containerklasse multimap

### 9.3.1 Der multimap Operator =

Der Operator ersetzt den Inhalt eines **multimap**-Objektes durch den Inhalt des zugewiesenen **multimap**-Objektes gleichen Typs.

Syntax:

```
multimap<K,V,C>& operator= (const multimap<K,V,C>& m);
```

*Durch die Zuweisung werden alle Iteratoren, Verweise und Referenzen auf dieser Multimap ungültig. Alle Iteratoren, Verweise und Referenzen müssen neu ermittelt werden.*

```
//=====
// PROGRAMM: MULTIMAP_BSP_14
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>
```



```

using namespace std;

void main (void)
{
    typedef multimap<int, double, less<int> > mymaptype;
    typedef pair<int, double> aValue_typ;
    mymaptype aMap1;
    mymaptype aMap2;
    mymaptype::iterator aI;

    aValue_typ aEins (1, 7.0);
    aValue_typ aZwei (2, 1.2);
    aValue_typ aDrei (3, 3.14);

    aMap1.insert(aEins);
    aMap1.insert(aZwei);
    aMap1.insert(aDrei);

    aMap2 = aMap1;

    for (aI=aMap1.begin(); aI!=aMap1.end(); aI++)
    {
        cout << "aMap1 - Key: " << (*aI).first
              << " - Wert: " << (*aI).second << endl;
    }

    for (aI=aMap2.begin(); aI!=aMap2.end(); aI++)
    {
        cout << "aMap2 - Key: " << (*aI).first
              << " - Wert: " << (*aI).second << endl;
    }
}

```

### 9.3.2 Der multimap Operator ==

Der Operator prüft, ob die Inhalte des linken und rechten Operanden (**multimap**-Objekte gleichen Typs) übereinstimmen. Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn beide **multimap**-Objekte die gleichen Key/Value-Paare in der gleichen Reihenfolge enthalten.

Syntax:

```
bool operator==(const multimap<K, V, C>& m) const;
```



```

//=====
// PROGRAMM: MULTIMAP_BSP_15
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    typedef multimap<int, double, less<int> > mymaptype;
    typedef pair<int, double> aValue_typ;

    mymaptype aMap1;
    mymaptype aMap2;

```

```

mymaptype::iterator aI;

aValue_typ aEins (1, 7.0);

aMap1.insert(aEins);

if (aMap1 == aMap2)
{
    cout << "Die Multimap-Objekte sind gleich" << endl;
}
else
{
    cout << "Die Multimap-Objekte sind nicht gleich" << endl;
}

aMap2.insert(aEins);

if (aMap1 == aMap2)
{
    cout << "Die Multimap-Objekte sind gleich" << endl;
}
else
{
    cout << "Die Multimap-Objekte sind nicht gleich" << endl;
}
}

```

### 9.3.3 Der multimap Operator !=

Der Operator prüft, ob die Inhalte des linken und rechten Operanden (**multimap**-Objekte gleichen Typs) verschieden sind. Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn beide **multimap**-Objekte unterschiedliche key/Value-Paare oder gleiche Key/Value-Paare in unterschiedlicher Reihenfolge enthalten.

Syntax:

```
bool operator!=(const multimap<K, V, C>& m) const;
```

```

//=====
// PROGRAMM: MULTIMAP_BSP_16
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    typedef multimap<int, double, less<int> > mymaptype;
    typedef pair<int, double> aValue_typ;

    mymaptype aMap1;
    mymaptype aMap2;
    mymaptype::iterator aI;

    aValue_typ aEins (1, 7.0);

```



```

aMap1.insert(aEins);

if (aMap1 != aMap2)
{
    cout << "Die Multimap-Objekte sind ungleich" << endl;
}
else
{
    cout << "Die Multimap-Objekte sind nicht ungleich" << endl;
}

aMap2.insert(aEins);

if (aMap1 != aMap2)
{
    cout << "Die Multimap-Objekte sind ungleich" << endl;
}
else
{
    cout << "Die Multimap-Objekte sind nicht ungleich" << endl;
}
}
    
```

### 9.3.4 Der multimap Operator <

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch kleiner ist, als der des rechten Operanden (**multimap**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „<“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch kleiner ist als der rechte Operand.

Syntax:

```
bool operator< (const multimap<K, V, C>& m) const;
```



```

//=====
// PROGRAMM: MULTIMAP_BSP_17
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    multimap<int, double, less<int> > aMap1;
    multimap<int, double, less<int> > aMap2;
    typedef pair<int, double> aValue_typ;

    aValue_typ a1 (1, 3.14);
    aValue_typ a2 (1, 3.15);

    aMap1.insert(a1);
    aMap2.insert(a2);

    if (aMap1 < aMap2)
    {
        cout << "aMap1 ist kleiner als aMap2" << endl;
    }
    else
    
```

```

    {
        cout << " aMap1 ist nicht kleiner als aMap2" << endl;
    }
}

```

### 9.3.5 Der multimap Operator >

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch größer ist, als der des rechten Operanden (**multimap**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „>“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch größer ist als der rechte Operand.

Syntax:

```
bool operator> (const multimap<K, V, C>& m) const;
```

```

//=====
// PROGRAMM: MULTIMAP_BSP_18
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    multimap<int, double, less<int> > aMap1;
    multimap<int, double, less<int> > aMap2;
    typedef pair<int, double> aValue_typ;

    aValue_typ a1 (1, 3.14);
    aValue_typ a2 (1, 3.15);

    aMap1.insert(a1);
    aMap2.insert(a2);

    if (aMap1 > aMap2)
    {
        cout << "aMap1 ist größer als aMap2" << endl;
    }
    else
    {
        cout << " aMap1 ist nicht größer als aMap2" << endl;
    }
}

```



### 9.3.6 Der multimap Operator <=

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch kleiner oder gleich mit dem rechten Operanden ist (**multimap**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „<=“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch kleiner oder gleich mit dem rechten Operanden ist.

Syntax:

```
bool operator<= (const multimap<K, V, C>& m) const;
```



```
//=====
// PROGRAMM: MULTIMAP_BSP_19
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    multimap<int, double, less<int> > aMap1;
    multimap<int, double, less<int> > aMap2;
    typedef pair<int, double> aValue_typ;

    aValue_typ a1 (1, 3.14);
    aValue_typ a2 (1, 3.15);

    aMap1.insert(a1);
    aMap2.insert(a2);

    if (aMap1 <= aMap2)
    {
        cout << "aMap1 ist kleiner gleich aMap2" << endl;
    }
    else
    {
        cout << " aMap1 ist nicht kleiner gleich aMap2" << endl;
    }
}

```

### 9.3.7 Der multimap Operator >=

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch größer oder gleich mit dem rechten Operanden ist (**multimap**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „>=“ aufgerufen wird. Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch größer oder gleich mit dem rechten Operanden ist.

Syntax:

```
bool operator>= (const multimap<K, V, C>& m) const;
```



```
//=====
// PROGRAMM: MULTIMAP_BSP_20
//=====

#include <iomanip.h>
#include <iostream.h>
#include <map>
using namespace std;

void main (void)
{
    multimap<int, double, less<int> > aMap1;
    multimap<int, double, less<int> > aMap2;
    typedef pair<int, double> aValue_typ;

    aValue_typ a1 (1, 3.14);

```

```
aValue_typ a2 (1, 3.15);

aMap1.insert(a1);
aMap2.insert(a2);

if (aMap1 >= aMap2)
{
    cout << "aMap1 ist größer gleich aMap2" << endl;
}
else
{
    cout << " aMap1 ist nicht größer gleich aMap2" << endl;
}
}
```

## 10 Die Adapterklasse stack

Die Adapterklasse **stack** ermöglicht es (laut STL-Definition), jede Containerklasse **P**, welche die Methoden **push\_back()** und **pop\_back()** implementiert in Form einer **LIFO**- bzw. **FILO**-Datenstruktur<sup>1</sup> zu verwenden.

Da **stack**-Objekte immer nur den Zugriff auf das „oberste“ Datenelement erlaubt, werden keinerlei Iteratoren unterstützt.

Anstelle von **push\_back()** und **pop\_back()** verwenden **stack**-Objekte die Methodennamen **push()** und **pop()**, die durch Inlinemethoden auf **push\_back()** bzw. **pop\_back()** abgebildet werden. Diese Verallgemeinerung ist bei **stack**-Objekten möglich, da die funktionale Definition des **stack**-Objektes eine Unterscheidung zwischen **push\_back()** / **push\_front()** bzw. **pop\_back()** / **pop\_front()** unnötig macht.

Die Deklaration eines **stack**-Objekttyps kann durch eine **typedef**-Anweisung weiter vereinfacht werden und hat folgenden grundsätzlichen Aufbau:

```
stack<Pushpopcontainerklasse<Datentyp> > Variablenname;
typedef stack<Pushpopcontainerklasse<Datentyp> > Datentypname;
```

**ACHTUNG:** Das Leerzeichen zwischen den beiden schließenden, spitzen Klammern „> >“ ist zwingend notwendig, da der Compiler zusammenhängende spitze Klammern „>>“ versucht als den entsprechenden Shift-Operator zu verwenden.

Soweit die Theorie. Sowohl der Microsoft-, als auch der Borland-Compiler sind nicht in der Lage die in der einschlägigen Literatur so angegebenen Beispiele zu übersetzen. Vielmehr verhält sich die **stack**-Klasse wie eine normale Containerklasse, die mit Objektklassen anstelle von Containerklassen arbeitet.



```
stack<Datentyp> Variablenname;
typedef stack<Datentyp> Datentypname;
```

### 10.1 Die stack Constructoren

Die folgende Tabelle faßt die Constructoren der Adapterklasse **stack** zusammen. In den nachstehenden Abschnitten werden die Constructoren einzeln behandelt und mit Beispielen erläutert.

<b>Constructoren der Adapterklasse STACK</b>	
<i>Constructor</i>	<i>Bedeutung</i>
stack<P> VarName;	Standard-Constructor, erzeugt ein neues, leeres <b>stack</b> -Objekt für die Containerklasse <b>P</b>

Tabelle 10-1– Constructoren der Adapterklasse stack

#### 10.1.1 Der stack Standard-Constructor

Der **stack** Standard-Constructor, erzeugt ein neues, leeres **stack**-Objekt

<sup>1</sup> FILO = First-In-Last-Out. D.h. die in einer solchen Datenstruktur befindlichen Datensätze können nur in der umgekehrten Reihenfolge ihrer Speicherung wieder abgerufen werden. LIFO = Last-In-First-Out. Identisch mit FILO.

vom Typ **P**, wobei es sich bei **P** um eine Containerklasse handeln muß, die einen Datenzugriff mit **push\_back()** / **pop\_back()** ermöglicht. In den verfügbaren STL-Implementationen sind dies die Containerklassen **vector**, **deque** und **list**.

```
Syntax (gemäß STL-Definition):
    stack<Container<T> > Variablenname;

Syntax (zumeist realisiert):
    stack<T> Variablenname;
```

Das folgende Beispielprogramm zeigt den Einsatz des Standard-Constructors innerhalb eines Programms:



```
//=====
// PROGRAMM: STACK_BSP_01
//=====

#include <stack>
#include <vector>
#include <list>
using namespace std;

void main (void)
{
    stack<int>  aStack1;
    stack< double> aStack2;
}
```

## 10.2 Die stack Methoden

Die folgende Tabelle faßt die Methoden der **stack** Adapterklasse zusammen. In den nachstehenden Abschnitten werden die Methoden einzeln behandelt und mit Beispielen erläutert.

<b>Methoden der Adapterklasse STACK</b>	
<i> Methode </i>	<i> Bedeutung </i>
empty	Gibt den Wert <b>true</b> zurück, wenn der <b>stack</b> keine Datenelemente enthält
pop	Entfernen des obersten Datenelementes welches über den <b>stack</b> verwaltet wird
push	Hinzufügen eines neuen Datenelementes ans oberster Position im <b>stack</b>
size	Gibt die Anzahl von Datenelementen zurück, die aktuell über den <b>stack</b> verwaltet werden
top	Gibt das „oberste“ Datenelement im <b>stack</b> zurück

Tabelle 10-1– Methoden der Adapterklasse stack

### 10.2.1 Die stack Methode empty

Die Methode **empty()** prüft, ob der **stack** Datenelemente enthält oder nicht. Sind Datenelemente enthalten (dies entspricht einem Rückgabewert von **size()** größer als Null), so wird der Wert **false** zurückgegeben, sind keine Datenelemente vorhanden ist der Rückgabewert **true**. Der Aufruf von **empty()** verändert den Inhalt des **stack**-Objektes nicht.

Syntax:

```
bool empty () const;
```

```
//=====
// PROGRAMM: STACK_BSP_02
//=====

#include <iomanip.h>
#include <iostream.h>
#include <stack>
using namespace std;

void main (void)
{
    stack<int> aStack;

    if (aStack.empty())
    {
        cout << "Stack ist leer" << endl;
    }
    else
    {
        cout << "Stack enthält Datenelemente" << endl;
    }

    aStack.push(3);
    aStack.push(6);
    aStack.push(7);

    if (aStack.empty())
    {
        cout << "Stack ist leer" << endl;
    }
    else
    {
        cout << "Stack enthält Datenelemente" << endl;
    }
}
```



### 10.2.2 Die stack Methode pop

Die Methode **pop** kapselt in einem **stack**-Objekt den Aufruf von **pop\_back()** der adaptierten Containerklasse. Der Befehl löscht das oberste im **stack**-Objekt gespeicherte Datenelement und entfernt es aus der Containerklasse, so daß das darunterliegende Datenelement zugänglich wird.

Syntax:

```
void pop ();
```

```
//=====
// PROGRAMM: STACK_BSP_03
//=====

#include <iomanip.h>
#include <iostream.h>
#include <stack>
using namespace std;
```



```

void main (void)
{
    stack<int> aStack;

    aStack.push(1);
    aStack.push(3);
    aStack.push(4);

    aStack.pop(); // Entfernt Datenelement „4“
    cout << "aktuelles Datenelement: " << aStack.top();
}

```

### 10.2.3 Die stack Methode push

Die Methode **push** kapselt in einem **stack**-Objekt den Aufruf von **push\_back()** der adaptierten Containerklasse. Der Befehl fügt ein neues Datenelement als oberstes Element im **stack**-Objekt ein. Dadurch wird das neue eingefügte Datenelement für die Methode **top()** sichtbar. Das bisher an dieser Stelle liegende Datenelement wird dadurch in der Sichtbarkeit verborgen und ist erst wieder sichtbar, wenn das gerade eingefügte Element über die Methode **pop()** entfernt wurde.

Syntax:

```
void push (const T& value);
```



```

//=====
// PROGRAMM: STACK_BSP_04
//=====

#include <iomanip.h>
#include <iostream.h>
#include <stack>
using namespace std;

void main (void)
{
    stack<int> aStack;

    aStack.push(1);
    aStack.push(4);

    cout << "aktuelles Datenelement: " << aStack.top();
}

```

### 10.2.4 Die stack Methode size

Gibt die Anzahl von Datenelementen zurück, die aktuell im **stack** enthalten sind. Da es sich bei **stack** um eine Adapterklasse handelt, die selbst gar keine Datenelemente verwaltet, wird über **size()** die entsprechende **size()**-Methode des enthaltenen Containerobjektes aufgerufen.

Syntax:

```
size_type size () const;
```

Der Aufruf von **size()** verändert den Inhalt des **stack** bzw. der adaptierten Containerklasse nicht.

```
//=====
// PROGRAMM: STACK_BSP_05
//=====

#include <iomanip.h>
#include <iostream.h>
#include <stack>
using namespace std;

void main (void)
{
    stack<int> aStack;

    aStack.push(1);
    aStack.push(3);
    aStack.push(4);

    cout << "Anzahl aktuell: " << aStack.size() << endl;
}

```



### 10.2.5 Die stack Methode top

Die Methode gibt das oberste Datenelement, welches im **stack** gespeichert ist. Dies ist die einzige Methode der Adapterklasse, um auf Dateninhalte zuzugreifen. Die Aufruf von **top()** entfernt das oberste Element nicht aus der adaptierten Containerklasse, dies muß mit Hilfe von **pop()** geschehen.

Syntax:

```
T& top ();
const T& top ();
```

### 10.3 Die stack Operatoren

Mit den folgenden Operatoren kann auf eine Adapterklasse vom Typ **stack** zugegriffen werden:

<b>Operatoren der Adapterklasse STACK</b>	
<i>Operator</i>	<i>Bedeutung</i>
=	Zuweisung zwischen zwei <b>stack</b> -Objekten gleichen Typs
==	Vergleich zwischen zwei <b>stack</b> -Objekten gleichen Typs
!=	Vergleich zwischen zwei <b>stack</b> -Objekten gleichen Typs
<	Lexikographischer Vergleich zwischen zwei <b>stack</b> -Objekten gleichen Typs
>	Lexikographischer Vergleich zwischen zwei <b>stack</b> -Objekten gleichen Typs
>=	Lexikographischer Vergleich zwischen zwei <b>stack</b> -Objekten gleichen Typs
<=	Lexikographischer Vergleich zwischen zwei <b>stack</b> -Objekten gleichen Typs

Tabelle 10-1– Operatoren der Adapterklasse stack

#### 10.3.1 Der stack Operator =

Der Operator ersetzt den Inhalt eines **stack**-Objektes durch den Inhalt des zugewiesenen **stack**-Objektes gleichen Typs.

Syntax (gemäß STL-Definition):

```
stack<Container<T> >& operator=
    (const stack<Container <T> >& s);
```

Syntax (zumeist realisiert):

```
stack<T>& operator= (const stack<T>& s);
```

*Durch die Zuweisung werden alle Verweise und Referenzen auf diesem Stack ungültig. Alle Verweise und Referenzen müssen neu ermittelt werden.*



```
//=====
// PROGRAMM: STACK_BSP_06
//=====

#include <iomanip.h>
#include <iostream.h>
#include <stack>
using namespace std;

void main (void)
{
    typedef stack<int> mystacktype;
    mystacktype aStack1;
    mystacktype aStack2;

    aStack1.push(1);
    aStack1.push(2);
    aStack1.push(3);

    aStack2 = aStack1;

    cout << "aStack1 Wert: " << aStack1.top() << endl;
    cout << "aStack2 Wert: " << aStack2.top() << endl;
}
```

### 10.3.2 Der stack Operator ==

Der Operator prüft, ob die Inhalte des linken und rechten Operanden (**stack**-Objekte gleichen Typs) übereinstimmen. Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn beide **stack**-Objekte die gleichen Datenelemente in der gleichen Reihenfolge enthalten.

Syntax (gemäß STL-Definition):

```
bool operator== (const stack<Container<T> >& s) const;
```

Syntax (zumeist realisiert):

```
bool operator== (const stack<T>& s) const;
```



```
//=====
// PROGRAMM: STACK_BSP_07
//=====

#include <iomanip.h>
#include <iostream.h>
```

```

#include <stack>
using namespace std;

void main (void)
{
    typedef stack<int> mystacktype;
    mystacktype aStack1;
    mystacktype aStack2;

    aStack1.push(1);

    if (aStack1 == aStack2)
    {
        cout << "Die Stack-Objekte sind gleich" << endl;
    }
    else
    {
        cout << "Die Stack-Objekte sind nicht gleich" << endl;
    }

    aStack2.push(1);

    if (aStack1 == aStack2)
    {
        cout << "Die Stack-Objekte sind gleich" << endl;
    }
    else
    {
        cout << "Die Stack-Objekte sind nicht gleich" << endl;
    }
}

```

### 10.3.3 Der stack Operator !=

Der Operator prüft, ob die Inhalte des linken und rechten Operanden (**stack**-Objekte gleichen Typs) verschieden sind. Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn beide **stack**-Objekte unterschiedliche Datenelemente oder gleiche Datenelemente in unterschiedlicher Reihenfolge enthalten.

Syntax (gemäß STL-Definition):

```
bool operator!= (const stack<Container<T> >& s) const;
```

Syntax (zumeist realisiert):

```
bool operator!= (const stack<T>& s) const;
```

```

//=====
// PROGRAMM: STACK_BSP_08
//=====

#include <iomanip.h>
#include <iostream.h>
#include <stack>
using namespace std;

void main (void)
{
    typedef stack<int> mystacktype;

```



```

mystacktype aStack1;
mystacktype aStack2;

aStack1.push(1);

if (aStack1 != aStack2)
{
    cout << "Die Stack-Objekte sind ungleich" << endl;
}
else
{
    cout << "Die Stack-Objekte sind nicht ungleich" << endl;
}

aStack2.push(1);

if (aStack1 != aStack2)
{
    cout << "Die Stack-Objekte sind ungleich" << endl;
}
else
{
    cout << "Die Stack-Objekte sind nicht ungleich" << endl;
}
}

```

### 10.3.4 Der stack Operator <

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch kleiner ist, als der des rechten Operanden (**stack**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „<“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch kleiner ist als der rechte Operand.

Syntax (gemäß STL-Definition):

```
bool operator< (const stack<Container<T> >& s) const;
```

Syntax (zumeist realisiert):

```
bool operator< (const stack<T>& s) const;
```



```

//=====
// PROGRAMM: STACK_BSP_09
//=====

#include <iomanip.h>
#include <iostream.h>
#include <stack>
using namespace std;

void main (void)
{
    stack<int> aStack1;
    stack<int> aStack2;

    aStack1.push(1);
    aStack2.push(2);

    if (aStack1 < aStack2)
    {

```

```

    cout << "aStack1 ist kleiner als aStack2" << endl;
}
else
{
    cout << " aStack1 ist nicht kleiner als aStack2" << endl;
}
}

```

### 10.3.5 Der stack Operator >

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch größer ist, als der des rechten Operanden (**stack**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „>“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch größer ist als der rechte Operand.

Syntax (gemäß STL-Definition):

```
bool operator> (const stack<Container<T> >& s) const;
```

Syntax (zumeist realisiert):

```
bool operator> (const stack<T>& s) const;
```

```

//=====
// PROGRAMM: STACK_BSP_10
//=====

#include <iomanip.h>
#include <iostream.h>
#include <stack>
using namespace std;

void main (void)
{
    stack<int> aStack1;
    stack<int> aStack2;

    aStack1.push(1);
    aStack2.push(2);

    if (aStack1 > aStack2)
    {
        cout << "aStack1 ist größer als aStack2" << endl;
    }
    else
    {
        cout << " aStack1 ist nicht größer als aStack2" << endl;
    }
}

```



### 10.3.6 Der stack Operator <=

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch kleiner oder gleich mit dem rechten Operanden ist (**stack**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „<=“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch kleiner oder gleich mit dem rechten Operanden ist.

Syntax (gemäß STL-Definition):

```
bool operator<= (const stack<Container<T> >& s) const;
```

Syntax (zumeist realisiert):

```
bool operator<= (const stack<T>& s) const;
```



```
//=====
// PROGRAMM: STACK_BSP_11
//=====

#include <iomanip.h>
#include <iostream.h>
#include <stack>
using namespace std;

void main (void)
{
    stack<int> aStack1;
    stack<int> aStack2;

    aStack1.push(1);
    aStack2.push(2);

    if (aStack1 <= aStack2)
    {
        cout << "aStack1 ist kleiner gleich aStack2" << endl;
    }
    else
    {
        cout << " aStack1 ist nicht kleiner gleich aStack2" << endl;
    }
}
```

### 10.3.7 Der stack Operator >=

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch größer oder gleich mit dem rechten Operanden ist (**stack**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „>=“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch größer oder gleich mit dem rechten Operanden ist.

Syntax (gemäß STL-Definition):

```
bool operator>= (const stack<Container<T> >& s) const;
```

Syntax (zumeist realisiert):

```
bool operator>= (const stack<T>& s) const;
```



```
//=====
// PROGRAMM: STACK_BSP_12
//=====

#include <iomanip.h>
#include <iostream.h>
#include <stack>
using namespace std;

void main (void)
{
```

```
stack<int> aStack1;
stack<int> aStack2;

aStack1.push(1);
aStack2.push(2);

if (aStack1 >= aStack2)
{
    cout << "aStack1 ist größer gleich aStack2" << endl;
}
else
{
    cout << " aStack1 ist nicht größer gleich aStack2" << endl;
}
}
```

## 11. Die Adapterklasse queue

Die Adapterklasse **queue** ermöglicht es, jede Containerklasse **P**, welche die Methoden **push\_back()** und **pop\_front()** implementiert in Form einer **FIFO**- bzw. **LIFO**-Datenstruktur<sup>1</sup> zu verwenden (Warteschlange).

Da **queue**-Objekte immer nur den Zugriff auf das „älteste“ Datenelement erlaubt, werden keinerlei Iteratoren unterstützt.

Anstelle von **push\_back()** und **pop\_front()** verwenden **queue**-Objekte die Methodennamen **push()** und **pop()**, die durch Inlinemethoden auf **push\_back()** bzw. **pop\_front()** abgebildet werden. Diese Verallgemeinerung ist bei **queue**-Objekten möglich, da die funktionale Definition des **queue**-Objektes eine Unterscheidung zwischen **push\_back()** / **push\_front()** bzw. **pop\_back()** / **pop\_front()** unnötig macht.

Die Deklaration eines **queue**-Objektyps kann durch eine **typedef**-Anweisung weiter vereinfacht werden und hat folgenden grundsätzlichen Aufbau:

```
queue<Pushpopcontainerklasse<Datentyp> > Variablenname;
typedef queue<Pushpopcontainerklasse<Datentyp> > Datentypname;
```

**ACHTUNG:** Das Leerzeichen zwischen den beiden schließenden, spitzen Klammern „> >“ ist zwingend notwendig, da der Compiler zusammenhängende spitze Klammern „>>“ versucht als den entsprechenden Shift-Operator zu verwenden.

Soweit die Theorie. Sowohl der Microsoft-, als auch der Borland-Compiler sind nicht in der Lage die in der einschlägigen Literatur so angegebenen Beispiele zu übersetzen. Vielmehr verhält sich die **queue**-Klasse wie eine normale Containerklasse, die mit Objektklassen anstelle von Containerklassen arbeitet.



```
queue<Datentyp> Variablenname;
typedef queue<Datentyp> Datentypname;
```

### 11.1 Die queue Constructoren

Die folgende Tabelle faßt die Constructoren der Adapterklasse **queue** zusammen. In den nachstehenden Abschnitten werden die Constructoren einzeln behandelt und mit Beispielen erläutert.

<b>Constructoren der Adapterklasse QUEUE</b>	
<i>Constructor</i>	<i>Bedeutung</i>
queue<P> VarName;	Standard-Constructor, erzeugt ein neues, leeres <b>queue</b> -Objekt für die Containerklasse <b>P</b>

Tabelle 11-1– Constructoren der Adapterklasse queue

<sup>1</sup> FIFO = First-In-First-Out. D.h. die in einer solchen Datenstruktur befindlichen Datensätze können nur in der Reihenfolge ihrer Speicherung wieder abgerufen werden. LIFO = Last-In-Last-Out. Identisch mit FIFO.

### 11.1.1 Der queue Standard-Constructor

Der **queue** Standard-Constructor, erzeugt ein neues, leeres **queue**-Objekt vom Typ **P**, wobei es sich bei **P** um eine Containerklasse handeln muß, die einen Datenzugriff mit **push\_back()** / **pop\_front()** ermöglicht. In den verfügbaren STL-Implementationen sind dies die Containerklassen **deque** und **list**.

```
Syntax (gemäß STL-Definition):
    queue<Container<T> > Variablenname;

Syntax (zumeist realisiert):
    queue <T> Variablenname;
```

Das folgende Beispielprogramm zeigt den Einsatz des Standard-Constructors innerhalb eines Programms:



```
//=====
// PROGRAMM: QUEUE_BSP_01
//=====

#include <queue>
using namespace std;

void main (void)
{
    queue<int>    aQueue1;
    queue<double> aQueue2;
}
```

### 11.2 Die queue Methoden

Die folgende Tabelle faßt die Methoden der **queue** Adapterklasse zusammen. In den nachstehenden Abschnitten werden die Methoden einzeln behandelt und mit Beispielen erläutert.

<b>Methoden der Adapterklasse QUEUE</b>	
<i> Methode </i>	<i> Bedeutung </i>
back	Gibt einen Zeiger auf das letzte (neueste) Datenelement in der <b>queue</b> zurück
empty	Gibt den Wert <b>true</b> zurück, wenn die <b>queue</b> keine Datenelemente enthält
front	Gibt einen Zeiger auf das erste (älteste) Datenelement in der <b>queue</b> zurück
pop	Entfernen des ersten (ältesten) Datenelementes welches über die <b>queue</b> verwaltet wird
push	Hinzufügen eines Datenelementes Ende der <b>queue</b> (Warteschlange)
size	Gibt die Anzahl von Datenelementen zurück, die aktuell über die <b>queue</b> verwaltet werden

Tabelle 11-1– Methoden der Adapterklasse queue

### 11.2.1 Die queue Methode back

Die Methode **back()** liefert einen Zeiger auf das „neueste“ Datenelement in der **queue**. Da die **queue** eine einfache Warteschlange realisiert, ist dies das Datenelement am „Ende“ der Warteschlange.

Syntax:

```
T& back ();
Const T& back () const;
```

```
//=====
// PROGRAMM: QUEUE_BSP_02
//=====

#include <iomanip.h>
#include <iostream.h>
#include <queue>
using namespace std;

void main (void)
{
    queue<int> aQueue;

    aQueue.push(1);
    aQueue.push(5);
    aQueue.push(11);

    cout << "letztes Datenelement: " << aQueue.back() << endl;
}
```



### 11.2.2 Die queue Methode empty

Die Methode **empty()** prüft, ob die **queue** Datenelemente enthält oder nicht. Sind Datenelemente enthalten (dies entspricht einem Rückgabewert von **size()** größer als Null), so wird der Wert **false** zurückgegeben, sind keine Datenelemente vorhanden ist der Rückgabewert **true**.

Der Aufruf von **empty()** verändert den Inhalt des **queue**-Objektes nicht.

Syntax:

```
bool empty () const;
```

```
//=====
// PROGRAMM: QUEUE_BSP_03
//=====

#include <iomanip.h>
#include <iostream.h>
#include <queue>
using namespace std;

void main (void)
{
    queue<int> aQueue;

    if (aQueue.empty())
    {
        cout << "Queue ist leer" << endl;
    }
    else
```



```

    {
        cout << "Queue enthält Datenelemente" << endl;
    }

    aQueue.push(3);
    aQueue.push(6);
    aQueue.push(7);

    if (aQueue.empty())
    {
        cout << "Queue ist leer" << endl;
    }
    else
    {
        cout << "Queue enthält Datenelemente" << endl;
    }
}

```

### 11.2.3 Die queue Methode front

Die Methode **front()** liefert einen Zeiger auf das „älteste“ Datenelement in der **queue**. Da die **queue** eine einfache Warteschlange realisiert, ist dies das Datenelement am „Anfang“ der Warteschlange.

Syntax:

```

T& front ();
Const T& front () const;

```



```

//=====
// PROGRAMM: QUEUE_BSP_04
//=====

#include <iomanip.h>
#include <iostream.h>
#include <queue>
using namespace std;

void main (void)
{
    queue<int> aQueue;

    aQueue.push(1);
    aQueue.push(5);
    aQueue.push(11);

    cout << "erstes Datenelement: " << aQueue.front() << endl;
}

```

### 11.2.4 Die queue Methode pop

Die Methode **pop** kapselt in einem **queue**-Objekt den Aufruf von **pop\_front()** der adaptierten Containerklasse. Der Befehl löscht das „älteste“ im **queue**-Objekt gespeicherte Datenelement und entfernt es aus der Containerklasse, so daß das nächste Datenelement zugänglich wird.

Syntax:

```

void pop ();

```

```
//=====
// PROGRAMM: QUEUE_BSP_05
//=====

#include <iomanip.h>
#include <iostream.h>
#include <queue>
using namespace std;

void main (void)
{
    queue<int> aQueue;

    aQueue.push(1);
    aQueue.push(3);
    aQueue.push(4);

    aQueue.pop(); // Entfernt Datenelement „4“
    cout << "aktuelles Datenelement: " << aQueue.front();
}

```



### 11.2.5 Die queue Methode push

Die Methode **push** kapselt in einem **queue**-Objekt den Aufruf von **push\_back()** der adaptierten Containerklasse. Der Befehl fügt ein neues Datenelement als „neuestes“ Element am Ende der Warteschlange ein. Dadurch wird das neue eingefügte Datenelement für die Methode **back()** sichtbar.

Syntax:

```
void push (const T& value);
```

```
//=====
// PROGRAMM: QUEUE_BSP_06
//=====

#include <iomanip.h>
#include <iostream.h>
#include <queue>
using namespace std;

void main (void)
{
    queue<int> aQueue;

    aQueue.push(1);
    aQueue.push(4);

    cout << "aktuelles Datenelement: " << aQueue.front();
}

```



### 11.2.6 Die queue Methode size

Gibt die Anzahl von Datenelementen zurück, die aktuell in der **queue** enthalten sind. Da es sich bei **queue** um eine Adapterklasse handelt, die selbst gar keine Datenelemente verwaltet, wird über **size()** die entsprechende **size()**-Methode des enthaltenen Containerobjektes aufgerufen.



```
queue <T>& operator= (const queue <T>& s);
```

*Durch die Zuweisung werden alle Verweise und Referenzen auf dieser Queue ungültig. Alle Verweise und Referenzen müssen neu ermittelt werden.*

```
//=====
// PROGRAMM: QUEUE_BSP_08
//=====

#include <iomanip.h>
#include <iostream.h>
#include <queue>
using namespace std;

void main (void)
{
    typedef queue<int> myqueuetype;
    myqueuetype aQueue1;
    myqueuetype aQueue2;

    aQueue1.push(1);
    aQueue1.push(2);
    aQueue1.push(3);

    aQueue2 = aQueue1;

    cout << "aQueue1 Wert: " << aQueue1.front() << endl;
    cout << "aQueue2 Wert: " << aQueue2.front() << endl;
}
```



### 11.3.2 Der queue Operator ==

Der Operator prüft, ob die Inhalte des linken und rechten Operanden (**queue**-Objekte gleichen Typs) übereinstimmen. Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn beide **queue**-Objekte die gleichen Datenelemente in der gleichen Reihenfolge enthalten.

Syntax (gemäß STL-Definition):

```
bool operator== (const queue<Container<T> >& s) const;
```

Syntax (zumeist realisiert):

```
bool operator== (const queue<T>& s) const;
```

```
//=====
// PROGRAMM: QUEUE_BSP_09
//=====

#include <iomanip.h>
#include <iostream.h>
#include <queue>
using namespace std;

void main (void)
{
```



```

typedef queue<int> myqueuetype;
myqueuetype aQueue1;
myqueuetype aQueue2;

aQueue1.push(1);

if (aQueue1 == aQueue2)
{
    cout << "Die Queue-Objekte sind gleich" << endl;
}
else
{
    cout << "Die Queue-Objekte sind nicht gleich" << endl;
}

aQueue2.push(1);

if (aQueue1 == aQueue2)
{
    cout << "Die Queue-Objekte sind gleich" << endl;
}
else
{
    cout << "Die Queue-Objekte sind nicht gleich" << endl;
}
}

```

### 11.3.3 Der queue Operator !=

Der Operator prüft, ob die Inhalte des linken und rechten Operanden (**queue**-Objekte gleichen Typs) verschieden sind. Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn beide **queue**-Objekte unterschiedliche Datenelemente oder gleiche Datenelemente in unterschiedlicher Reihenfolge enthalten.

Syntax (gemäß STL-Definition):

```
bool operator!= (const queue<Container<T> >& s) const;
```

Syntax (zumeist realisiert):

```
bool operator!= (const queue<T>& s) const;
```



```

//=====
// PROGRAMM: QUEUE_BSP_10
//=====

#include <iomanip.h>
#include <iostream.h>
#include <queue>
using namespace std;

void main (void)
{
    typedef queue<int> myqueuetype;
    myqueuetype aQueue1;
    myqueuetype aQueue2;

    aQueue1.push(1);

```

```

if (aQueue1 != aQueue2)
{
    cout << "Die Queue-Objekte sind ungleich" << endl;
}
else
{
    cout << "Die Queue-Objekte sind nicht ungleich" << endl;
}

aQueue2.push(1);

if (aQueue1 != aQueue2)
{
    cout << "Die Queue-Objekte sind ungleich" << endl;
}
else
{
    cout << "Die Queue-Objekte sind nicht ungleich" << endl;
}
}

```

### 11.3.4 Der queue Operator <

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch kleiner ist, als der des rechten Operanden (**queue**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „<“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch kleiner ist als der rechte Operand.

Syntax (gemäß STL-Definition):

```
bool operator< (const queue<Container<T> >& s) const;
```

Syntax (zumeist realisiert):

```
bool operator< (const queue<T>& s) const;
```

```

//=====
// PROGRAMM: QUEUE_BSP_11
//=====

#include <iomanip.h>
#include <iostream.h>
#include <queue>
using namespace std;

void main (void)
{
    queue<int> aQueue1;
    queue<int> aQueue2;

    aQueue1.push(1);
    aQueue2.push(2);

    if (aQueue1 < aQueue2)
    {
        cout << "aQueue1 ist kleiner als aQueue2" << endl;
    }
    else
    {
        cout << " aQueue1 ist nicht kleiner als aQueue2" << endl;
    }
}

```



```
}
}
```

### 11.3.5 Der queue Operator >

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch größer ist, als der des rechten Operanden (**queue**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „>“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch größer ist als der rechte Operand.

Syntax (gemäß STL-Definition):

```
bool operator> (const queue<Container<T> >& s) const;
```

Syntax (zumeist realisiert):

```
bool operator> (const queue<T>& s) const;
```



```
//=====
// PROGRAMM: QUEUE_BSP_12
//=====

#include <iomanip.h>
#include <iostream.h>
#include <queue>
using namespace std;

void main (void)
{
    queue<int> aQueue1;
    queue<int> aQueue2;

    aQueue1.push(1);
    aQueue2.push(2);

    if (aQueue1 > aQueue2)
    {
        cout << "aQueue1 ist größer als aQueue2" << endl;
    }
    else
    {
        cout << " aQueue1 ist nicht größer als aQueue2" << endl;
    }
}
```

### 11.3.6 Der queue Operator <=

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch kleiner oder gleich mit dem rechten Operanden ist (**queue**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „<=“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch kleiner oder gleich mit dem rechten Operanden ist.

Syntax (gemäß STL-Definition):

```
bool operator<= (const queue<Container<T> >& s) const;
```

Syntax (zumeist realisiert):

```
bool operator<= (const queue<T>& s) const;
```

```
//=====
// PROGRAMM: QUEUE_BSP_13
//=====

#include <iomanip.h>
#include <iostream.h>
#include <queue>
using namespace std;

void main (void)
{
    queue<int> aQueue1;
    queue<int> aQueue2;

    aQueue1.push(1);
    aQueue2.push(2);

    if (aQueue1 <= aQueue2)
    {
        cout << "aQueue1 ist kleiner gleich aQueue2" << endl;
    }
    else
    {
        cout << " aQueue1 ist nicht kleiner gleich aQueue2" << endl;
    }
}
```



### 11.3.7 Der queue Operator >=

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch größer oder gleich mit dem rechten Operanden ist (**queue**-Objekte gleichen Typs). Dazu werden die enthaltenen Datensätze Element für Element verglichen, indem deren Vergleichsoperator „>=“ aufgerufen wird.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch größer oder gleich mit dem rechten Operanden ist.

Syntax (gemäß STL-Definition):

```
bool operator>= (const queue<Container<T> >& s) const;
```

Syntax (zumeist realisiert):

```
bool operator>= (const queue<T>& s) const;
```

```
//=====
// PROGRAMM: QUEUE_BSP_14
//=====

#include <iomanip.h>
#include <iostream.h>
#include <queue>
using namespace std;

void main (void)
{
    queue<int> aQueue1;
    queue<int> aQueue2;

    aQueue1.push(1);
```



```
aQueue2.push(2);

if (aQueue1 >= aQueue2)
{
    cout << "aQueue1 ist größer gleich aQueue2" << endl;
}
else
{
    cout << " aQueue1 ist nicht größer gleich aQueue2" << endl;
}
}
```

## 12. Die Adapterklasse `priority_queue`

Die Adapterklasse **`priority_queue`** ermöglicht es, jede Containerklasse **P**, welche einen Iterator für wahlfreien Zugriff implementiert, eine sortierte Liste von Elementen verwalten zu lassen.

Obwohl der Name eigentlich anderes vermuten läßt, basieren die **`priority_queue`**-Objekte weniger auf dem Konzept der **`queue`** als auf dem Konzept des **`stack`**. Vom **`stack`** unterscheidet sich die **`priority_queue`** dadurch, daß die Reihenfolge der Elemente nicht allein durch ihre **`push`**-Reihenfolge bestimmt wird, sondern zudem eine Ordnungsklasse dafür sorgt, daß bestimmte Datenelemente (solche mit höherer Priorität) weiter „oben“ einsortiert werden.

Die Deklaration eines **`priority_queue`**-Objekttyps kann durch eine **`typedef`**-Anweisung weiter vereinfacht werden und hat folgenden grundsätzlichen Aufbau:

```
priority_queue<Containerklasse<Datentyp>, Ordnungsklasse>
    Variablenname;
typedef priority_queue<Containerklasse<Datentyp>, Ordnungsklasse>
    Datentypname;
```

Soweit die Theorie. Sowohl der Microsoft-, als auch der Borland-Compiler sind nicht in der Lage die in der einschlägigen Literatur so angegebenen Beispiele zu übersetzen. Vielmehr verhält sich die **`priority_queue`**-Klasse wie eine normale Containerklasse, die mit Objektklassen anstelle von Containerklassen arbeitet. Als Ordnungsklasse wird automatisch **`less<T>`** angenommen.



```
priority_queue<Datentyp> Variablenname;
typedef priority_queue<Datentyp> Datentypname;
```

### 12.1 Die `priority_queue` Constructoren

Die folgende Tabelle faßt die Constructoren der Adapterklasse **`priority_queue`** zusammen. In den nachstehenden Abschnitten werden die Constructoren einzeln behandelt und mit Beispielen erläutert.

<b>Constructoren der Adapterklasse <code>PRIORITY_QUEUE</code></b>	
<i>Constructor</i>	<i>Bedeutung</i>
<code>priority_queue&lt;P,C&gt;</code> <code>VarName;</code>	Standard-Constructor, erzeugt ein neues, leeres <b><code>priority_queue</code></b> -Objekt für die Containerklasse <b>P</b> mit der Ordnungsklasse <b>C</b>

Tabelle 12-1– Constructoren der Adapterklasse `priority_queue`

#### 12.1.1 Der `priority_queue` Standard-Constructor

Der **`priority_queue`** Standard-Constructor, erzeugt ein neues, leeres **`priority_queue`**-Objekt vom Typ **P**, wobei es sich bei **P** um eine Containerklasse handeln muß, die einen Iterator für wahlfreien Datenzugriff implementiert. In der STL sind dies die Containerklassen **`vector`** und **`deque`**.

Syntax (gemäß STL-Definition):

```
priority_queue<Container<T>, Ordnungsklasse C>
    Variablenname;
```

Syntax (zumeist realisiert):

```
priority_queue<T> Variablenname;
```

Das folgende Beispielprogramm zeigt den Einsatz des Standard-Constructors innerhalb eines Programms:



```
//=====
// PROGRAMM: PRIORITY_QUEUE_BSP_01
//=====

#include <queue>
using namespace std;

void main (void)
{
    priority_queue<int>    aQueue1;
    priority_queue<double> aQueue2;
}
```

## 12.2 Die priority\_queue Methoden

Die folgende Tabelle faßt die Methoden der **priority\_queue** Adapterklasse zusammen. In den nachstehenden Abschnitten werden die Methoden einzeln behandelt und mit Beispielen erläutert.

<b>Methoden der Adapterklasse PRIORITY_QUEUE</b>	
<i> Methode </i>	<i> Bedeutung </i>
empty	Gibt den Wert <b>true</b> zurück, wenn die <b>priority_queue</b> keine Datenelemente enthält
pop	Entfernen des obersten Datenelementes welches über die <b>priority_queue</b> verwaltet wird
push	Einfügen eines Datenelementes an der Stelle, die gemäß der Priorität des Datenelementes ermittelt wird
size	Gibt die Anzahl von Datenelementen zurück, die aktuell über die <b>priority_queue</b> verwaltet werden
top	Rückgabe des „obersten“ Datenelementes, d.h. jenes Datenelementes mit der höchsten Priorität.

Tabelle 12-1– Methoden der Adapterklasse priority\_queue

### 12.2.1 Die priority\_queue Methode empty

Die Methode **empty()** prüft, ob die **priority\_queue** Datenelemente enthält oder nicht. Sind Datenelemente enthalten (dies entspricht einem Rückgabewert von **size()** größer als Null), so wird der Wert **false** zurückgegeben, sind keine Datenelemente vorhanden ist der Rückgabewert **true**.

Der Aufruf von **empty()** verändert den Inhalt des **priority\_queue**-Objektes nicht.

Syntax:

```
bool empty () const;
```

```

//=====
// PROGRAMM: PRIORITY_QUEUE_BSP_02
//=====

#include <iomanip.h>
#include <iostream.h>
#include <queue>
using namespace std;

void main (void)
{
    priority_queue<int> aQueue;

    if (aQueue.empty())
    {
        cout << "Queue ist leer" << endl;
    }
    else
    {
        cout << "Queue enthält Datenelemente" << endl;
    }

    aQueue.push(3);
    aQueue.push(6);
    aQueue.push(7);

    if (aQueue.empty())
    {
        cout << "Queue ist leer" << endl;
    }
    else
    {
        cout << "Queue enthält Datenelemente" << endl;
    }
}

```



### 12.2.2 Die priority\_queue Methode pop

Die Methode löscht das „oberste“ im **priority\_queue**-Objekt gespeicherte Datenelement (das Datenelement mit der höchsten Priorität) und entfernt es aus der Containerklasse, so daß das nächste Datenelement zugänglich wird.

Syntax:

```
void pop ();
```

```

//=====
// PROGRAMM: PRIORITY_QUEUE_BSP_03
//=====

#include <iomanip.h>
#include <iostream.h>
#include <queue>
using namespace std;

void main (void)
{
    priority_queue<int> aQueue;

    aQueue.push(1);
    aQueue.push(3);
    aQueue.push(4);
}

```



```
aQueue.pop(); // Entfernt Datenelement „4“
cout << "aktuelles Datenelement: " << aQueue.top();
}
```

### 12.2.3 Die priority\_queue Methode push

Die Methode **push** fügt ein neues Datenelement in die **priority\_queue** ein. Wo das Element eingefügt wird, hängt dabei von der Priorität des Datenelementes ab, welche über die Ordnungsklasse bestimmt wird.

Syntax:

```
void push (const T& value);
```



```
//=====
// PROGRAMM: PRIORITY_QUEUE_BSP_04
//=====

#include <iomanip.h>
#include <iostream.h>
#include <queue>
using namespace std;

void main (void)
{
    priority_queue<int> aQueue;

    aQueue.push(1);
    aQueue.push(4);

    cout << "aktuelles Datenelement: " << aQueue.top();
}
```

### 12.2.4 Die priority\_queue Methode size

Gibt die Anzahl von Datenelementen zurück, die aktuell in der **priority\_queue** enthalten sind. Da es sich bei **priority\_queue** um eine Adapterklasse handelt, die selbst gar keine Datenelemente verwaltet, wird über **size()** die entsprechende **size()**-Methode des enthaltenen Containerobjektes aufgerufen.

Syntax:

```
size_type size () const;
```

Der Aufruf von **size()** verändert den Inhalt des **priority\_queue** bzw. der adaptierten Containerklasse nicht.



```
//=====
// PROGRAMM: PRIORITY_QUEUE_BSP_05
//=====

#include <iomanip.h>
#include <iostream.h>
#include <queue>
using namespace std;

void main (void)
{
    priority_queue<int> aQueue;
```

```

aQueue.push(1);
aQueue.push(3);
aQueue.push(4);

cout << "Anzahl aktuell: " << aQueue.size() << endl;
}

```

### 12.2.5 Die `priority_queue` Methode `top`

Die Methode gibt das oberste Datenelement der **priority\_queue** zurück, also das Datenelement mit der höchsten Priorität. Dies ist die einzige Methode der Adapterklasse, um auf Dateninhalte zuzugreifen. Die Aufruf von **top()** entfernt das oberste Element nicht aus der adaptierten Containerklasse, dies muß mit Hilfe von **pop()** geschehen.

Syntax:

```

T& top ();
const T& top ();

```

## 12.3 Die `priority_queue` Operatoren

Mit den folgenden Operatoren kann auf eine Adapterklasse vom Typ **priority\_queue** zugegriffen werden:

<b>Operatoren der Adapterklasse <code>PRIORITY_QUEUE</code></b>	
<i>Operator</i>	<i>Bedeutung</i>
=	Zuweisung zwischen zwei <b>priority_queue</b> -Objekten gleichen Typs

Tabelle 12-1– Operatoren der Adapterklasse `priority_queue`

### 12.3.1 Der `priority_queue` Operator =

Der Operator ersetzt den Inhalt eines **priority\_queue**-Objektes durch den Inhalt des zugewiesenen **priority\_queue**-Objektes gleichen Typs.

Syntax (gemäß STL-Definition):

```

priority_queue<Container<T> >& operator=
    (const priority_queue<Container <T> >& s);
    (const queue<Container <T> >& s);

```

Syntax (zumeist realisiert):

```

Priority_queue <T>& operator=
    (const priority_queue <T>& s);

```

*Durch die Zuweisung werden alle Iteratoren, Verweise und Referenzen auf dieser `Priority_queue` ungültig. Alle Iteratoren, Verweise und Referenzen müssen neu ermittelt werden.*

```

//=====
// PROGRAMM: PRIORITY_QUEUE_BSP_06
//=====

#include <iomanip.h>
#include <iostream.h>

```



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

void main (void)
{
    typedef priority_queue<int> myqueuetype;
    myqueuetype aQueue1;
    myqueuetype aQueue2;

    aQueue1.push(1);
    aQueue1.push(2);
    aQueue1.push(3);

    aQueue2 = aQueue1;

    cout << "aQueue1 Wert: " << aQueue1.top() << endl;
    cout << "aQueue2 Wert: " << aQueue2.top() << endl;
}
```

### 13. Die dynamische String-Klasse

Der **string** realisiert die in C/C++ fehlende, dynamische Zeichenkette.

Dynamisch bedeutet, daß die Größe eines **string**-Objekte sich zur Laufzeit des Programms ändern kann. Dies hat primär den Vorteil, daß man sich zum Zeitpunkt der Programmerstellung nur wenig Gedanken über die Größe der einzelnen Zeichenketten machen muß und auch Speicherüberläufe bei Stringoperationen wie der Konkatenation<sup>1</sup> gehören damit (weitgehend) der Vergangenheit an. Die dazu notwendige Speicherverwaltung wird automatisch vorgenommen und ist so effizient wie möglich realisiert. Trotz aller effizienten Programmierung ist aber zu bedenken, daß auch ein dynamischer String einen geschlossenen Speicherbereich benötigt, damit die darauf basierende Pointerarithmetik<sup>2</sup> funktioniert. Um eine Zeichenkette zu erweitern muß das eingebettete Character-Array daher relativ häufig im Speicher komplett umkopiert werden. Grundsätzlich gilt daher, daß Pointer, die von einem **string**-Objekt zurückgeliefert werden mit äußerster Vorsicht zu behandeln sind und möglichst nicht gepuffert werden sollten.

Da ein dynamischer **string** ohnehin starke Ähnlichkeiten mit einem **vector** aufweist, liegt eine Implementierung als dynamisches Array nahe. Es kommen lediglich einige spezielle Methoden zur Verbindung dynamischer **string**-Objekte mit den statischen Zeichenketten (realisiert über **char**-Arrays und **char**-Pointer) von C/C++ hinzu

Wie ein gewöhnliches **char**-Array ist **string** eine komplexe Datenstruktur, die mit Indexwerten angesprochen werden kann. Auch der STL-**string** beginnt, wie ein normales **char**-Array, bei der Indexzählung mit Null.

Da der STL-**string** den **operator[]** (Zugriff über Indexklammer) korrekt überlädt, kann auf die Inhalte des dynamischen Strings (fast) wie gewohnt zugegriffen werden.

Neue Zeichen können mit den entsprechenden Methoden an beliebiger Stelle eines Strings eingefügt werden. Allerdings ist zu beachten, das die Erweiterung eines **string**-Objektes nur am Ende mit hoher Effizienz erfolgt.

#### **ACHTUNG:**

*Sollen string-Objekte auf cout ausgegeben oder über cin eingelesen werden, so muß anstelle der üblicherweise eingebundenen Headerdateien:*

```
#include <iostream.h>
#include <iomanip.h>
```

*die für die STL erweiterte Headerdatei:*

```
#include <iostream>
#include <iomanip>
```

*(ohne „.h“) eingebunden werden.*



<sup>1</sup> Anhängen von Zeichenketten

<sup>2</sup> Die Pointerarithmetik ist nachzulesen im Kapitel über Arrays

Ein weiterer Vorteil, der aus der Verwendung der STL-Stringklasse erwächst, ist der Umstand, dass diese endlich eine Lösung für ein altes Problem liefert – das Einlesen beliebig langer Zeichenketten. Sowohl in C mit der Standard-IO<sup>3</sup>, wie auch unter C++ in der Stream-IO<sup>4</sup> besteht die Notwendigkeit, dass die maximale Länge einer einzulesenden Zeichenkette vor dem Aufruf der Lesefunktion festgelegt werden muss:



```
//=====
// PROGRAMM: STRING_BSP_LESEN_STANDARD_1
//=====

#include <stdio.h>

void main (void)
{
    char sTestStd[10];
    printf ("Bitte Text eingeben: ");
    fgets (sTestStd, 10, stdin);
    printf ("Eingegebener Text: %s\n", sTestStd);
}
```



```
//=====
// PROGRAMM: STRING_BSP_LESEN_STREAM_1
//=====

#include <iostream.h>

void main (void)
{
    char sTestStream[10];
    cout << "Bitte Text eingeben: ";
    cin.getline (sTestStream, 10);
    cout << "Eingegebener Text: " << sTestStream << endl;
}
```

Wie anhand der drei Beispielprogramme zu erkennen ist, kann lediglich bei der STL-Variante die Angabe einer Länge unterbleiben.

Zwar arbeiten auch die Standard-IO- und die Stream-IO-Variante inhaltlich korrekt, indem bei einer übergebenen Länge  $n$  nur  $n-1$  Zeichen eingelesen werden, kritisch wird es jedoch, wenn mehr als die im Beispiel vorgesehenen Zeichen eingegeben werden.

Diese zusätzlichen Zeichen bleiben bei der Standard-IO- und Stream-IO-Variante im Tastaturpuffer stehen und werden bei der nächsten Einlese-Operation herangezogen. Ist dies dann ein anderer Typ als Text wie z.B. ein Integerwert, so kann die Einlesefunktion die im Tastaturpuffer befindlichen Zeichen nicht verwerten und gibt – bei einzulesenden Zahlenwerten – ein undefiniertes Ergebnis (bestenfalls Null) zurück.

Die Standard-IO belässt die Zeichen weiterhin im Tastaturpuffer, die Stream-IO löscht danach immerhin den Tastaturpuffer.

<sup>3</sup> Als Standard-IO wird die Verwendung der Funktionen **getchar**, **putchar**, **scanf** und **printf** bezeichnet (sowie ihre Varianten und Spielarten).

<sup>4</sup> Als Stream-IO wird die Ein- und Ausgabe über Objekte bezeichnet, die sich, wie z.B. die Klassen **istream** und **ostream** (mit den zugehörigen Standard-Objekten **cin** und **cout**) von der Basisklasse **ios** ableiten.

Dieses Fehlerverhalten kann leicht geprüft werden, wenn bei den folgenden Beispielen mehr als die maximale Zahl der einlesbaren  $n-1$  Zeichen angegeben wird:

```
//=====
// PROGRAMM: STRING_BSP_LESEN_STANDARD_2
//=====

#include <stdio.h>

void main (void)
{
    int  nA;
    char sTestStd[10];

    printf ("Bitte Text eingeben: ");
    fgets  (sTestStd, 10, stdin);
    printf ("Eingegebener Text: %s\n", sTestStd);

    scanf ("%d", &nA);
    printf ("Eingegebene Zahl: %d\n", nA);

    fgets  (sTestStd, 10, stdin);
    printf ("Restlicher Text: %s\n", sTestStd);
}
```



```
//=====
// PROGRAMM: STRING_BSP_LESEN_STREAM_2
//=====

#include <iostream.h>

void main (void)
{
    int  nA;
    char sTestStream[10];

    cout << "Bitte Text eingeben: ";
    cin.getline (sTestStream, 10);
    cout << "Eingegebener Text: " << sTestStream << endl;

    cin >> nA;
    cout << "Eingegebene Zahl: " << nA << endl;

    cin.getline (sTestStream, 10);
    cout << " Restlicher Text: " << sTestStream << endl;
}
```



Da die STL-Stream-IO auf einem dynamischen String-Objekt arbeitet, entfallen diese Einschränkungen. Es wird stets die gesamte Eingabe in das Stringobjekt eingelesen. Ist diese zu lang für das nutzende Programm, so kann man die Zeichenkette mit den Methoden der **string**-Klasse verkürzen oder anderweitig verarbeiten. Folgefehler wie in den oben angeführten Beispielen entstehen nicht.

```
//=====
// PROGRAMM: STRING_BSP_LESEN_STL
//=====
```



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

using namespace std;

void main (void)
{
    string sTest;
    cout << "Bitte Text eingeben: ";
    cin >> sTest;
    cout << "Eingegebener Text: " << sTest <<endl;
}
```

### 13.1 Die string Constructoren

Die folgende Tabelle faßt die Constructoren der **string** Containerklasse zusammen. In den nachstehenden Abschnitten werden die Constructoren einzeln behandelt und mit Beispielen erläutert.

<b>Constructoren der Containerklasse <i>STRING</i></b>	
<i>Constructor</i>	<i>Bedeutung</i>
string VarName;	Standard-Constructor, erzeugt ein neues, leeres <b>string</b> -Objekt
string VarName (const char* str);	Dieser Constructor erzeugt einen neuen <b>string</b> und speichert eine Kopie der Zeichenkette <b>str</b>
string VarName (const string& str);	Dieser Constructor erzeugt einen neuen <b>string</b> und speichert eine Kopie der Zeichenkette <b>str</b>
string VarName (const char* str, size_t n);	Dieser Constructor erzeugt einen neuen <b>string</b> und speichert eine Teilkopie der Zeichenkette <b>str</b> , wobei die Teilzeichenkette die <b>n</b> Zeichen von <b>str</b> lang ist
string VarName (const string& str, size_t <sup>5</sup> pos, size_t n);	Dieser Constructor erzeugt einen neuen <b>string</b> und speichert eine Teilkopie der Zeichenkette <b>str</b> , wobei die Teilzeichenkette bei Position <b>pos</b> in <b>str</b> beginnt und maximal <b>n</b> Zeichen lang ist
string VarName (char c, size_t n);	Dieser Constructor erzeugt einen neuen <b>string</b> und wird mit <b>n</b> Kopien des Zeichens <b>c</b> initialisiert

Tabelle 13-1– Constructoren der Containerklasse *string*

#### 13.1.1 Der string Standard-Constructor

Der **string** Standard-Constructor, erzeugt ein neues, leeres **string**-Objekt.

Syntax:  
string Variablenname;

<sup>5</sup> Logischer Größentyp für eine Anzahl von Elementen vom Typ **T**

Das folgende Beispielprogramm zeigt den Einsatz des Standard-Constructors innerhalb eines Programms:

```
//=====
// PROGRAMM: STRING_BSP_01
//=====

#include <string>
using namespace std;

void main (void)
{
    string sMyString;
}
```



### 13.1.2 Der string mit Initialisierung über char-Pointer

Dieser Constructor erzeugt einen neuen **string**, der mit einer bereits bestehenden Zeichenkette oder einer Zeichenkettenkonstanten initialisiert wird. Die Initialisierung über einen **char**-Pointer dient auch der Anbindung von Programmteilen, die nicht mit STL-Funktionalitäten erstellt wurden. Die Standard-C Zeichenkette **str** wird vom **string**-Constructor nicht verändert.

Syntax:

```
string Variablenname (const char* str);
```

Das folgende Beispielprogramm zeigt den Einsatz des Constructors innerhalb eines Programms:

```
//=====
// PROGRAMM: STRING_BSP_02
//=====

#include <string>
using namespace std;

char sText [] = "Hallo Welt";

void main (void)
{
    string sMyString (sText);
}
```



Diese Form des Constructors ist besonders nützlich, wenn man bereits existierende Programmteile erweitern will, oder das Betriebssystem Texte von der Oberfläche in Form von **char**-Pointern zurückliefert.

### 13.1.3 Der string Constructor mit Initialisierung über Referenz

Dieser Constructor erzeugt einen neuen **string**, der mit einer bereits bestehenden Zeichenkette initialisiert wird. Die Initialisierung erfolgt über eine Referenz auf einen bereits bestehenden STL-String. Die Zeichenkette **str** wird vom **string**-Constructor nicht verändert.

Syntax:

```
string Variablenname (const string& str);
```

Das folgende Beispielprogramm zeigt den Einsatz des Constructors innerhalb eines Programms:



```
//=====
// PROGRAMM: STRING_BSP_03
//=====

#include <string>
using namespace std;

void main (void)
{
    string sEins ("Hallo Welt");
    string sZwei (sEins);
}
```

### 13.1.4 Der string Constructor mit Initialisierung über eine Teilzeichenkette (1)

Dieser Constructor erzeugt einen neuen **string**, der mit einer bereits bestehenden Zeichenkette initialisiert wird. Die Initialisierung erfolgt über einen Pointer auf einen bereits bestehende Standard-C Zeichenkette. Als Initialisierungsstring werden die ersten maximal **n** Zeichen der Zeichenkette **str** verwendet. Die Zeichenkette **str** wird vom **string**-Constructor nicht verändert.

Syntax:

```
string Variablenname (const char* str, size_t n);
```

Das folgende Beispielprogramm zeigt den Einsatz des Constructors innerhalb eines Programms:



```
//=====
// PROGRAMM: STRING_BSP_04
//=====

#include <string>
#include <iostream>
using namespace std;

void main (void)
{
    char sEins[] = "Charly Brown";
    string sZwei (sEins, 6);

    cout << sZwei << endl;
}
```

### 13.1.5 Der string Constructor mit Initialisierung über eine Teilzeichenkette (2)

Dieser Constructor erzeugt einen neuen **string**, der mit einer bereits bestehenden Zeichenkette initialisiert wird. Die Initialisierung erfolgt über eine Referenz auf einen bereits bestehenden STL-String. Als Initialisierungsstring wird eine Teilzeichenkette von maximal **n** Zeichen ab der Position **pos** in der Zeichenkette **str** verwendet. Die Zeichenkette **str** wird vom **string**-Constructor nicht verändert.

Syntax:

```
string Variablenname (const string& str, size_t pos,
                      size_t n);
```

Das folgende Beispielprogramm zeigt den Einsatz des Constructors innerhalb eines Programms:

```
//=====
// PROGRAMM: STRING_BSP_05
//=====

#include <string>
#include <iostream>
using namespace std;

void main (void)
{
    string sEins ("Peter Paul Rubens");
    string sZwei (sEins, 6, 4);

    cout << sZwei << endl;
}
```



### 13.1.6 Der string Constructor mit Initialisierung über Zeichenwiederholung

Dieser Constructor erzeugt einen neuen **string**, der mit einer Folge von **n**-Wiederholungen des Zeichens **c** initialisiert wird.

Syntax:

```
string Variablenname (size_t n, char c);
```

Das folgende Beispielprogramm zeigt den Einsatz des Constructors innerhalb eines Programms:

```
//=====
// PROGRAMM: STRING_BSP_06
//=====

#include <string>
#include <iostream>
using namespace std;

void main (void)
{
    string sLinie (80, '-');

    cout << sLinie << endl;
}
```



## 13.2 Die string Methoden

Die folgende Tabelle faßt die Methoden der **string** Containerklasse zusammen. In den nachstehenden Abschnitten werden die Methoden einzeln behandelt und mit Beispielen erläutert.

<b>Methoden der Containerklasse <i>STRING</i></b>	
<i>Methode</i>	<i>Bedeutung</i>
append	Anhängen von Zeichen an ein <b>string</b> -Objekt an
assign	Ersetzt den Inhalt des <b>string</b> -Objektes
c_str	Gibt einen <b>char</b> -Pointer auf den Inhalt des <b>string</b> -Objektes zurück
compare	Vergleicht <b>string</b> -Objekte
copy	Kopiert Zeichenfolgen des <b>string</b> -Objektes
data	Gibt einen nicht <b>NULL</b> -terminierten <b>char</b> -Pointer auf den Inhalt des <b>string</b> -Objektes zurück
empty	Gibt den Wert <b>true</b> zurück, wenn der <b>string</b> keine Zeichen enthält.
find	Suche innerhalb des <b>string</b> -Objektes
find_first_not_of	Suche innerhalb des <b>string</b> -Objektes
find_first_of	Suche innerhalb des <b>string</b> -Objektes
find_last_not_of	Suche innerhalb des <b>string</b> -Objektes
find_last_of	Suche innerhalb des <b>string</b> -Objektes
insert	Einfügen innerhalb des <b>string</b> -Objektes
length	Maximale Menge an Zeichen, die das <b>string</b> -Objekt enthalten kann
replace	Ersetzen von Zeichen oder Zeichenfolgen innerhalb des <b>string</b> -Objektes
reserve	Setzt die Anzahl der Zeichen fest, die ein <b>string</b> -Objekt vorab allokiert, um häufige Reallokationen zu vermeiden
resize	Kürzen oder Erweitern der Zeichenkette
rfind	Suche innerhalb des <b>string</b> -Objektes
size	Gibt die Anzahl von Zeichen zurück, die aktuell im <b>string</b> enthalten sind
substr	Rückgabe einer Teilzeichenkette

Tabelle 13-1– Methoden der Containerklasse *string*

### 13.2.1 Die *string* Methode **append**

Die Methode **append** dient dazu Zeichen an das Ende eines **string**-Objekt anzuhängen.

Syntax:

```
string& append (const char* str);
string& append (const char* str, size_t n);
string& append (size_t n, const char c);
string& append (const string& str, size_t pos=0,
               size_t n=-1);
```

Die erste Form der **append**-Methode erweitert das **string**-Objekt um den Inhalt der Standard-C Zeichenkette **str**.

Die zweite Form der **append**-Methode erweitert das **string**-Objekt um die ersten **n** Zeichen der Standard-C Zeichenkette **str**.

Die dritte Form der **append**-Methode erweitert das **string**-Objekt um **n** Wiederholungen des Zeichens **c**.

Die vierte Form der **append**-Methode erweitert das **string**-Objekt um den Inhalt des **string**-Objektes **str**. Dabei kann das gesamte **string**-Objekt angehängt werden (dann können die Parameter **n** und **pos** entfallen) oder ein Teilstring aus **str** angehängt werden.

Rückgabewert ist jeweils die Referenz auf das erweiterte **string**-Objekt, damit dieses in weitere, komplexere Ausdrücke eingesetzt werden kann.

Da die **append**-Methoden über die += Zuweisungsoperatoren überladen sind, werden sie nur selten direkt aufgerufen.

```
//=====
// PROGRAMM: STRING_BSP_07
//=====

#include <iostream>
#include <string>
using namespace std;

void main (void)
{
    string sTest ("noch nix");
    sTest.append (5, '_');
    sTest.append ("passiert");

    cout << sTest << endl;
}
```



### 13.2.2 Die string Methode assign

Die Methode **assign** weist dem **string**-Objekt einen neuen Inhalt zu.

Syntax:

```
string& assign (const char* str);
string& assign (const char* str, size_t n);
string& assign (size_t n, char c);
string& assign (const string& str, size_t pos,
               size_t n=-1);
```

Die erste Form der **assign**-Methode ersetzt den Inhalt des **string**-Objekts durch den Inhalt der Standard-C Zeichenkette **str**.

Die zweite Form der **assign**-Methode ersetzt den Inhalt des **string**-Objekts durch die ersten **n** Zeichen der Standard-C Zeichenkette **str**.

Die dritte Form der **assign**-Methode ersetzt den Inhalt des **string**-Objekts durch **n** Wiederholungen des Zeichens **c**.

Die vierte Form der **assign**-Methode ersetzt den Inhalt des **string**-Objekts durch den Inhalt des **string**-Objektes **str** ab der Position **pos**.

Rückgabewert ist jeweils die Referenz auf das erweiterte **string**-Objekt, damit dieses in weitere, komplexere Ausdrücke eingesetzt werden kann.

Da die **assign**-Methoden über die = Zuweisungsoperatoren überladen sind, werden sie nur selten direkt aufgerufen.



```
//=====
// PROGRAMM: STRING_BSP_08
//=====

#include <iomanip>
#include <iostream>
#include <string>
using namespace std;

void main (void)
{
    string sTest ("noch nix");
    cout << sTest << endl;

    sTest.assign (5, '_');
    cout << sTest << endl;

    sTest.assign ("passiert");
    cout << sTest << endl;
}
```

### 13.2.3 Die string Methode c\_str

Die Methode **c\_str()** gibt einen Standard-C kompatiblen **char**-Pointer auf die im **string**-Objekt beinhaltete Zeichenkette zurück. Dieser Zeiger kann in alle für Standard-C geschriebene Funktionen eingesetzt werden.

Syntax:

```
const char* c_str () const;
```

Damit die Daten im **string**-Objekt nicht über diesen Zeiger manipuliert werden können (und damit ggf. die Datenintegrität des Objektes zerstören), schließlich handelt es sich bei **string**-Objekten um dynamische Zeichenketten, über die außerhalb keine internen Zustände bekannt sind (wie z.B. die maximale Länge), ist nur ein **const char**-Pointer abrufbar.

Das letzte Zeichen in der durch **c\_str()** referenzierten Zeichenfolge ist das Zeichen **TRAITS::eos()**.



```
//=====
// PROGRAMM: STRING_BSP_09
//=====

#include <iomanip>
#include <iostream>
#include <string>
using namespace std;

void main (void)
{
    string sText ("Mein string");
    char  sT [200];

    strcpy (sT, sText.c_str());
   strupr (sT);
}
```

```

sText.assign (sT);
cout << sText << endl;
}

```

### 13.2.4 Die string Methode compare

Diese Methode vergleicht den Inhalt des **string**-Objektes mit einer anderen Zeichenkette.

Syntax:

```

int compare (const string& str, size_t pos=0,
             size_t n=-1) const;
int compare (char* str, size_t pos, size_t n) const;

```

Die erste Form der **compare**-Methode vergleicht den Inhalt des **string**-Objektes mit dem Inhalt des **string**-Objektes **str**. Der Vergleich mit einer Teilzeichenkette aus **str** ist möglich über Angabe von **pos** und **n**. Ist der Vergleich ganzer **string**-Objekte gewünscht, so können die Defaultparameter genutzt werden, die Angabe von **pos** und **n** entfällt dann.

Die zweite Form der **compare**-Methode vergleicht den Inhalt des **string**-Objektes mit dem Inhalt der Standard-C Zeichenfolge **str**. Der Vergleich mit einer Teilzeichenkette aus **str** ist möglich über Angabe von **pos** und **n**. Sollen ganze Zeichenketten verglichen werden, so ist für **pos** der Wert 0 und für **n** der Wert **strlen(str)** zu übergeben.

Der Rückgabewert der **compare**-Methode ist ein Integer-Wert, der in seiner Bedeutung dem Ergebnis der Standard-C Funktion **strcmp()** entspricht.

Da die **compare**-Methoden über die Vergleichsoperatoren überladen sind, werden sie nur selten direkt aufgerufen.

```

//=====
// PROGRAMM: STRING_BSP_10
//=====

#include <iomanip>
#include <iostream>
#include <string>
using namespace std;

void main (void)
{
    string sText1 ("Hallo Welt");
    string sText2 ("Hallo Wält");

    int nVergleich = sText1.compare (sText2);
    cout << nVergleich << endl;
}

```



### 13.2.5 Die string Methode copy

Die Methode **copy** kopiert eine Folge von **n** ab der Position **pos** in die angegebene Zeichenkette **str**, *ohne* das abschließende Nullbyte anzufügen.

Syntax:

```
size_type copy (char* str, size_t n, size_t pos=0)
               const;
```

Der Rückgabewert der **copy**-Methode ist die Anzahl der kopierten Zeichen.



```
//=====
// PROGRAMM: STRING_BSP_11
//=====

#include <iostream>
#include <string>
using namespace std;

void main (void)
{
    char pText1[80] = "*****";
    string aText2("zu kopierender Text");

    aText2.copy(pText1, 7, 4);

    cout << "[" << aText2 << "]" << endl;
    cout << "[" << pText1 << "]" << endl;
}
```

### 13.2.6 Die string Methode data

Die Methode **data()** gibt einen Standard-C kompatiblen **char**-Pointer auf die im **string**-Objekt beinhaltete Zeichenkette zurück. Im Gegensatz zur Methode **c\_str()** ist die referenzierte Zeichenkette jedoch nicht über ein Stringende-Zeichen terminiert.

Syntax:

```
const char* data () const;
```

Damit die Daten im **string**-Objekt nicht über diesen Zeiger manipuliert werden können (und damit ggf. die Datenintegrität des Objektes zerstören), schließlich handelt es sich bei **string**-Objekten um dynamische Zeichenketten, über die außerhalb keine internen Zustände bekannt sind (wie z.B. die maximale Länge), ist nur ein **const char**-Pointer abrufbar.

### 13.2.7 Die string Methode empty

Die Methode **empty()** prüft, ob der **string** Zeichen enthält oder nicht. Sind Zeichen enthalten, so wird der Wert **false** zurückgegeben, sind keine Zeichen vorhanden ist der Rückgabewert **true**.

Der Aufruf von **empty()** verändert den Inhalt des **string**-Objektes nicht.

Syntax:

```
bool empty () const;
```



```
//=====
// PROGRAMM: STRING_BSP_12
//=====

#include <iomanip>
```

```

#include <iostream>
#include <string>
using namespace std;

void main (void)
{
    string sText;

    if (sText.empty())
    {
        cout << "String ist leer" << endl;
    }
    else
    {
        cout << "String enthält Zeichen" << endl;
    }

    sText.assign ("Hallo Welt");

    if (sText.empty())
    {
        cout << "String ist leer" << endl;
    }
    else
    {
        cout << "String enthält Zeichens" << endl;
    }
}

```

### 13.2.8 Die string Methode find

Die Methode **find** ermöglicht es innerhalb einer Zeichenkette nach Teil-Zeichenketten zu suchen.

Syntax:

```

size_t find (const string& str, size_t pos) const;
size_t find (const char* str, size_t pos) const;
size_t find (const char* str, size_t pos, size_t n)
           const;

```

Die erste Form der **find**-Methode sucht im **string**-Objekt ab der Position **pos** nach dem Inhalt des Strings **str**.

Die zweite Form der **find**-Methode sucht im **string**-Objekt ab der Position **pos** nach dem Inhalt der Standard-C Zeichenfolge **str**.

Die dritte Form der **find**-Methode sucht im **string**-Objekt ab der Position **pos** nach den ersten **n** Zeichen der Standard-C Zeichenfolge **str**.

Der Rückgabewert der **find**-Methode ist ein Integer-Wert, welcher der Position des ersten Auftretens der gesuchten Zeichenkette im durchsuchten **string**-Objekt entspricht. Wird die gesuchte Zeichenfolge nicht gefunden, so wird als Ergebnis die Konstante **-1** zurückgegeben.

```

//=====
// PROGRAMM: STRING_BSP_13
//=====

```



```
#include <iomanip>
#include <iostream>
#include <string>
using namespace std;

void main (void)
{
    string sEins ("Charly Brown, Snoopy & Woodstock");
    string sZwei ("Snoopy");

    int nResult = sEins.find(sZwei, 0);
    cout << "Ergebnis: " << nResult << endl;

    nResult = sEins.find("Lucy", 0);
    cout << "Ergebnis: " << nResult << endl;
}
```

### 13.2.9 Die string Methode find\_first\_not\_of

Die Methode **find\_first\_not\_of** durchsucht das **string**-Objekt nach dem ersten Auftreten eines Zeichens, welches nicht in der übergebenen Zeichenmenge **str** vorkommt.

Syntax:

```
size_t find_first_not_of (const string& str,
                          size_t pos) const;
size_t find_first_not_of (const char& c,
                          size_t pos) const;
size_t find_first_not_of (const char* str,
                          size_t pos) const;
size_t find_first_not_of (const char* str,
                          size_t pos, size_t n) const;
```

Die erste Form der **find\_first\_not\_of**-Methode sucht im **string**-Objekt ab der Position **pos** nach einem Zeichen, welches nicht im Strings **str** enthalten ist.

Die zweite Form der **find\_first\_not\_of**-Methode sucht im **string**-Objekt ab der Position **pos** nach einem Zeichen, welches nicht dem Zeichen **c** entspricht.

Die dritte Form der **find\_first\_not\_of**-Methode sucht im **string**-Objekt ab der Position **pos** nach einem Zeichen, welches nicht in der Standard-C Zeichenfolge **str** enthalten ist.

Die vierte Form der **find\_first\_not\_of**-Methode sucht im **string**-Objekt ab der Position **pos** nach einem Zeichen, welches nicht in den ersten **n** Zeichen der Standard-C Zeichenfolge **str** enthalten ist.

Der Rückgabewert der **find\_first\_not\_of**-Methode ist ein Integer-Wert, welcher der Position des ersten Auftretens eines solchen Zeichens im durchsuchten **string**-Objekt entspricht. Wird kein Zeichen gefunden, so wird als Ergebnis die Konstante **-1** zurückgegeben.

```

//=====
// PROGRAMM: STRING_BSP_14
//=====

#include <iomanip>
#include <iostream>
#include <string>
using namespace std;

#define NPOS -1

void main (void)
{
    string sEins ("1110 0000 1100 1100 0101");
    int nPosition = sEins.find_first_not_of (" 01", 0);

    if (nPosition == NPOS)
    {
        cout << "Zeichenkette enthält Binärkode" << endl;
    }
    else
    {
        cout << "Zeichenkette enthält keinen Binärkode" << endl;
    }
}

```



### 13.2.10 Die string Methode find\_first\_of

Die Methode **find\_first\_of** durchsucht das **string**-Objekt nach dem ersten Auftreten eines Zeichens, welches in der übergebenen Zeichenmenge **str** vorkommt.

Syntax:

```

size_t find_first_of (const string& str,
                    size_t pos) const;
size_t find_first_of (const char& c,
                    size_t pos) const;
size_t find_first_of (const char* str,
                    size_t pos) const;
size_t find_first_of (const char* str,
                    size_t pos, size_t n) const;

```

Die erste Form der **find\_first\_of**-Methode sucht im **string**-Objekt ab der Position **pos** nach einem Zeichen, welches im Strings **str** enthalten ist.

Die zweite Form der **find\_first\_of**-Methode sucht im **string**-Objekt ab der Position **pos** nach einem Zeichen, welches dem Zeichen **c** entspricht.

Die dritte Form der **find\_first\_of**-Methode sucht im **string**-Objekt ab der Position **pos** nach einem Zeichen, welches in der Standard-C Zeichenfolge **str** enthalten ist.

Die vierte Form der **find\_first\_of**-Methode sucht im **string**-Objekt ab der Position **pos** nach einem Zeichen, welches in den ersten **n** Zeichen der Standard-C Zeichenfolge **str** enthalten ist.

Der Rückgabewert der **find\_first\_of**-Methode ist ein Integer-Wert, welcher der Position des ersten Auftretens eines solchen Zeichens im durchsuchten **string**-Objekt entspricht. Wird kein Zeichen gefunden, so wird als Ergebnis die Konstante **-1** zurückgegeben.



```
//=====
// PROGRAMM: STRING_BSP_15
//=====

#include <iomanip>
#include <iostream>
#include <string>
using namespace std;

void main (void)
{
    string sEins ("Donald Duck");
    int nPosition = sEins.find_first_of ("aeiouAEIOU", 0);

    if (nPosition == -1)
    {
        cout << "Zeichenkette enthält keine Vokale" << endl;
    }
    else
    {
        cout << "Zeichenkette enthält Vokale" << endl;
    }
}
```

### 13.2.11 Die string Methode find\_last\_not\_of

Die Methode **find\_last\_not\_of** durchsucht das **string**-Objekt nach dem letzten Auftreten eines Zeichens, welches nicht in der übergebenen Zeichenmenge **str** vorkommt.

Syntax:

```
size_t find_last_not_of (const string& str,
                        size_t pos) const;
size_t find_last_not_of (const char& c,
                        size_t pos) const;
size_t find_last_not_of (const char* str,
                        size_t pos) const;
size_t find_last_not_of (const char* str,
                        size_t pos, size_t n) const;
```

Die erste Form der **find\_last\_not\_of**-Methode sucht im **string**-Objekt ab der Position **pos** nach dem letzten Zeichen, welches nicht im Strings **str** enthalten ist.

Die zweite Form der **find\_last\_not\_of**-Methode sucht im **string**-Objekt ab der Position **pos** nach dem letzten Zeichen, welches nicht dem Zeichen **c** entspricht.

Die dritte Form der **find\_last\_not\_of**-Methode sucht im **string**-Objekt ab der Position **pos** nach dem letzten Zeichen, welches nicht in der Standard-C Zeichenfolge **str** enthalten ist.

Die vierte Form der **find\_last\_not\_of**-Methode sucht im **string**-Objekt ab der Position **pos** nach dem letzten Zeichen, welches nicht in den ersten **n** Zeichen der Standard-C Zeichenfolge **str** enthalten ist.

Der Rückgabewert der **find\_last\_not\_of**-Methode ist ein Integer-Wert, welcher der Position des letzten Auftretens eines solchen Zeichens im durchsuchten **string**-Objekt entspricht. Wird kein Zeichen gefunden, so wird als Ergebnis die Konstante **-1** zurückgegeben.

### 13.2.12 Die string Methode find\_last\_of

Die Methode **find\_last\_of** durchsucht das **string**-Objekt nach dem letzten Auftreten eines Zeichens, welches in der übergebenen Zeichenmenge **str** vorkommt.

Syntax:

```
size_t find_last_of (const string& str,
                    size_t pos) const;
size_t find_last_of (const char& c,
                    size_t pos) const;
size_t find_last_of (const char* str,
                    size_t pos) const;
size_t find_last_of (const char* str,
                    size_t pos, size_t n) const;
```

Die erste Form der **find\_last\_of**-Methode sucht im **string**-Objekt ab der Position **pos** nach dem letzten Zeichen, welches im Strings **str** enthalten ist.

Die zweite Form der **find\_last\_of**-Methode sucht im **string**-Objekt ab der Position **pos** nach dem letzten Zeichen, welches dem Zeichen **c** entspricht.

Die dritte Form der **find\_last\_of**-Methode sucht im **string**-Objekt ab der Position **pos** nach dem letzten Zeichen, welches in der Standard-C Zeichenfolge **str** enthalten ist.

Die vierte Form der **find\_last\_of**-Methode sucht im **string**-Objekt ab der Position **pos** nach dem letzten Zeichen, welches in den ersten **n** Zeichen der Standard-C Zeichenfolge **str** enthalten ist.

Der Rückgabewert der **find\_last\_of**-Methode ist ein Integer-Wert, welcher der Position des letzten Auftretens eines solchen Zeichens im durchsuchten **string**-Objekt entspricht. Wird kein Zeichen gefunden, so wird als Ergebnis die Konstante **-1** zurückgegeben.

### 13.2.13 Die string Methode insert

Die Methode **insert** fügt eine Zeichenfolge in das **string**-Objekt an der Stelle **pos** ein.

Syntax:

```
string& insert (size_t pos, string& str,
               size_t pos2=0, size_t n=-1);
```

```
string& insert (size_t pos, size_t n, char* str);
string& insert (size_t pos, char c, size_t n=1);
```

Die erste Form der **insert**-Methode fügt, ab der Position **pos**, den **string str** in das **string**-Objekt ein. Auf Wunsch kann auch eine Teilzeichenkette aus **str** eingefügt werden, deren Grenzen mit **pos2** (Anfang des Teilstrings) und **n** (Länge des Teilstrings) angegeben werden. Soll **str** komplett eingefügt werden, so können die Defaultparameter der Methode genutzt werden, so daß die Parameter **pos2** und **n** entfallen.

Die zweite Form der **insert**-Methode fügt, ab der Position **pos**, die ersten **n** Zeichen der Standard-C Zeichenkette **str** in das **string**-Objekt ein. Soll **str** komplett eingefügt werden, so kann für **n** das Funktionsergebnis von **strlen(str)** übergeben werden.

Die dritte Form der **insert**-Methode fügt, ab der Position **pos**, die **n** Wiederholungen des Zeichens **c** in das **string**-Objekt ein. Soll nur ein Zeichen eingefügt werden, so kann der Defaultparameter der Methode genutzt werden, so daß der Parameter **n** entfällt.

Rückgabewert ist jeweils die Referenz auf das erweiterte **string**-Objekt, damit dieses in weitere, komplexere Ausdrücke eingesetzt werden kann.



```
//=====
// PROGRAMM: STRING_BSP_16
//=====

#include <iomanip>
#include <iostream>
#include <string>
using namespace std;

void main (void)
{
    string sName1 ("Donald Duck");
    string sName2 ("Fauntleroy");

    sName1.insert(6, 1, ' ');
    sName1.insert(7, sName2);

    cout << "Name: " << sName1 << endl;
}
```

### 13.2.14 Die string Methode length

Gibt die maximale Anzahl an Datenelementen zurück, die der **string** insgesamt enthalten kann.

```
Syntax:
    size_t length () const;
```

**Achtung:** Obwohl der Methodenname es nahelegt, ist dies *nicht* die aktuelle Länge der im **string**-Objekt enthaltenen Zeichenkette. Zur Ermittlung der aktuellen Länge der Zeichenkette kann die Methode **size** verwendet werden.

```
//=====
// PROGRAMM: STRING_BSP_17
//=====

#include <iomanip>
#include <iostream>
#include <string>
using namespace std;

void main (void)
{
    string sText ("Hallo Welt");
    cout << "Länge: " << sText.length() << endl;
}

```



### 13.2.15 Die string Methode replace

Die Methode **replace** ersetzt einen Teil der im **string**-Objekt enthaltenen Zeichenkette durch eine andere übergebene Zeichenfolge.

Syntax:

```
string& replace (size_t pos, size_t n,
               const string& str,
               size_t pos2=0, size_t n2=-1);
string& replace (size_t pos, size_t n,
               const char* str, size_t n2);
string& replace (size_t pos, size_t n,
               const char c, size_t n2=1);
string& replace (size_t pos, size_t n,
               const char* str);

```

Die erste Form der **replace**-Methode ersetzt, ab der Position **pos**, die **n** Zeichen des **string**-Objektes durch die Zeichenkette **str** bzw. die Teilzeichenkette aus **str**, die durch die Defaultparameter **pos2** und **n2** definiert wird. Soll **str** komplett eingefügt werden, so können die Defaultparameter der Methode genutzt werden, so daß die Parameter **pos2** und **n2** entfallen.

Die zweite Form der **replace**-Methode ersetzt, ab der Position **pos**, die **n** Zeichen des **string**-Objektes durch die ersten **n2** Zeichen der Standard-C Zeichenkette **str**. Soll **str** komplett eingefügt werden, so kann für **n2** das Funktionsergebnis von **strlen(str)** übergeben werden.

Die dritte Form der **replace**-Methode ersetzt, ab der Position **pos**, die **n** Zeichen des **string**-Objektes durch **n2** Wiederholungen des Zeichens **c**. Soll das Zeichen **c** nur einmal eingefügt werden, so kann der Defaultparameter für **n2** genutzt werden und die Angabe von **n2** entfällt.

Die vierte Form der **replace**-Methode ersetzt, ab der Position **pos**, die **n** Zeichen des **string**-Objektes durch die Standard-C Zeichenkette **str**.

```
//=====
// PROGRAMM: STRING_BSP_18
//=====

```



```

#include <iomanip>
#include <iostream>
#include <string>
using namespace std;

void main (void)
{
    string sText ("Hallo Welt");
    cout << "Text: " << sText << endl;

    sText.replace(4, 3, "ERSETZUNG");
    cout << "Text: " << sText << endl;
}

```

Rückgabewert ist jeweils die Referenz auf das veränderte **string**-Objekt, damit dieses in weitere, komplexere Ausdrücke eingesetzt werden kann.

### 13.2.16 Die string Methode reserve

Die Methode **reserve** verwaltet den Pufferbereich des **string**-Objektes. Der **string**-Puffer bezeichnet die reservierte Größe für einen **string**, so daß nicht bei jeder Zeichenkettenmanipulation sofort eine Verschiebung und Reallokation der Speicherbereiche notwendig wird. Dies entspricht im wesentlichen dem unter C/C++ üblichen Vorgehen, eine Zeichenkette „großzügig“ zu dimensionieren. Erst wenn der **string** die über **reserve** definierte Puffergröße überschreitet wird eine Reallokation nötig.

Syntax:

```

size_t reserve () const;
void reserve (size_t anz);

```

Die erste Form der **resize**-Methode ermittelt die dem **string**-Objekt zugewiesene Puffergröße und gibt diese als Anzahl Zeichen zurück.

Die zweite Form der **resize**-Methode weist dem **string**-Objekt eine Puffergröße zu.

### 13.2.17 Die string Methode resize

Die **resize**-Methode verkürzt oder verlängert das **string**-Objekt. Wird eine neue Länge angegeben, die kürzer als die aktuelle Länge ist, so werden die überflüssigen Zeichen abgeschnitten. Wird eine Länge angegeben, die größer als die aktuelle Länge des **string**-Objektes ist, so wird mit Füllzeichen auf die gewünschte Länge erweitert.

Syntax:

```

void resize (size_t n, char c);
void resize (size_t n);

```

Die erste Form der **resize**-Methode kürzt die Zeichenkette auf die Länge **n** oder füllt gegebenenfalls mit Wiederholungen des Zeichens **c** auf die gewünschte Länge auf.

Die erste Form der **resize**-Methode kürzt die Zeichenkette auf die Länge **n** oder füllt gegebenenfalls mit Wiederholungen des Nullbyte-Zeichens (String-Ende-Symbol) auf die gewünschte Länge auf.

```
//=====
// PROGRAMM: STRING_BSP_19
//=====

#include <iomanip>
#include <iostream>
#include <string>
using namespace std;

void main (void)
{
    string sText ("Hallo Welt");
    cout << "Text: " << sText << endl;

    sText.resize(5);
    cout << "Text: " << sText << endl;

    sText.resize(10, '!');
    cout << "Text: " << sText << endl;
}
```



### 13.2.18 Die string Methode rfind

Die Methode **rfind** ermöglicht es innerhalb einer Zeichenkette nach Teil-Zeichenketten zu suchen. Im Gegensatz zur Methode **find** wird das **string**-Objekt aber von rechts nach links statt von links nach rechts durchsucht.

Syntax:

```
size_t rfind (const string& str, size_t pos=-1)
             const;
size_t rfind (const char* str, size_t pos=-1)
             const;
size_t rfind (const char* str, size_t pos, size_t n)
             const;
```

Die erste Form der **rfind**-Methode sucht im **string**-Objekt ab der Position **pos** nach dem Inhalt des Strings **str**.

Die zweite Form der **rfind**-Methode sucht im **string**-Objekt ab der Position **pos** nach dem Inhalt der Standard-C Zeichenfolge **str**.

Die dritte Form der **rfind**-Methode sucht im **string**-Objekt ab der Position **pos** nach den ersten **n** Zeichen der Standard-C Zeichenfolge **str**.

Der Rückgabewert der **rfind**-Methode ist ein Integer-Wert, welcher der Position des ersten Auftretens der gesuchten Zeichenkette im durchsuchten **string**-Objekt entspricht. Wird die gesuchte Zeichenfolge nicht gefunden, so wird als Ergebnis die Konstante **-1** zurückgegeben.

```
//=====
// PROGRAMM: STRING_BSP_20
//=====
```



```

#include <iomanip>
#include <iostream>
#include <string>
using namespace std;

void main (void)
{
    string sEins ("Charly Brown, Snoopy & Woodstock");
    string sZwei ("Snoopy");

    int nResult = sEins.rfind(sZwei);
    cout << "Ergebnis: " << nResult << endl;
}

```

### 13.2.19 Die string Methode size

Gibt die Anzahl von Zeichen zurück, die aktuell im **string** enthalten sind.

Syntax:

```
size_t size () const;
```

Der Aufruf von **size()** verändert den Inhalt des **string**-Objektes nicht.



```

//=====
// PROGRAMM: STRING_BSP_21
//=====

#include <iomanip>
#include <iostream>
#include <string>
using namespace std;

void main (void)
{
    string sText ("Hallo Welt");
    cout << "Anzahl aktuell: " << sText.size() << endl;
}

```

### 13.2.20 Die string Methode substr

Die Methode **substr** gibt eine Teilzeichenkette des **string**-Objektes zurück.

Syntax:

```
string& substr (size_t pos=0, size_t n=-1) const;
```

Die Teilzeichenkette besteht auf **n** Zeichen ab der Position **pos**. Wird für **n** die symbolische Konstante **-1** übergeben, so wird die restliche Zeichenkette bis zum String-Ende-Symbol zurückgegeben. Da **-1** als Defaultparameter für **n** definiert ist, kann dann die Angabe von **n** entfallen.

Für **pos** ist Null als Defaultparameter definiert, so daß der Aufruf von **substr** ohne Parameter eine Kopie des gesamten **string**-Objektes liefert.



```

//=====
// PROGRAMM: STRING_BSP_22
//=====

#include <iomanip>

```

```

#include <iostream>
#include <string>
using namespace std;

void main (void)
{
    string sText ("Hallo Welt");
    cout << "Teilstring: " << sText.substr(3,3) << endl;
    cout << "Reststring: " << sText.substr(3) << endl;
    cout << "String      : " << sText.substr() << endl;
}

```

### 13.3 Die string Operatoren

Mit den folgenden Operatoren kann auf eine Containerklasse vom Typ **string** zugegriffen werden:

<b>Operatoren der Containerklasse STRING</b>	
<i>Operato r</i>	<i>Bedeutung</i>
[ ]	Indezzugriff auf den Inhalt des <b>string</b>
=	Zuweisung zwischen zwei <b>string</b> -Objekten
+=	Konkatenation eines <b>string</b> -Objektes mit Zuweisung
==	Vergleich zwischen zwei <b>string</b> -Objekten
!=	Vergleich zwischen zwei <b>string</b> -Objekten
<	Lexikographischer Vergleich zwischen zwei <b>string</b> -Objekten
>	Lexikographischer Vergleich zwischen zwei <b>string</b> -Objekten
>=	Lexikographischer Vergleich zwischen zwei <b>string</b> -Objekten
<=	Lexikographischer Vergleich zwischen zwei <b>string</b> -Objekten
+=	Konkatenation zweier <b>string</b> -Objekte

Tabelle 13-1– Operatoren der Containerklasse string

#### 13.3.1 Der string Operator [ ]

Mittels der eckigen Klammern kann direkt auf ein im **string** gespeichertes Zeichen zugegriffen werden.

Syntax:

```

char& operator[] (size_t pos);
char operator[] (size_t pos) const;

```

Wird ein ungültiger Index angegeben, so wird eine **out\_of\_range** Exception ausgelöst:

```

//=====
// PROGRAMM: STRING_BSP_23
//=====

#include <iomanip>
#include <iostream>
#include <string>
using namespace std;

```



```

void main (void)
{
    string sText ("Bugs Bunny");

    try
    {
        cout << "Zeichen: " << sText[1] << endl;
        sText[1] = 'a';
        sText[6] = 'a';
        cout << "Text: " << sText << endl;
        cout << "Zeichen: " << sText[50] << endl; // Exception
    }
    catch (out_of_range& aError)
    {
        cout << "Zugriff auf illegales Element" << endl;
    }
}

```

### 13.3.2 Der string Operator =

Der Operator verwendet die **assign**-Methode um dem **string** einen neuen Inhalt zuzuweisen.

Syntax:

```

string& operator= (const string& str);
string& operator= (const char* str);
string& operator= (char c);

```

Durch Operator-Overloading können als Quelle unterschiedliche Datentypen verwendet werden. Eine Zuweisung auf ein **string**-Objekt kann somit sowohl über ein anderes **string**-Objekt (erste Form), eine Standard-C Zeichenkette (**char**-Pointer, zweite Form) oder ein einzelnes Zeichen (dritte Form) erfolgen.

Rückgabewert ist jeweils die Referenz auf das **string**-Objekt, damit dieses in weitere, komplexere Ausdrücke eingesetzt werden kann.



```

//=====
// PROGRAMM: STRING_BSP_24
//=====

#include <iomanip>
#include <iostream>
#include <string>
using namespace std;

void main (void)
{
    string sText1;
    string sText2 ("Daniel Düsentrieb");

    sText1 = sText2;
    cout << sText1 << endl;

    sText1 = "Donald Duck";
    cout << sText1 << endl;

    sText1 = '$';
}

```

```
    cout << sText1 << endl;
}
```

### 13.3.3 Der string Operator +=

Der Operator verwendet die **append**-Methode um an den **string** eine zweite Zeichenfolge anzuhängen.

Syntax:

```
string& operator+= (const string& str);
string& operator+= (const char* str);
string& operator+= (char c);
```

Durch Operator-Overloading können als Quelle unterschiedliche Datentypen verwendet werden. Eine Erweiterung (Konkatenation) des **string**-Objekts kann somit sowohl über ein anderes **string**-Objekt (erste Form), eine Standard-C Zeichenkette (**char**-Pointer, zweite Form) oder ein einzelnes Zeichen (dritte Form) erfolgen.

Rückgabewert ist jeweils die Referenz auf das **string**-Objekt, damit dieses in weitere, komplexere Ausdrücke eingesetzt werden kann.

```
//=====
// PROGRAMM: STRING_BSP_25
//=====

#include <iomanip>
#include <iostream>
#include <string>
using namespace std;

void main (void)
{
    string sText1 ("Daniel");
    string sText2 ("Düsentrieb");

    sText1 += ' ';
    cout << sText1 << endl;

    sText1 += sText2;
    cout << sText1 << endl;

    sText1 += " (Erfinder)";
    cout << sText1 << endl;
}
```



### 13.3.4 Der string Operator ==

Der Operator prüft, ob die Inhalte des linken und rechten **string**-Objekts übereinstimmen. Dazu werden die Inhalte Zeichen für Zeichen verglichen, indem die ANSI-Werte der Zeichen voneinander subtrahiert werden.

Das Ergebnis der Operation ist **true** wenn beide **strings** den gleichen Inhalt haben.

Syntax:

```
bool operator== (const string& str) const;
bool operator== (const char* str) const;
```

Der Rückgabewert ist ein **bool** anstelle des bei **compare** üblichen **int**. Dies erleichtert zwar die Abfrage­logik, führt aber dazu, daß ggf. zwei Vergleichsoperationen notwendig sind, um zwischen zwei **string**-Objekten eine eindeutige, lexikographische Reihenfolge zu ermitteln.



```
//=====
// PROGRAMM: STRING_BSP_26
//=====

#include <iomanip>
#include <iostream>
#include <string>
using namespace std;

void main (void)
{
    string sText1 ("Hallo");
    string sText2 ("Hello");

    if (sText1 == sText2)
    {
        cout << "Die Strings sind gleich" << endl;
    }
    else
    {
        cout << "Die Strings sind nicht gleich" << endl;
    }

    sText1 = sText2;
    if (sText1 == sText2)
    {
        cout << "Die Strings sind gleich" << endl;
    }
    else
    {
        cout << "Die Strings sind nicht gleich" << endl;
    }
}
}
```

### 13.3.5 Der string Operator !=

Der Operator prüft, ob die Inhalte des linken und rechten **string**-Objekts voneinander abweichen. Dazu werden die Inhalte Zeichen für Zeichen verglichen, indem die ANSI-Werte der Zeichen voneinander subtrahiert werden.

Das Ergebnis der Operation ist **true** wenn beide **strings** den verschiedene Inhalte haben.

Syntax:

```
bool operator!= (const string& str) const;
bool operator!= (const char* str) const;
```

Der Rückgabewert ist ein **bool** anstelle des bei **compare** üblichen **int**. Dies erleichtert zwar die Abfrage­logik, führt aber dazu, daß ggf. zwei Vergleichsoperationen notwendig sind, um zwischen zwei **string**-Objekten eine eindeutige, lexikographische Reihenfolge zu ermitteln.

```

//=====
// PROGRAMM: STRING_BSP_27
//=====

#include <iomanip>
#include <iostream>
#include <string>
using namespace std;

void main (void)
{
    string sText1 ("Hallo");
    string sText2 ("Hello");

    if (sText1 != sText2)
    {
        cout << "Die Strings sind ungleich" << endl;
    }
    else
    {
        cout << "Die Strings sind gleich" << endl;
    }

    sText1 = sText2;
    if (sText1 != sText2)
    {
        cout << "Die Strings sind ungleich" << endl;
    }
    else
    {
        cout << "Die Strings sind gleich" << endl;
    }
}

```



### 13.3.6 Der string Operator <

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch kleiner ist, als der des rechten Operanden. Dazu werden die Inhalte Zeichen für Zeichen verglichen, indem die ANSI-Werte der Zeichen voneinander subtrahiert werden.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch kleiner ist als der rechte Operand.

Syntax:

```

bool operator< (const string& str) const;
bool operator< (const char* str) const;

```

Der Rückgabewert ist ein **bool** anstelle des bei **compare** üblichen **int**. Dies erleichtert zwar die Abfragelogik, führt aber dazu, daß ggf. zwei Vergleichsoperationen notwendig sind, um zwischen zwei **string**-Objekten eine eindeutige, lexikographische Reihenfolge zu ermitteln.

```

//=====
// PROGRAMM: STRING_BSP_28
//=====

#include <iomanip>
#include <iostream>
#include <string>

```



```
using namespace std;

void main (void)
{
    string sText1 ("Hallo");
    string sText2 ("Hello");

    if (sText1 < sText2)
    {
        cout << "Text1 ist kleiner als Text2" << endl;
    }
    else
    {
        cout << "Text1 ist nicht kleiner als Text2" << endl;
    }
}
```

### 13.3.7 Der string Operator >

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch größer ist, als der des rechten Operanden. Dazu werden die Inhalte Zeichen für Zeichen verglichen, indem die ANSI-Werte der Zeichen voneinander subtrahiert werden.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch größer ist als der rechte Operand.

Syntax:

```
bool operator> (const string& str) const;
bool operator> (const char* str) const;
```

Der Rückgabewert ist ein **bool** anstelle des bei **compare** üblichen **int**. Dies erleichtert zwar die Abfrage-logik, führt aber dazu, daß ggf. zwei Vergleichsoperationen notwendig sind, um zwischen zwei **string**-Objekten eine eindeutige, lexikographische Reihenfolge zu ermitteln.



```
//=====
// PROGRAMM: STRING_BSP_29
//=====

#include <iomanip>
#include <iostream>
#include <string>
using namespace std;

void main (void)
{
    string sText1 ("Hallo");
    string sText2 ("Hello");

    if (sText1 > sText2)
    {
        cout << "Text1 ist größer als Text2" << endl;
    }
    else
    {
        cout << "Text1 ist nicht größer als Text2" << endl;
    }
}
```

### 13.3.8 Der string Operator <=

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch kleiner oder gleich dem rechten Operanden ist. Dazu werden die Inhalte Zeichen für Zeichen verglichen, indem die ANSI-Werte der Zeichen voneinander subtrahiert werden.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch kleiner oder gleich dem rechten Operanden ist.

Syntax:

```
bool operator<= (const string& str) const;
bool operator<= (const char* str) const;
```

Der Rückgabewert ist ein **bool** anstelle des bei **compare** üblichen **int**. Dies erleichtert zwar die Abfragelogik, führt aber dazu, daß ggf. zwei Vergleichsoperationen notwendig sind, um zwischen zwei **string**-Objekten eine eindeutige, lexikographische Reihenfolge zu ermitteln.

```
//=====
// PROGRAMM: STRING_BSP_30
//=====

#include <iomanip>
#include <iostream>
#include <string>
using namespace std;

void main (void)
{
    string sText1 ("Hallo");
    string sText2 ("Hello");

    if (sText1 <= sText2)
    {
        cout << "Text1 ist kleiner gleich als Text2" << endl;
    }
    else
    {
        cout << "Text1 ist größer als Text2" << endl;
    }
}
```



### 13.3.9 Der string Operator >=

Der Operator prüft, ob der Inhalt des linken Operanden lexikographisch größer oder gleich dem rechten Operanden ist. Dazu werden die Inhalte Zeichen für Zeichen verglichen, indem die ANSI-Werte der Zeichen voneinander subtrahiert werden.

Das Ergebnis der Operation ist **true** wenn der linke Operand lexikographisch größer oder gleich dem rechten Operanden ist.

Syntax:

```
bool operator>= (const string& str) const;
bool operator>= (const char* str) const;
```

Der Rückgabewert ist ein **bool** anstelle des bei **compare** üblichen **int**. Dies erleichtert zwar die Abfragelogik, führt aber dazu, daß ggf. zwei

Vergleichsoperationen notwendig sind, um zwischen zwei **string**-Objekten eine eindeutige, lexikographische Reihenfolge zu ermitteln.



```
//=====
// PROGRAMM: STRING_BSP_31
//=====

#include <iomanip>
#include <iostream>
#include <string>
using namespace std;

void main (void)
{
    string sText1 ("Hallo");
    string sText2 ("Hello");

    if (sText1 >= sText2)
    {
        cout << "Text1 ist größer gleich als Text2" << endl;
    }
    else
    {
        cout << "Text1 ist kleiner als Text2" << endl;
    }
}
```

### 13.3.10 Der string Operator +

Der Operator + erzeugt ein temporäres **string**-Objekt, welches die Konkatenation des linken und rechten Operanden darstellt. Dieses temporäre Objekt kann dann wiederum in einen komplexeren Ausdruck eingesetzt oder an ein anderes **string**-Objekt zugewiesen werden.

Syntax:

```
string& operator+ (const string& s1, const string& s2);
string& operator+ (const char* s1, const string& s2);
string& operator+ (const string& s1, const char* s2);
string& operator+ (const char c, const string& s2);
string& operator+ (const string& s1, const char c);
```

Als Ausgangs-Zeichenfolgen können entweder ein **string**-Objekte oder Standard-C Zeichenfolgen dienen.



```
//=====
// PROGRAMM: STRING_BSP_32
//=====

#include <iomanip>
#include <iostream>
#include <string>
using namespace std;

void main (void)
{
    string sText1;
    string sText2 ("Donald");

    sText1 = sText2 + ' ' + "Duck";
}
```

```
} cout << sText1 << endl;
```

## 14. Algorithmen im Überblick

In diesem Abschnitt wird die Verwendung jedes Algorithmus durch ein kurzes Beispiel erklärt. Einige der Algorithmen werden auch in den Beispielprogrammen verwendet, die in den Abschnitten zu den verschiedenen Container-Klassen zu finden sind.

Um die generischen Algorithmen zu nutzen muss die Headerdatei **algorithm** in die Programme mit eingebunden werden (bei einigen Compilern und STL-Implementationen kann es notwendig sein, die Include-Anweisungen auf **algorithm**, **algorithm.h** oder **algorithm.h** zu ändern). Die Funktionen **accumulate**, **inner\_product**, **partial\_sum**, und **adjacent\_difference** sind hingegen in der Header-Datei **numeric** definiert, so dass diese ggf. zusätzlich einzubinden ist.

<b>Anwendungsalgorithmen (applying algorithms)</b>	
<i>Name</i>	<i>Bedeutung</i>
for_each	Wendet eine Funktion auf jedes Element in einer Folge an

*Tabelle 14-1– Anwendungsalgorithmen*

<b>Aufteilungsalgorithmen (partitioning algorithms)</b>	
<i>Name</i>	<i>Bedeutung</i>
nth_element	Teilt die Elemente einer Folge am $n$ -ten Element
partition	Teilt die Elemente einer Folge in zwei Gruppen auf
stable_partition	Teilt die Elemente einer Folge unter Beibehaltung der ursprünglichen Ordnung auf

*Tabelle 14-2– Aufteilungsalgorithmen*

<b>Begrenzungsalgorithmen (bounding algorithms)</b>	
<i>Name</i>	<i>Bedeutung</i>
equal_range	Suchen eines Bereiches
lower_bound	Suchen der unteren Bereichsbegrenzung
upper_bound	Suchen der oberen Bereichsbegrenzung

*Tabelle 14-3– Begrenzungsalgorithmen*

<b>Berechnungsalgorithmen (math algorithms)</b>	
<i>Name</i>	<i>Bedeutung</i>
accumulate	Zählt eine Folge zusammen
adjacent_difference	erzeugt eine Folge benachbarter Differenzen
inner_product	Inneres Produkt zweier paralleler Folgen
partial_sum	Füllt eine Folge mit einer Prüfsummen

*Tabelle 14-4– Berechnungsalgorithmen*

<b>Einsortierungsalgorithmen (merging algorithms)</b>	
<i>Name</i>	<i>Bedeutung</i>
inplace_merge	führt zwei sortierte Folgen zu einer Folge zusammen (ohne Hilfsspeicher)
merge	führt zwei sortierte Folgen zu einer Folge zusammen

*Tabelle 14-5– Einsortierungsalgorithmen*

<b>Ersetzungsalgorithmen (replacing algorithms)</b>	
<i>Name</i>	<i>Bedeutung</i>
replace	Ersetzt bestimmte Elemente durch einen neuen

<b>Ersetzungsalgorithmen (replacing algorithms)</b>	
<i>Name</i>	<i>Bedeutung</i>
	Wert
replace_copy	Kopiert eine Folge und ersetzt dabei bestimmte Werte durch einen neuen Wert
replace_copy_if	Kopiert eine Folge und ersetzt dabei die Elemente durch einen neuen Wert, die einer Bedingung genügen
replace_if	Ersetzt die Elemente durch einen neuen Wert, die einer Bedingung genügen

*Tabelle 14-6– Ersetzungsalgorithmen*

<b>Filteralgorithmen (filtering algorithms)</b>	
<i>Name</i>	<i>Bedeutung</i>
unique	Entfernt alle doppelten Element, außer dem jeweils ersten Element
unique_copy	Kopiert eine Folge und entfernt dabei alle doppelten Element, außer dem jeweils ersten Element

*Tabelle 14-7– Filteralgorithmen*

<b>Füllalgorithmen (filling algorithms)</b>	
<i>Name</i>	<i>Bedeutung</i>
fill	Füllt eine Folge mit einem Initialwert
fill_n	Füllt n Positionen mit einem Initialwert

*Tabelle 14-8– Füllalgorithmen*

<b>Generierungsalgorithmen (generating algorithms)</b>	
<i>Name</i>	<i>Bedeutung</i>
generate	Initialisiert eine Folge durch einen Generator
generate_n	Initialisiert n Positionen durch einen Generator

*Tabelle 14-9– Generierungsalgorithmen*

<b>Heapalgorithmen (heap algorithms)</b>	
<i>Name</i>	<i>Bedeutung</i>
make_heap	Wandelt eine Folge in einen Heap um
pop_heap	Entfernt das oberste Element eines Heap
push_heap	Legt ein Element auf dem Heap ab
sort_heap	Sortiert einen Heap

*Tabelle 14-10– Heapalgorithmen*

<b>Kopieralgorithmen (copying algorithms)</b>	
<i>Name</i>	<i>Bedeutung</i>
copy	Kopiert eine Folge in eine andere Folge
copy_backward	Kopiert eine Folge in eine andere Folge

*Tabelle 14-11– Kopieralgorithmen*

<b>Löschalgorithmen (removing algorithms)</b>	
<i>Name</i>	<i>Bedeutung</i>
remove	Entfernt alle mit einem Wert übereinstimmenden Elemente
remove_copy	Kopiert eine Folge und entfernt dabei Elemente

<b>Löschalgorithmen (removing algorithms)</b>	
<i>Name</i>	<i>Bedeutung</i>
remove_copy_if	Kopiert eine Folge und entfernt dabei Elemente, die einer Bedingung genügen
remove_if	Entfernt alle Elemente, die einer Bedingung genügen

Tabelle 14-12– Löschalgorithmen

<b>Mengenalgorithmen (set algorithms)</b>	
<i>Name</i>	<i>Bedeutung</i>
includes	Prüft, ob alle Elemente einer Folge in einer anderen Folge enthalten sind
set_difference	Erzeugt eine neue Folge, welche die Differenz zweier Folgen darstellt
set_intersection	Erzeugt eine neue Folge, welche die Schnittmenge zweier Folgen darstellt
set_symmetric_difference	Erzeugt eine neue Folge, welche die gegenseitige Differenz zweier Folgen darstellt
set_union	Erzeugt eine neue Folge, welche die Vereinigungsmenge zweier Folgen darstellt

Tabelle 14-13– Mengenalgorithmen

<b>Minimum/Maximum-Algorithmen (min/max algorithms)</b>	
<i>Name</i>	<i>Bedeutung</i>
max	gibt das größere von zwei Elementen zurück
min	gibt das kleinere von zwei Elementen zurück
max_element	sucht den Maximalwert in einer Folge
min_element	Sucht den Minimalwert in einer Folge

Tabelle 14-14– Minimum/Maximum-Algorithmen

<b>Mischalgorithmen (shuffle algorithms)</b>	
<i>Name</i>	<i>Bedeutung</i>
random_shuffle	Ordnet die Elemente einer Folge in einer zufälligen Reihenfolge an

Tabelle 14-15– Mischalgorithmen

<b>Permutationsalgorithmen (permutation algorithms)</b>	
<i>Name</i>	<i>Bedeutung</i>
next_permutation	Erzeugt eine Permutationenfolge
prev_permutation	Erzeugt Permutationen in umgekehrter Reihenfolge

Tabelle 14-16– Permutationsalgorithmen

<b>Rotierungsalgorithmen (rotating algorithms)</b>	
<i>Name</i>	<i>Bedeutung</i>
rotate	Rotiert die Elemente in einer Folge um einen Punkt
rotate_copy	Kopiert eine Folge und rotiert dabei die Elemente um einen Punkt

Tabelle 14-17– Rotierungsalgorithmen

<b>Sortieralgorithmen (sorting algorithms)</b>	
<i>Name</i>	<i>Bedeutung</i>
partial_sort	Sortiert die n kleinsten Elemente einer Folge
partial_sort_copy	Kopiert die n kleinsten Elemente einer Folge und sortiert sie dabei
sort	Sortiert eine Folge
stable_sort	Sortiert eine Folge

*Tabelle 14-18– Sortieralgorithmen*

<b>Suchalgorithmen (searching algorithms)</b>	
<i>Name</i>	<i>Bedeutung</i>
adjacent_find	Sucht aufeinanderfolgende, doppelte Elemente
binary_search	Prüft ob ein Element in einer sortierten Folge enthalten ist
find	Sucht ein Element, das dem Argument entspricht
find_if	Sucht ein Element, das einer Bedingung genügt
search	Sucht nach einer Unterfolge innerhalb einer Folge

*Tabelle 14-19– Suchalgorithmen*

<b>Tauschalgorithmen (swapping algorithms)</b>	
<i>Name</i>	<i>Bedeutung</i>
iter_swap	tauscht die Werte auf die zwei Iteratoren zeigen
swap	tauscht zwei Werte aus
swap_ranges	tauscht Werte aus zwei parallelen Folgen aus

*Tabelle 14-20– Tauschalgorithmen*

<b>Transformierungsalgorithmen (transforming algorithms)</b>	
<i>Name</i>	<i>Bedeutung</i>
transform	wandelt jedes Element um

*Tabelle 14-21– Transformierungsalgorithmen*

<b>Umkehrungsalgorithmen (reversing algorithms)</b>	
<i>Name</i>	<i>Bedeutung</i>
reverse	Kehrt die Reihenfolge der Elemente in einer Folge um
reverse_copy	Kopiert eine Folge und kehrt dabei die Reihenfolge der Elemente um

*Tabelle 14-22– Umkehrungsalgorithmen*

<b>Vergleichsalgorithmen (comparing algorithms)</b>	
<i>Name</i>	<i>Bedeutung</i>
equal	Überprüft zwei Folgen auf Gleichheit
lexicographical_compare	Vergleicht zwei Folgen auf ihre Reihenfolge
mismatch	Sucht die erste Nichtentsprechung in parallelen Folgen

*Tabelle 14-23– Vergleichsalgorithmen*

<b>Zählalgorithmen (counting algorithms)</b>	
<i>Name</i>	<i>Bedeutung</i>
count	Zählt die Elemente, die dem Wert entsprechen

<b>Zählalgorithmen (counting algorithms)</b>	
<i>Name</i>	<i>Bedeutung</i>
count_if	Zählt die Elemente, die einer Bedingung entsprechen

Tabelle 14-24– Zählalgorithmen

Die in den Syntaxangaben verwendeten Iterator-Bezeichnungen haben folgende Bedeutung:

<b>Iteratortyp</b>	<b>Beschreibung</b>	<b>Unterstützende Container</b>
Read	Iterator, über den ein Element gelesen werden kann	<b>Deque</b> <b>List</b> <b>Map</b> <b>Multimap</b> <b>Multiset</b> <b>Set</b> <b>Vector</b>
Write	Iterator, über den ein Element geschrieben werden kann	<b>Deque</b> <b>List</b> <b>Map</b> <b>Multimap</b> <b>Multiset</b> <b>Set</b> <b>Vector</b>
Forward	Kombination aus <b>read_iterator</b> und <b>write_iterator</b> , kann sich auf dem Container nur vorwärts bewegen  <i>Definiert: operator++</i>	<b>Deque</b> <b>List</b> <b>Map</b> <b>Multimap</b> <b>Multiset</b> <b>Set</b> <b>Vector</b>
Bidirectional	Erweiterung des <b>forward_iterator</b> , kann sich auf dem Container vorwärts und rückwärts bewegen  <i>Definiert: operator ++, operator --</i>	<b>Deque</b> <b>List</b> <b>Map</b> <b>Multimap</b> <b>Multiset</b> <b>Set</b> <b>Vector</b>
RandomAccess	Erweiterung des <b>bidirectional_iterator</b> , kann sich auf dem Container vorwärts und rückwärts bewegen , sowie beliebig springen  <i>Definiert: operator ++, operator --, operator+, operator-, operator()</i>	<b>Deque</b> <b>Vector</b>

## 14.1 accumulate

Der generische Algorithmus **accumulate** „summiert“ alle Datenelemente einer gegebenen Folge gemäß der übergebenen Funktion auf und gibt das Ergebnis zurück:

Syntax:

```
T accumulate (read_iterator first,
              read_iterator last,
              T init);

T accumulate (read_iterator first,
              read_iterator last,
              T init,
              binär_Operation binop);
```

Typ:	Berechnungsalgorithmus (math algorithm)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren <b>operator+</b> , <b>operator=</b>
Rückgabe:	Ergebnis der Berechnung über alle Elemente der Folge

Der Algorithmus verwendet einen temporären Iterator **iter** um die Folge parallel über insgesamt **last-first** Elemente zu durchlaufen. Die Elemente werden dabei über das folgende Schema miteinander verknüpft:

$$\mathbf{init = init \ binop1 \ *iter}$$

Der Unterschied der beiden **accumulate**-Varianten besteht darin, dass die erste Form den Plus-Operator zur Addition verwendet, bzw. die entsprechende, überladene Operatorfunktion **operator+** für selbstdefinierte Klassen.

Die zweite Form hingegen benutzt statt des Plus-Operators die zu übergebene Binäroperation **binop**. Dadurch ist z.B. auch ein multiplizieren der Folge möglich.

Über den Initialwert **init** kann ein Startwert für die Verarbeitung vorgegeben werden, die Variable dient gleichzeitig als Zwischenspeicher.

**Achtung:** es erfolgt keine automatische Initialisierung der Variablen **init** durch den Algorithmus.

Das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.



```
//=====
// PROGRAMM: ALGORITHM_BSP_01
//=====
```

```
#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>
#include <numeric>

using namespace std;

void main (void)
{
    vector<double> aVect;
    double fErgebnis = 0.0;

    aVect.push_back (7.0);
    aVect.push_back (4.0);
    aVect.push_back (3.2);
    aVect.push_back (1.1);

    fErgebnis = accumulate (aVect.begin(), aVect.end(), 0.0);
    cout << "Ergebnis: " << fErgebnis << endl;
}
```

```
//=====
// PROGRAMM: ALGORITHM_BSP_02
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>
#include <numeric>

using namespace std;

double fnMult (double fInitial, double fElement)
{
    return fInitial * fElement;
}

void main (void)
{
    vector<double> aVect;
    double fErgebnis = 0.0;

    aVect.push_back (7.0);
    aVect.push_back (4.0);
    aVect.push_back (3.2);
    aVect.push_back (1.1);

    fErgebnis = accumulate (aVect.begin(), aVect.end(), 1.0, fnMult);
    cout << "Ergebnis: " << fErgebnis << endl;
}
```



## 14.2 adjacent\_difference

Der generische Algorithmus **adjacent\_difference** bildet die Differenz benachbarter Elemente und speichert diese in einer Ergebnisfolge ab. Eine Rückspeicherung in das Ausgangsintervall ist möglich.

Syntax:

```
write_iterator adjacent_difference (read_iterator first,
                                   read_iterator last,
                                   write_iterator output);

write_iterator adjacent_difference (read_iterator first,
                                   read_iterator last,
                                   write_iterator output,
                                   binär_Operation binop);
```

Typ:	Berechnungsalgorithmus (math algorithm)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren <b>operator-</b> , <b>operator=</b>
Rückgabe:	Iterator hinter das letzte Element der Ergebnisfolge

Der Unterschied der beiden **adjacent\_difference**-Varianten besteht darin, dass die erste Form den Minus-Operator zur Differenzbildung verwendet, bzw. die entsprechende, überladene Operatorfunktion **operator-** für selbstdefinierte Klassen.

Die zweite Form hingegen benutzt statt des Minus-Operators die zu übergebene Binäroperation **binop**. Dadurch sind auch anderen Berechnungen über benachbarte Elemente möglich.

**Achtung:** das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** die Folge, in welche die Ausgabe erfolgt, muss bereits hinreichend Platz zur Aufnahme der Elemente reserviert haben. Das Kopieren erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**.



```
//=====
// PROGRAMM: ALGORITHM_BSP_03
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>
#include <numeric>

using namespace std;

void main (void)
{
```

```

vector<int> aEin;

aEin.push_back (1);
aEin.push_back (2);
aEin.push_back (4);
aEin.push_back (66);

vector<int> aAus (aEin.size()); // groß genug machen

adjacent_difference (aEin.begin(),aEin.end(), aAus.begin());

for (int i=0; i<aAus.size(); i++)
{
    cout << i << ". Wert: " << aAus[i] << endl;
}
}

```

```

//=====
// PROGRAMM: ALGORITHM_BSP_04
//=====

#include <iomanip>
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>

using namespace std;

int fnMul(int nEins, int nZwei)
{
    return nEins * nZwei;
}

void main (void)
{
    vector<int> aEin;

    aEin.push_back (1);
    aEin.push_back (2);
    aEin.push_back (4);
    aEin.push_back (66);

    vector<int> aAus (aEin.size()); // groß genug machen

    adjacent_difference(aEin.begin(),aEin.end(),aAus.begin(),fnMul);

    for (int i=0; i<aAus.size(); i++)
    {
        cout << i << ". Wert: " << aAus[i] << endl;
    }
}

```



### 14.3 adjacent\_find

Der generische Algorithmus **adjacent\_find** ermittelt die Position gleicher, aufeinanderfolgender Elemente in einer Folge von Elementen.

Syntax:

```
read_iterator adjacent_find (read_iterator first,
                             read_iterator last);

read_iterator adjacent_find (read_iterator first,
                             read_iterator last,
                             binär_Bewertung binfunc);
```

Typ:	Suchalgorithmus (searching algorithm)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren
Rückgabe:	Iterator auf die erste gefundene Folge, bzw. <b>last</b> wenn keine Folge gefunden wurde

Der Unterschied der beiden **adjacent\_find**-Varianten besteht darin, dass die erste Form den Vergleichsoperator verwendet, bzw. die entsprechende, überladene Operatorfunktion **operator==** für selbstdefinierte Klassen.

Die zweite Form hingegen benutzt statt des Vergleichsoperators die zu übergebene, binäre Bewertungsfunktion **binfunc**. Dadurch sind auch andere Bewertungsformen möglich.

**Achtung:** das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.



```
//=====
// PROGRAMM: ALGORITHM_BSP_05
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<int> aEins (10);
    vector<int>::iterator aI;

    for (int i=0; i<10; i++)
    {
        aEins[i] = i; // aEins enthält 0123456789
    }

    aI = adjacent_find (aEins.begin(),aEins.end());
    if (aI != aEins.end())
```

```

    {
        cout << "Folge gefunden: " << *aI << endl;
    }
    else
    {
        cout << "keine Folge gefunden";
    }

    aEins[4] = 3; // jetzt enthält aEins = 0123356789

    aI = adjacent_find (aEins.begin(),aEins.end());
    if (aI != aEins.end())
    {
        cout << "Folge gefunden: " << *aI << endl;
    }
    else
    {
        cout << "keine Folge gefunden";
    }
}

```

```

//=====
// PROGRAMM: ALGORITHM_BSP_06
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

char* aText [] = {"Donald", "Duck", "Duck", "Dagobert", "Daniel"};

//-----
// gibt 1 zurück, wenn zwei Texte gleich sind. Hier kann jede
// beliebige Bewertungsfunktion stehen, solange als Ergebnis 1
// oder 0 geliefert wird
//-----
int fnVgl (char *sEins, char *sZwei)
{
    return !(::strcmp (sEins, sZwei));
}

void main (void)
{
    vector<char *> aName;
    vector<char *>::iterator aI;

    //-----
    // Anzahl der Texte im Array oben ermitteln = Länge des Arrays
    // (ist ein Array of char*) dividiert durch die Länge eines
    // Pointers
    //-----
    int nTextCount = sizeof(aText) / sizeof(aText[0]);

    for (int i=0; i<nTextCount; i++)
    {
        aName.push_back (aText[i]);
    }

    aI = adjacent_find (aName.begin(), aName.end(), fnVgl);

```



```
if (aI != aName.end())
{
    cout << " gleiche Namen gefunden: " << *aI << " und "
        << *(aI+1) << endl;
}
}
```

## 14.4 binary\_search

Der generische Algorithmus **binary\_search** ermittelt ob eine gesuchtes Datenelement in einer sortierten Folge von Elementen enthalten ist.

Syntax:

```
bool binary_search (forward_iterator first,
                  forward_iterator last,
                  const T& value);

bool binary_search (forward_iterator first,
                  forward_iterator last,
                  const T& value,
                  Vergleichsfunktion compare);
```

Typ:	Suchalgorithmus (searching algorithm)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren Folge muss sortiert sein <b>operator&lt;</b> , <b>operator==</b>
Rückgabe:	boolescher Wert, der besagt, ob das gesuchte Element in der Folge enthalten ist

Der Unterschied der beiden **binary\_search**-Varianten besteht darin, dass die erste Form voraussetzt, dass die zu durchsuchende Folge mit dem Kleiner-als-Operator sortiert ist, bzw. zur Sortierung die entsprechende, überladene Operatorfunktion **operator<** für selbstdefinierte Klassen verwendet wurde.

Die zweite Form hingegen benutzt statt des Kleiner-als-Operators die zu übergebene Vergleichsfunktion **compare**. Dadurch sind auch andere Sortierformen möglich.

**Achtung:** das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

```
//=====
// PROGRAMM: ALGORITHM_BSP_07
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<int> aFolge (1000);
    bool bFound = false;

    for (int i=0; i<1000; i++)
    {
        aFolge[i] = i; // aFolge enthält 0..999
    }
}
```



```

}

bFound = binary_search (aFolge.begin(), aFolge.end(), 555);
if (bFound)
{
    cout << "555 wurde gefunden: " << endl;
}

bFound = binary_search (aFolge.begin(), aFolge.end(), 1111);
if (bFound)
{
    cout << "1111 wurde gefunden: " << endl;
}
}

```



```

//=====
// PROGRAMM: ALGORITHM_BSP_08
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

char* aText [4] = {"Dagobert", "Daisy" , "Daniel", "Donald"};

//-----
// gibt 1 zurück, wenn zwei Texte sEins kleiner ist als sZwei
//-----
int fnCmp (const char *sEins, const char *sZwei)
{
    int nErgebnis = ::strcmp (sEins, sZwei);
    return nErgebnis < 0 ? 1 : 0;
}

void main (void)
{
    vector<char *> aName;

    for (int i=0; i<4; i++)
    {
        aName.push_back (aText[i]);
    }

    if (binary_search (aName.begin(), aName.end(), "Daisy", fnCmp))
    {
        cout << "Daisy gefunden" << endl;
    }
}

```

## 14.5 copy

Der generische Algorithmus **copy** kopiert eine Folge von einem Bereich in einen anderen Bereich gleicher Größe.

Syntax:

```
write_iterator copy (read_iterator first,
                    read_iterator last,
                    write_iterator output);
```

Typ:	Kopieralgorithmus (copying algorithm)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren Beide Folgen müssen die gleiche Länge besitzen <b>operator=</b>
Rückgabe:	Iterator der hinter das letzte kopierte Element in der Zielfolge zeigt

Die bereits vorhandenen Elemente des Zielbereiches werden mittels des Zuweisungsoperators **operator=** überschrieben. Dies ist insbesondere für selbstdefinierte Klassen relevant. Ist hier der **operator=** so überladen, dass nicht alle Zustände des Objektes zugewiesen werden, so gehen diese fehlenden Eigenschaft auch bei Zuweisung über den generischen Algorithmus **copy** verloren.

**Achtung:** das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** die Folge, in welche die Ausgabe erfolgt, muss bereits hinreichend Platz zur Aufnahme der Elemente reserviert haben. Das Kopieren erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**.

```
//=====
// PROGRAMM: ALGORITHM_BSP_09
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<double> aQuelle (100);
    vector<double> aZiel(10);
    int i;

    for (i=0; i<100; i++)
    {
        aQuelle[i] = i * 3.14;
    }

    copy (aQuelle.begin()+5, aQuelle.begin()+15, aZiel.begin());
```



```
for (i=0; i<10; i++)
{
    cout << aZiel[i] << endl;
}
}
```

## 14.6 copy\_backward

Der generische Algorithmus **copy\_backward** kopiert eine Folge von einem Bereich in einen anderen Bereich gleicher Größe. Der Algorithmus entspricht dem **copy**, geht dabei aber vom Endpunkt aus.

Syntax:

```
bidirectional_iterator copy_backward
    (bidirectional_iterator first,
     bidirectional_iterator last,
     bidirectional_iterator output);
```

Typ:	Kopieralgorithmus (copying algorithm)
Zeitbedarf:	Linear
Platzbedarf:	Konstant
Voraussetzung:	Container unterstützt Iteratoren Beide Folgen müssen die gleiche Länge besitzen <b>operator=</b>
Rückgabe:	Iterator der vor das letzte kopierte Element in der Zielfolge zeigt

Die bereits vorhandenen Elemente des Zielbereiches werden mittels des Zuweisungsoperators **operator=** überschrieben. Dies ist insbesondere für selbstdefinierte Klassen relevant. Ist hier der **operator=** so überladen, dass nicht alle Zustände des Objektes zugewiesen werden, so gehen diese fehlenden Eigenschaft auch bei Zuweisung über den generischen Algorithmus **copy\_backward** verloren.

**Achtung:** das Element **first** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge. Für der Ziel ist der Endpunkt anzugeben, da rückwärts kopiert wird. Die Reihenfolge der Elemente bleibt unberührt.

**Achtung:** die Folge, in welche die Ausgabe erfolgt, muss bereits hinreichend Platz zur Aufnahme der Elemente reserviert haben. Das Kopieren erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**.

```
//=====
// PROGRAMM: ALGORITHM_BSP_10
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<double> aQuelle (100);
    vector<double> aZiel(10);
    int i;

    for (i=0; i<100; i++)
```



```
{
    aQuelle[i] = i * 3.14;
}

copy_backward (aQuelle.begin()+5, aQuelle.begin()+15,
              aZiel.end());

for (i=0; i<10; i++)
{
    cout << aZiel[i] << endl;
}
}
```

## 14.7 count

Der generische Algorithmus **count** zählt die Elemente einer Folge, die einem vorgegebenen Element entsprechen.

Syntax (Standard):

```
void count (read_iterator first,
           read_iterator last,
           T& value,
           Size& anz);
```

Syntax (Microsoft):

```
Size count (read_iterator first,
           read_iterator last,
           T& value);
```

Typ:	Zählalgorithmus (counting algorithm)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren
	<b>operator==</b>
Rückgabe:	keine

Die Elemente der über **first** bis **last** definierten Folge werden mittels des Vergleichsoperators **operator==** mit **value** verglichen. Ergibt der Vergleich **true**, so wird der Wert von **anz** um Eins erhöht.

**Achtung:** es erfolgt keine automatische Initialisierung der Zählvariablen **anz** durch den Algorithmus.

Das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

```
//=====
// PROGRAMM: ALGORITHM_BSP_11
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<int> aFolge (100);
    int nAnzahl = 0;

    for (int i=0; i<100; i++)
    {
        aFolge[i] = i % 4;
    }

    //-----
    // Umsetzung gemäß Standard
```



```
//-----  
// count (aFolge.begin(), aFolge.end(), 1, nAnzahl);  
  
//-----  
// Umsetzung Microsoft  
//-----  
nAnzahl = count (aFolge.begin(), aFolge.end(), 1);  
  
cout << "Anzahl der Elemente mit Wert 1: " << nAnzahl << endl;  
}
```

## 14.8 count\_if

Der generische Algorithmus **count\_if** zählt die Elemente einer Folge, die bei der Bewertung durch eine unäre Bewertungsfunktion der Bedingung für **true** entsprechen.

Syntax (Standard):

```
void count_if (read_iterator first,
              read_iterator last,
              unär_Bewertung func,
              Size& anz);
```

Syntax (Microsoft):

```
Size count_if (read_iterator first,
              read_iterator last,
              unär_Bewertung func);
```

Typ:	Zählalgorithmus (counting algorithm)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren
Rückgabe:	keine

Die Elemente der über **first** bis **last** definierten Folge werden mittels der unären Bewertungsfunktion **func** bewertet. Ergibt die Bewertung **true**, so wird der Wert von **anz** um Eins erhöht.

**Achtung:** es erfolgt keine automatische Initialisierung der Zählvariablen **anz** durch den Algorithmus.

Das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

```
//=====
// PROGRAMM: ALGORITHM_BSP_12
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

int fneven (int nElement)
{
    int nErgebnis = nElement % 2;
    return !nErgebnis;
}

void main (void)
{
    vector<int> aFolge (100);
    int nAnzahl = 0;

    for (int i=0; i<100; i++)
```



```
{
    aFolge[i] = i;
}

//-----
// Umsetzung gemäß Standard
//-----
// count_if (aFolge.begin(), aFolge.end(), fneven, nAnzahl);

//-----
// Umsetzung Microsoft
//-----
nAnzahl = count_if (aFolge.begin(), aFolge.end(), fneven);
cout << "Anzahl der geraden Elemente: " << nAnzahl << endl;
}
```

## 14.9 equal

Der generische Algorithmus **equal** vergleicht, ob zwei Folgen gleicher Größe miteinander übereinstimmen. Dafür werden die enthaltenen Elemente der Reihenfolge nach miteinander verglichen.

Syntax:

```
bool equal (read_iterator first1,
           read_iterator last1,
           read_iterator first2);

bool equal (read_iterator first1,
           read_iterator last1,
           read_iterator first2,
           binär_Bewertung binfunc);
```

Typ:	Vergleichsalgorithmus (comparing algorithm)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren Beide Folgen müssen die gleiche Länge haben
Rückgabe:	<b>operator==</b> boolescher Wert, der angibt, ob die Folgen die gleichen Elemente in der gleichen Reihenfolge enthalten

Der Unterschied der beiden **equal**-Varianten besteht darin, dass die erste Form den Vergleichsoperator **operator==** verwendet, um die Elemente der beiden Folgen zu vergleichen. Für selbstdefinierte Klassen wird die entsprechende, überladene Operatorfunktion **operator==** verwendet.

Die zweite Form hingegen benutzt statt des Vergleichsoperators die zu übergebene binäre Bewertungsfunktion **binfunc**. Dadurch sind auch andere Vergleichsformen möglich.

**Achtung:** das Element **last1** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

```
//=====
// PROGRAMM: ALGORITHM_BSP_13
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<int> aFolge1 (10);
    vector<int> aFolge2 (10);

    for (int i=0; i<10; i++)
```



```
{
    aFolge1[i] = i;
    aFolge2[i] = i;
}

if (equal (aFolge1.begin(), aFolge1.end(), aFolge2.begin()))
{
    cout << "Die Folgen sind gleich" << endl;
}
}
```



```
//=====
// PROGRAMM: ALGORITHM_BSP_14
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

bool fnNear (int nEins, int nZwei)
{
    int nDiff = nEins - nZwei;
    return ((nDiff >= -1) && (nDiff <= 1));
}

void main (void)
{
    vector<int> aOne (10);
    vector<int> aTwo (10);

    for (int i=0; i<10; i++)
    {
        aOne[i] = i;
        aTwo[i] = i+1;
    }

    if (equal (aOne.begin(), aOne.end(), aTwo.begin(), fnNear))
    {
        cout << "Die Folgen sind annähernd gleich" << endl;
    }
}
```

## 14.10 equal\_range

Der generische Algorithmus **equal\_range** ermittelt die obere und untere Grenze eines Bereiches. Die häufigste Anwendung innerhalb der STL ist die Bestimmung der Positionen, an denen ein Element sortiert eingefügt werden kann.

Syntax:

```
pair<forward_iterator, forward_iterator>
    equal_range (forward_iterator first,
                forward_iterator last,
                const T& value);

pair<forward_iterator, forward_iterator>
    equal_range (forward_iterator first,
                forward_iterator last,
                const T& value,
                Vergleichsfunktion compare);
```

Typ:	Begrenzungsalgorithmus (bounding algorithm)
Zeitbedarf:	$O \log(n)$ bei Iteratoren mit wahlfreiem Zugriff sonst $O(n)$
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren Folge muss sortiert sein <b>operator&lt;, operator==</b>
Rückgabe:	Iteratoren- <b>pair</b> , welches auf die unterste und oberste Grenze verweist, an denen ein bestimmtes Element eingefügt werden kann.

Der Unterschied der beiden **equal\_range**-Varianten besteht darin, dass die erste Form voraussetzt, dass die Folge mit dem Vergleichsoperator **operator<** sortiert wurde. Für selbstdefinierte Klassen muss die entsprechende, überladene Operatorfunktion **operator<** verwendet worden sein.

Die zweite Form hingegen setzt statt des Vergleichsoperators **operator<** die Sortierung mit der zu übergebenen Vergleichsfunktion **compare** voraus. Dadurch lassen sich beliebig sortierte Folgen verarbeiten.

**Achtung:** das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

```
//=====
// PROGRAMM: ALGORITHM_BSP_15
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<int> aFolge;
```



```
pair <vector<int>::iterator, vector<int>::iterator> aBereich;

aFolge.push_back(0);
aFolge.push_back(0);
aFolge.push_back(1);
aFolge.push_back(1);
aFolge.push_back(2);
aFolge.push_back(2);

aBereich = equal_range (aFolge.begin(), aFolge.end(), 1);
cout << "Eine 2 kann eingefügt werden von Index: "
      << aBereich.first - aFolge.begin()
      << " bis Index: "
      << aBereich.second - aFolge.begin() << endl;
}
```

## 14.11 fill

Der generische Algorithmus **fill** setzt jedes Element einer Folge auf den übergebenen Wert.

Syntax:

```
void fill (forward_iterator first,
          forward_iterator last,
          const T& value);
```

Typ:	Füllalgorithmus (filling algorithm)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren
Rückgabe:	keine

Weist jedem Element der Folge mittels des Zuweisungsoperators **operator=** den über **value** vorgegebenen Wert zu. Dies ist insbesondere für selbstdefinierte Klassen relevant. Ist hier der **operator=** so überladen, dass nicht alle Zustände des Objektes zugewiesen werden, so gehen diese fehlenden Eigenschaft auch bei Zuweisung über den generischen Algorithmus **fill** verloren.

**Achtung:** das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.

```
//=====
// PROGRAMM: ALGORITHM_BSP_16
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<double> aTest (10);
    int i;

    for (i=0; i<10; i++)
    {
        aTest[i] = i * 3.14;
    }

    fill (aTest.begin(), aTest.end(), 1.0);

    for (i=0; i<10; i++)
    {
```



```
    cout << aTest[i] << endl;  
  }  
}
```

## 14.12 fill\_n

Der generische Algorithmus **fill\_n** setzt jedes Element einer Folge auf den übergebenen Wert.

Syntax:

```
void fill_n (write_iterator first,
            Size n,
            const T& value);
```

Typ:	Füllalgorithmus (filling algorithm)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren
Rückgabe:	keine

Weist den ersten **n** Elementen der bei **first** beginnenden Folge mittels des Zuweisungsoperators **operator=** den über **value** vorgegebenen Wert zu. Dies ist insbesondere für selbstdefinierte Klassen relevant. Ist hier der **operator=** so überladen, dass nicht alle Zustände des Objektes zugewiesen werden, so gehen diese fehlenden Eigenschaft auch bei Zuweisung über den generischen Algorithmus **fill\_n** verloren.

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.

```
//=====
// PROGRAMM: ALGORITHM_BSP_17
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<double> aTest (10);
    int i;

    for (i=0; i<10; i++)
    {
        aTest[i] = i * 3.14;
    }

    fill_n (aTest.begin(), aTest.size()-2, 1.0);

    for (i=0; i<10; i++)
    {
        cout << aTest[i] << endl;
    }
}
```



## 14.13 find

Der generische Algorithmus **find** ermittelt die Position eines vorgegebenen Elementes in einer Folge.

Syntax:

```
read_iterator find (read_iterator first,
                  read_iterator last,
                  const T& value);
```

Typ:	Suchalgorithmus (searching algorithm)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren
Rückgabe:	Iterator auf die erste gefundene Element, bzw. <b>last</b> wenn kein Element gefunden wurde

Der Algorithmus **find** sucht zwischen **first** und **last** nach dem Element **value**. Verwendet wird dazu der Vergleichsoperator, bzw. die entsprechende, überladene Operatorfunktion **operator==** für selbstdefinierte Klassen.

**Achtung:** das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.



```
//=====
// PROGRAMM: ALGORITHM_BSP_18
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<int> aEins (100);
    vector<int>::iterator aI;

    for (int i=0; i<100; i++)
    {
        aEins[i] = i;
    }

    aI = find (aEins.begin(), aEins.end(), 66);
    if (aI != aEins.end())
    {
        cout << "Wert 66 gefunden" << endl;
    }
}
```

## 14.14 find\_if

Der generische Algorithmus **find\_if** ermittelt die Position eines Elementes in einer Folge anhand einer Bedingung.

Syntax:

```
read_iterator find_if (read_iterator first,
                     read_iterator last,
                     unär_Bewertung func);
```

Typ:	Suchalgorithmus (searching algorithm)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren
Rückgabe:	Iterator auf die erste gefundene Element, bzw. <b>last</b> wenn kein Element gefunden wurde

Der Algorithmus **find** sucht zwischen **first** und **last** nach einem Element welches der Bedingung in der unären Bewertungsfunktion **func** genügt.

**Achtung:** das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

```
//=====
// PROGRAMM: ALGORITHM_BSP_19
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

char* aText [] = {"Donald", "Duck", "Duck", "Dagobert", "Daniel"};

//-----
// gibt 1 zurück, wenn zwei Texte gleich sind. Hier kann jede
// beliebige Bewertungsfunktion stehen, solange als Ergebnis 1
// oder 0 geliefert wird
//-----
int fnVgl (char *sEins)
{
    return !(::strcmp (sEins, "Duck"));
}

void main (void)
{
    vector<char *> aName;
    vector<char *>::iterator aI;

    //-----
    // Anzahl der Texte im Array oben ermitteln = Länge des Arrays
    // (ist ein Array of char*) dividiert durch die Länge eines
    // Pointers
    //-----
    int nTextCount = sizeof(aText) / sizeof(aText[0]);
```



```
for (int i=0; i<nTextCount; i++)
{
    aName.push_back (aText[i]);
}

aI = find_if (aName.begin(), aName.end(), fnVgl);

if (aI != aName.end())
{
    cout << "Name gefunden: " << *aI << endl;
}
}
```

## 14.15 for\_each

Der generische Algorithmus **for\_each** führt eine Funktion auf jedem Datenelement der Folge aus.

Syntax:

```
Unär_funktion for_each (read_iterator first,
                       read_iterator last,
                       unär_funktion func);
```

Typ:	Anwendungsalgorithmus (applying algorithm)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren Funktion muss ein Element vom korrekten Typ liefern
Rückgabe:	Iterator auf die erste gefundene Element, bzw. <b>last</b> wenn kein Element gefunden wurde

Der Algorithmus **for\_each** führt die unäre Funktion **func** auf jedem Element zwischen **first** und **last** aus.

Weist das Ergebnis mittels des Zuweisungsoperators **operator=** den über die zu. Die Zuweisung des Wertes über den **operator=** ist insbesondere für selbstdefinierte Klassen relevant. Ist hier der **operator=** so überladen, dass nicht alle Zustände des Objektes zugewiesen werden, so gehen diese fehlenden Eigenschaft auch bei Zuweisung über den generischen Algorithmus **for\_each** verloren.

**Achtung:** das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.

```
//=====
// PROGRAMM: ALGORITHM_BSP_20
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void fnUp (char *sText)
{
    cout << "Vorher: " << sText << endl;
    int len = ::strlen(sText);
    for (int i=0; i<len; i++)
    {
        switch (sText[i])
        {
            case 'ä': sText[i] = 'Ä'; break;
        }
    }
}
```



```
        case 'ö': sText[i] = 'Ö'; break;
        case 'ü': sText[i] = 'Ü'; break;
        default : sText[i] = toupper(sText[i]);
    }
}
cout << "Nachher: " << sText << endl;
}

void main (void)
{
    vector<char *> aVector;
    int i;
    char aText1[256] = "Dübel";
    char aText2[256] = "Schraube";
    char aText3[256] = "Nagel";

    aVector.push_back (aText1);
    aVector.push_back (aText2);
    aVector.push_back (aText3);

    for_each (aVector.begin(),aVector.end(), fnUp);

    for (i=0; i<aVector.size(); i++)
    {
        cout << (i+1) << ". Text nach Funktion: " << aVector[i] << endl;
    }
}
```

## 14.16 generate

Der generische Algorithmus **generate** setzt jedes Element einer Folge auf den über eine Funktion ermittelten (generierten) Wert.

Syntax:

```
void generate (forward_iterator first,
              forward_iterator last,
              Generatorfunktion genfunc);
```

Typ:	Generierungsalgorithmus (generating algorithm)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren Generatorfunktion muss ein Element vom korrekten Typ liefern
Rückgabe:	<b>operator=</b> keine

Weist jedem Element der Folge mittels des Zuweisungsoperators **operator=** den über die Generatorfunktion **genfunc** ermittelten Wert zu. **genfunc** ist eine Parameterlose Funktion, die einen Wert des in der Containerklasse verwalteten Typs **T** zurückliefern muss. Die Zuweisung des Wertes über den **operator=** ist insbesondere für selbstdefinierte Klassen relevant. Ist hier der **operator=** so überladen, dass nicht alle Zustände des Objektes zugewiesen werden, so gehen diese fehlenden Eigenschaft auch bei Zuweisung über den generischen Algorithmus **generate** verloren.

**Achtung:** das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.

```
//=====
// PROGRAMM: ALGORITHM_BSP_21
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>
#include <stdlib.h>
#include <time.h>

using namespace std;

int myrandom ()
{
    return (int)(rand() % 20); // Zufallszahlen im Bereich 0..19 ermitteln
}

void main (void)
{
```



```

vector<int> aTest (10);

srand((unsigned)time(NULL)); // Zufallszahlengenerator initialisieren

generate (aTest.begin(), aTest.end(), myrandom);

for (int i=0; i<10; i++)
{
    cout << aTest[i] << endl;
}
}

```



```

//=====
// PROGRAMM: ALGORITHM_BSP_22
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

class Doppler
{
    int puffer;
public:
    Doppler() {puffer = 1;}
    int operator() () {int e=puffer; puffer*=2; return e;}
};

void main (void)
{
    vector<int> aTest (10);
    Doppler Berechnungsklasse;

    generate (aTest.begin(), aTest.end(), Berechnungsklasse);
    for (int i=0; i<10; i++)
    {
        cout << aTest[i] << endl;
    }
}

```

## 14.17 generate\_n

Der generische Algorithmus **generate** setzt jedes Element einer Folge auf den über eine Funktion ermittelten (generierten) Wert.

Syntax:

```
void generate (write_iterator first,
              Size anz,
              Generatorfunktion genfunc);
```

Typ:	Generierungsalgorithmus (generating algorithm)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren Generatorfunktion muss ein Element vom korrekten Typ liefern
Rückgabe:	<b>operator=</b> keine

Weist den ersten **anz** Elementen der bei **first** beginnenden Folge mittels des Zuweisungsoperators **operator=** den über die Generatorfunktion **genfunc** ermittelten Wert zu. **genfunc** ist eine Parameterlose Funktion, die einen Wert des in der Containerklasse verwalteten Typs **T** zurückliefern muss. Die Zuweisung des Wertes über den **operator=** ist insbesondere für selbstdefinierte Klassen relevant. Ist hier der **operator=** so überladen, dass nicht alle Zustände des Objektes zugewiesen werden, so gehen diese fehlenden Eigenschaft auch bei Zuweisung über den generischen Algorithmus **generate\_n** verloren.

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.

```
//=====
// PROGRAMM: ALGORITHM_BSP_23
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>
#include <stdlib.h>
#include <time.h>

using namespace std;

int myrandom ()
{
    return (int)(rand() % 20); // Zufallszahlen im Bereich 0..19 ermitteln
}

void main (void)
{
    vector<int> aTest (10);
    int i;

    srand((unsigned)time(NULL)); // Zufallszahlengenerator initialisieren
```



```

for (i=0; i<10; i++)
{
    aTest[i] = 0;
}

generate_n (aTest.begin(), 5, myrandom);

for (i=0; i<10; i++)
{
    cout << aTest[i] << endl;
}
}

```



```

//=====
// PROGRAMM: ALGORITHM_BSP_24
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

class Doppler
{
    int puffer;
public:
    Doppler() {puffer = 1;}
    int operator() () {int e=puffer; puffer*=2; return e;}
};

void main (void)
{
    vector<int> aTest (10);
    Doppler Berechnungsklasse;
    int i;

    for (i=0; i<10; i++)
    {
        aTest[i] = 0;
    }

    generate_n (aTest.begin(), 5, Berechnungsklasse);

    for (i=0; i<10; i++)
    {
        cout << aTest[i] << endl;
    }
}

```

## 14.18 includes

Der generische Algorithmus **includes** ermittelt ob alle Elemente einer gegebenen Folge von Datenelementen (definiert durch **first2** und **last2**) in einer anderen Folge von Elementen (definiert durch **first1** und **last1**) enthalten sind.

Syntax:

```
bool includes (read_iterator first1,
              read_iterator last1,
              read_iterator first2,
              read_iterator last2);

bool includes (read_iterator first1,
              read_iterator last1,
              read_iterator first2,
              read_iterator last2,
              Vergleichsfunktion compare);
```

Typ:	Mengenalgorithmus (set algorithm)
Zeitbedarf:	Linear
Platzbedarf:	Konstant
Voraussetzung:	Container unterstützt Iteratoren Beide Folgen müssen gleichartig sortiert sein <b>operator&lt;, operator==</b>
Rückgabe:	boolescher Wert, der besagt, ob die Folge in der anderen Folge enthalten ist

Der Unterschied der beiden **includes**-Varianten besteht darin, dass die erste Form voraussetzt, dass die verwendeten Folgen mit dem Kleiner-als-Operator sortiert ist, bzw. zur Sortierung die entsprechende, überladene Operatorfunktion **operator<** für selbstdefinierte Klassen verwendet wurde.

Die zweite Form hingegen benutzt statt des Kleiner-als-Operators die zu übergebene Vergleichsfunktion **compare**. Dadurch sind auch andere Sortierformen möglich.

**Achtung:** die Elemente **last1** und **last2** gehören jeweils nicht mehr zum Vergleichsbereich der Folgen, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der jeweiligen Folge.

```
//=====
// PROGRAMM: ALGORITHM_BSP_25
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<int> aFolge (1000);
```



```

vector<int> aSuche;
bool bFound = false;

for (int i=0; i<1000; i++)
{
    aFolge[i] = i; // aFolge enthält 0..999
}

aSuche.push_back(5); // Reihenfolge ist hier wichtig, da Sortierung
aSuche.push_back(15); // vorausgesetzt wird
aSuche.push_back(555);

bFound = includes (aFolge.begin(), aFolge.end(),
                  aSuche.begin(),aSuche.end());
if (bFound)
{
    cout << "Die Folge ist enthalten" << endl;
}

aSuche.push_back(5555);

bFound = includes (aFolge.begin(), aFolge.end(),
                  aSuche.begin(),aSuche.end());
if (!bFound)
{
    cout << "Die Folge ist nicht enthalten" << endl;
}
}

```



```

//=====
// PROGRAMM: ALGORITHM_BSP_26
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

char* aText1 [4] = {"aa", "bb" , "cc", "dd"};
char* aText2 [2] = {"bb" , "dd"};

//-----
// gibt 1 zurück, wenn zwei Texte sEins kleiner ist als sZwei
//-----
int fnCmp (const char *sEins, const char *sZwei)
{
    int nErgebnis = ::strcmp (sEins, sZwei);
    return nErgebnis < 0 ? 1 : 0;
}

void main (void)
{
    vector<char *> aName;
    vector<char *> aSuch;
    int i;

    for (i=0; i<4; i++)
    {
        aName.push_back (aText1[i]);
    }
}

```

```
for (i=0; i<2; i++)
{
    aSuch.push_back (aText2[i]);
}

if (includes (aName.begin(), aName.end(),
             aSuch.begin(), aSuch.end(), fnCmp))
{
    cout << "Folge enthalten" << endl;
}
}
```

## 14.19 inner\_product

Der generische Algorithmus **inner\_product** bildet das innere Produkt zweier Folgen gemäß der übergebenen Funktionen und gibt das Ergebnis zurück:

Syntax:

```
T inner_product (read_iterator first1,
                read_iterator last1,
                read_iterator first2,
                T init);

T inner_product (read_iterator first1,
                read_iterator last1,
                read_iterator first2,
                T init,
                binär_Operation binop1,
                binär_Operation binop2);
```

Typ:	Berechnungsalgorithmus (math algorithm)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren <b>operator=, operator+, operator*</b>
Rückgabe:	Ergebnis der Berechnung über alle Elemente der Folge

Der Algorithmus verwendet zwei temporäre Iteratoren **iter1** und **iter2** um die beiden Folgen parallel über insgesamt **last1-first1** Elemente zu durchlaufen.

Die Elemente werden dabei über das folgende Schema gemäß der beiden binär-Operationen miteinander verknüpft:

$$\text{init} = \text{init} \text{ binop1 } (*\text{iter1} \text{ binop2 } *\text{iter2})$$

Der Unterschied der beiden **inner\_product**-Varianten besteht darin, dass die erste Form automatisch den Plus-Operator als erste binär-Operation und die Multiplikations-Operator als zweite binär-Operation verwendet, bzw. die entsprechenden, überladenen Operatorfunktionen **operator+** und **operator\*** für selbstdefinierte Klassen.

Die zweite Form hingegen benutzt statt des Plus-Operators die zu übergebene Binäroperation **binop1** und statt des Multiplikations-Operator die zu übergebene Binäroperation **binop2**. Dadurch sind auch andere Verknüpfungen zwischen den Folgen möglich.

Über den Initialwert **init** kann ein Startwert für die Verarbeitung vorgegeben werden, die Variable dient gleichzeitig als Zwischenspeicher.

**Achtung:** es erfolgt keine automatische Initialisierung der Variablen **init** durch den Algorithmus.

Das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.

```
//=====
// PROGRAMM: ALGORITHM_BSP_27
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>
#include <numeric>

using namespace std;

void main (void)
{
    vector<double> aVect1;
    vector<double> aVect2;
    double fErgebnis = 0.0;

    aVect1.push_back (7.0);
    aVect1.push_back (4.0);
    aVect1.push_back (3.2);

    aVect2.push_back (1.0);
    aVect2.push_back (2.0);
    aVect2.push_back (3.2);

    fErgebnis = inner_product (aVect1.begin(), aVect1.end(),
                               aVect2.begin(), 0.0);
    cout << "Ergebnis: " << fErgebnis << endl;
}
```



```
//=====
// PROGRAMM: ALGORITHM_BSP_28
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>
#include <numeric>

using namespace std;

double fndbl (double fEins, double fZwei)
{
    return fEins * fZwei * 2.0;
}

double fndivtrpl (double fEins, double fZwei)
{
    return fEins / fZwei * 3.0;
}

void main (void)
{
    vector<double> aVect1;
    vector<double> aVect2;
```



```
double fErgebnis = 0.0;

aVect1.push_back (7.0);
aVect1.push_back (4.0);
aVect1.push_back (3.2);

aVect2.push_back (1.0);
aVect2.push_back (2.0);
aVect2.push_back (3.2);

fErgebnis = inner_product (aVect1.begin(), aVect1.end(),
                           aVect1.begin(), 1.0, fndbl, fndivtrpl);
cout << "Ergebnis: " << fErgebnis << endl;
}
```

## 14.20 inplace\_merge

Der generische Algorithmus **inplace\_merge** vereinigt zwei sortierte Teilfolgen zu einer sortierten Folge.

Syntax:

```
void inplace_merge (bidirectional_iterator first,
                  bidirectional_iterator middle,
                  bidirectional_iterator last);

void inplace_merge (bidirectional_iterator first,
                  bidirectional_iterator middle,
                  bidirectional_iterator last,
                  Vergleichsfunktion compare);
```

Typ:	Einsortierungsalgorithmus (merging algorithm)
Zeitbedarf:	linear wenn genügend Platz vorhanden ist, sonst $N \log(N)$
Platzbedarf:	<b>last</b> – <b>first</b> wenn vorhanden, um Zeit zu sparen, sonst konstant zu Lasten des Zeitbedarfs
Voraussetzung:	Container unterstützt Iteratoren Folgen sind gleichartig sortiert <b>operator=</b> , <b>operator&lt;</b> , <b>operator==</b>
Rückgabe:	keine

Der Algorithmus fasst die beiden sortierten Teilfolgen **first** bis **middle** und **middle** bis **last** zu einer sortierten Folge zusammen.

Wenn beide Folgen die gleichen Elemente enthalten, werden die Werte der ersten Teilfolge als erste gespeichert.

Der Unterschied der beiden **inplace\_merge**-Varianten besteht darin, dass die erste Form voraussetzt, dass die Teilfolgen mit dem Kleiner-als-Operator **operator<** sortiert sind. Für selbstdefinierte Klassen wird die entsprechende, überladene Operatorfunktion **operator<** verwendet.

Die zweite Form hingegen benutzt statt des Kleiner-als-Operators die zu übergebene Vergleichsfunktion **compare**. Dadurch sind auch andere Sortierungen verarbeitbar.

**Achtung:** Das Element **middle** gehört nicht mehr zum Vergleichsbereich der ersten und **last** gehört nicht mehr zum Vergleichsbereich der zweiten Teilfolge, wie bei vielen STL-Methoden zeigen die Iteratoren hinter den jeweils letzten Wert der Folge.

**Achtung:** die Folge, in welche die Ausgabe erfolgt, muss bereits hinreichend Platz zur Aufnahme der Elemente reserviert haben. Das Kopieren erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**.

```
//=====
// PROGRAMM: ALGORITHM_BSP_29
//=====

#include <iomanip.h>
```



```

#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<double> aVector;

    aVector.push_back (2.6); // erste Teilfolge
    aVector.push_back (4.0);
    aVector.push_back (5.2);

    aVector.push_back (1.0); // zweite Teilfolge
    aVector.push_back (2.0);
    aVector.push_back (3.2);
    aVector.push_back (4.2);

    inplace_merge (aVector.begin(), aVector.begin()+3, aVector.end());

    for (int i=0; i<aVector.size(); i++)
    {
        cout << aVector[i] << endl;
    }
}

```



```

//=====
// PROGRAMM: ALGORITHM_BSP_30
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

char* aText [] = {"aa", "BB", "cc", "AA", "bc"};

//-----
// gibt 1 zurück, wenn Text2 kleiner als Text 1 ist. Hier kann
// jede beliebige Bewertungsfunktion stehen, solange als
// Ergebnis 1 oder 0 geliefert wird
//-----
int fnComp (char *sEins, char *sZwei)
{
    return ::stricmp (sEins, sZwei) < 0 ? 1 : 0;
}

void main (void)
{
    vector<char *> aName;
    int i;

    //-----
    // Anzahl der Texte im Array oben ermitteln = Länge des Arrays
    // (ist ein Array of char*) dividiert durch die Länge eines
    // Pointers
    //-----
    int nTextCount = sizeof(aText) / sizeof(aText[0]);
}

```

```
for (i=0; i<nTextCount; i++)
{
    aName.push_back (aText[i]);
}

inplace_merge (aName.begin(),aName.begin()+3, aName.end(), fnComp);

for (i=0; i< aName.size(); i++)
{
    cout << aName[i] << endl;
}
}
```

## 14.21 iter\_swap

Der generische Algorithmus **iter\_swap** tauscht die Inhalte zweier (über Iteratoren referenzierte) Objekte miteinander aus.

Syntax:

```
void iter_swap (forward_iterator one,
               forward_iterator two);
```

Typ:	Tauschalgorithmus (swapping algorithm)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren
	<b>operator=</b>
Rückgabe:	keine

Der Algorithmus verwendet den Zuweisungs-Operator **operator=**, um die Werte, auf welche die Iteratoren verweisen, auszutauschen. Dies ist insbesondere für selbstdefinierte Klassen relevant. Ist hier der **operator=** so überladen, daß nicht alle Zustände des Objektes zugewiesen werden, so gehen diese fehlenden Eigenschaft auch bei Zuweisung über **iter\_swap** verloren.

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.



```
//=====
// PROGRAMM: ALGORITHM_BSP_31
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>
#include <numeric>

using namespace std;

void main (void)
{
    vector<int> aVect;
    int i;

    for (i=0; i<10; i++)
    {
        aVect.push_back (i);
    }

    iter_swap (aVect.begin(), aVect.end()-1);

    for (i=0; i<10; i++)
    {
        cout << aVect[i] << endl;
    }
}
```

## 14.22 lexicographical\_compare

Mit dem generischen Algorithmus **lexicographical\_compare** können zwei Folgen hinsichtlich ihrer lexikographischen (=alphabetischen) Reihenfolge verglichen werden.

Syntax:

```
bool lexicographical_compare (read_iterator first1,
                             read_iterator last1,
                             read_iterator first2,
                             read_iterator last2);

bool lexicographical_compare (read_iterator first1,
                             read_iterator last1,
                             read_iterator first2,
                             read_iterator last2,
                             Vergleichsfunktion compare);
```

Typ:	Vergleichsalgorithmus (comparing algorithm)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren <b>operator&lt;, operator==</b>
Rückgabe:	boolescher Wert., der angibt, ob die Folgen in lexikographischer Reihenfolge sind.

Der Algorithmus durchläuft die beiden Folgen, die über die Iteratoren-Paare **first1 / last1** und **first2 / last2** definiert werden. Der Rückgabewert ist **true**, wenn der Inhalt der ersten Folge kleiner ist als jener der zweiten Folge. Die Rückgabe ist ebenfalls **true**, wenn das Ende der ersten vor dem Ende der zweiten Folge erreicht wurde, sonst **false**.

Der Unterschied der beiden **lexicographical\_compare**-Varianten besteht darin, dass die erste Form den Kleiner-als-Operator **operator<** verwendet, um die Elemente der beiden Folgen zu vergleichen. Für selbstdefinierte Klassen wird die entsprechende, überladene Operatorfunktion **operator<** verwendet.

Die zweite Form hingegen benutzt statt des Vergleichsoperators die zu übergebene Vergleichsfunktion **compare**. Dadurch sind auch andere Vergleichsformen möglich.

**Achtung:** die Elemente **last1** und **last2** gehören nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigen sie hinter den letzten Wert der jeweiligen Folge.

```
//=====
// PROGRAMM: ALGORITHM_BSP_32
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>
```



```

using namespace std;

void main (void)
{
    vector<char *> aVect1;
    vector<char *> aVect2;
    bool bResult;

    aVect1.push_back ("Duck");
    aVect1.push_back ("Donald");

    aVect2.push_back ("Duck");
    aVect2.push_back ("Dagobert");

    bResult = lexicographical_compare (aVect1.begin(), aVect1.end()-1,
                                       aVect2.begin(), aVect2.end()-1);

    if (bResult)
    {
        cout << "aVect1 ist lexikographisch kleiner" << endl;
    }
    else
    {
        cout << "aVect2 ist lexikographisch kleiner" << endl;
    }
}

```



```

//=====
// PROGRAMM: ALGORITHM_BSP_33
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

// Umlaute entfernen
void fnChangeUmlaute (char *sText)
{
    int nLen, i;
    nLen = strlen (sText);
    for (i=0; i<nLen; i++)
    {
        switch (sText[i])
        {
            case 'Ä': case 'ä':
            case 'Ö': case 'ö':
            case 'Ü': case 'ü':
            case 'ß': sText[i] = '_';
                    break;
        }
    }
}

bool fnIgnoreUmlaute (char* sOriginal1, char* sOriginal2)
{
    bool bRC = false;

    // Da die Originale nicht verändert werden dürfen,
    // hier erstmal Kopien anlegen

```

```
char *sCopy1 = new char[strlen(sOriginal1)+1];
char *sCopy2 = new char[strlen(sOriginal2)+1];
strcpy (sCopy1, sOriginal1);
strcpy (sCopy2, sOriginal2);

fnChangeUmlaute (sCopy1);
fnChangeUmlaute (sCopy2);

bRC = strcmp (sCopy1, sCopy2) < 0 ? true : false;
delete[] sCopy1;
delete[] sCopy2;
return bRC;
}

void main (void)
{
    vector<char *> aVect1;
    vector<char *> aVect2;
    bool bResult;

    aVect1.push_back ("ÄÄC");
    aVect1.push_back ("A");
    aVect2.push_back ("ÖÖB");
    aVect2.push_back ("A");

    bResult = lexicographical_compare (aVect1.begin(), aVect1.end()-1,
                                       aVect2.begin(), aVect2.end()-1);

    if (bResult)
    {
        cout << "Standard: aVect1 ist lexikographisch kleiner" << endl;
    }
    else
    {
        cout << "Standard: aVect2 ist lexikographisch kleiner" << endl;
    }

    bResult = lexicographical_compare (aVect1.begin(), aVect1.end()-1,
                                       aVect2.begin(), aVect2.end()-1,
                                       fnIgnoreUmlaute);

    if (bResult)
    {
        cout << "Eigene: aVect1 ist lexikographisch kleiner" << endl;
    }
    else
    {
        cout << "Eigene: aVect2 ist lexikographisch kleiner" << endl;
    }
}
```

## 14.23 lower\_bound

Der generische Algorithmus **lower\_bound** ermittelt die untere Grenze eines Bereiches. Die häufigste Anwendung innerhalb der STL ist die Bestimmung der Position, an denen ein Element sortiert eingefügt werden kann.

Syntax:

```
forward_iterator lower_bound (forward_iterator first,
                             forward_iterator last,
                             const T& value);

forward_iterator lower_bound (forward_iterator first,
                             forward_iterator last,
                             const T& value,
                             Vergleichsfunktion compare);
```

Typ:	Begrenzungsalgorithmus (bounding algorithm)
Zeitbedarf:	$O \log (n)$ bei Iteratoren mit wahlfreiem Zugriff sonst $O(n)$
Platzbedarf:	Konstant
Voraussetzung:	Container unterstützt Iteratoren Folge muss sortiert sein <b>operator&lt;, operator==</b>
Rückgabe:	Iterator, welcher auf die unterste Grenze verweist, an denen ein bestimmtes Element eingefügt werden kann.

Der Unterschied der beiden **lower\_bound**-Varianten besteht darin, dass die erste Form die Sortierung mit dem Kleiner-als-Operator **operator<** voraussetzt. Für selbstdefinierte Klassen wird die entsprechende, überladene Operatorfunktion **operator<** benutzt.

Die zweite Form hingegen setzt statt des Kleiner-als-Operators die Sortierung mit der Vergleichsfunktion **compare** voraus. Dadurch sind auch andere Sortierungen realisierbar.

**Achtung:** das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.



```
//=====
// PROGRAMM: ALGORITHM_BSP_34
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<int> aFolge;
    vector<int>::iterator aLower;

    aFolge.push_back(0);
```

```
aFolge.push_back(0);
aFolge.push_back(1);
aFolge.push_back(1);
aFolge.push_back(2);
aFolge.push_back(2);

aLower = lower_bound (aFolge.begin(), aFolge.end(), 1);
cout << "Eine 1 kann eingefügt werden an Index: "
      << aLower - aFolge.begin() << endl;
}
```

## 14.24 make\_heap

Wandelt eine Folge in einen Heap um.

Syntax:

```
void make_heap (randomaccess_iterator first,
               randomaccess_iterator last);

void make_heap (randomaccess_iterator first,
               randomaccess_iterator last,
               Vergleichsfunktion compare);
```

Typ:	Heapalgorithmen (heap algorithms)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Random-Access-Iteratoren
Rückgabe:	<b>operator&lt;, operator==</b> keine

Iteratoren mit wahlfreiem Zugriff sind nur für die Container-Klassen **vector** und **deque** definiert, so dass auch nur aus diesen Containern heraus ein **heap** erzeugt werden kann.

Der Unterschied der beiden **make\_heap**-Varianten besteht darin, dass die erste Form für die Anordnung im **heap** den Kleiner-als-Operator **operator<** verwendet. Für selbstdefinierte Klassen wird die entsprechende, überladene Operatorfunktion **operator<** benutzt.

Die zweite Form hingegen ruft statt des Kleiner-als-Operators die zu übergebene Vergleichsfunktion **compare** auf. Dadurch sind auch andere Sortierungen auf dem **heap** realisierbar.



```
//=====
// PROGRAMM: ALGORITHM_BSP_35
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<int> aFolge;
    int i;

    aFolge.push_back(123);
    aFolge.push_back(2);
    aFolge.push_back(345);
    aFolge.push_back(23);
    aFolge.push_back(4);
    aFolge.push_back(-4);
```

```

cout << "----- Folge zu Beginn -----" << endl;
for (i=0; i<6; i++)
{
    cout << "Folge: " << aFolge[i] << endl;
}
cout << endl;

make_heap (aFolge.begin(), aFolge.end());

cout << "----- Folge nach make_heap -----" << endl;
for (i=0; i<6; i++)
{
    cout << "Folge: " << aFolge[i] << endl;
}
cout << endl;

cout << "----- HEAP -----" << endl;
for (i=aFolge.size(); i>=1; i--)
{
    cout << "Heap: " << aFolge[0] << endl;
    pop_heap (aFolge.begin(), aFolge.begin()+i);
}
cout << endl;

cout << "----- Folge nach Heap-Abbau -----" << endl;
for (i=0; i<6; i++)
{
    cout << "Folge: " << aFolge[i] << endl;
}
cout << endl;
}

```

```

//=====
// PROGRAMM: ALGORITHM_BSP_36
//=====

#include <iomanip>
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

bool fnbigger (int i, int j)
{
    return i > j;
}

void main (void)
{
    vector<int> aFolge;
    int i;

    aFolge.push_back(123);
    aFolge.push_back(2);
    aFolge.push_back(345);
    aFolge.push_back(23);
    aFolge.push_back(4);
    aFolge.push_back(-4);

    for (i=0; i<6; i++)

```



```
{
    cout << "Folge: " << aFolge[i] << endl;
}
cout << endl;

make_heap (aFolge.begin(), aFolge.end(), fnbigger);

for (i=aFolge.size(); i>=1; i--)
{
    cout << "Heap: " << aFolge[0] << endl;
    pop_heap (aFolge.begin(), aFolge.begin()+i, fnbigger);
}
}
```

## 14.25 max

Gibt das Maximum zweier Werte zurück.

Syntax (Standard):

```
const T& max (const T& value1,
             const T& value2);

const T& max (const T& value1,
             const T& value2,
             Vergleichsfunktion compare);
```

Syntax (Microsoft):

Algorithmus ist nicht definiert (Stand: VC++ 6.0)

Typ:	Minimum/Maximum-Algorithmen (min/max algorithms)
Zeitbedarf:	konstant
Platzbedarf:	konstant
Voraussetzung:	<b>operator&lt;</b> , <b>operator==</b> , <b>operator=</b>
Rückgabe:	keine

Der Algorithmus gibt die Referenz auf den größeren der beiden übergebenen Werte zurück, bzw. auf den ersten Wert, wenn die Vergleichsfunktion feststellt, dass die Werte gleich sind.

Der Unterschied der beiden **max** -Varianten besteht darin, dass die erste Form den Vergleich mit dem Kleiner-als-Operator **operator<** vornimmt. Für selbstdefinierte Klassen wird die entsprechende, überladene Operatorfunktion **operator<** verwendet.

Die zweite Form hingegen benutzt statt des Kleiner-als-Operators die zu übergebene Vergleichsfunktion **compare**. Dadurch sind (z.B. bei eigenen Klassen) auch vom **operator<** abweichende Bewertungen möglich.

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.

**Achtung:** Der Visual C++ Compiler definiert das Template **max** im Rahmen der STL nicht. Die entsprechenden Templates sind jedoch sehr einfach und können daher sehr leicht in einem eigenen Headerfile hinzugefügt werden:

```
//=====
// PROGRAMM: STLMINMAX.H
//=====

#ifndef _STLMINMAX_H_
#define _STLMINMAX_H_

template <class T>
inline const T& max (const T& a, const T& b)
{
    return a < b ? b : a;
}
```



```

}

template <class T, class Compare>
inline const T& max (const T& a, const T& b, Compare comp)
{
    return comp(a, b) ? b : a;
}

template <class T>
inline const T& min (const T& a, const T& b)
{
    return b < a ? b : a;
}

template <class T, class Compare>
inline const T& min (const T& a, const T& b, Compare comp)
{
    return comp(b, a) ? b : a;
}
#endif

```



```

//=====
// PROGRAMM: ALGORITHM_BSP_37
//=====

#include <iomanip.h>
#include <iostream.h>
#include <algorithm>

// nur bei Microsoft-Compiler
#include "stlminmax.h"

using namespace std;

void main (void)
{
    cout << max (100, 1000) << endl;
    cout << max (1.2, 1.11) << endl;
    cout << max ('A', 'X') << endl;
}

```



```

//=====
// PROGRAMM: ALGORITHM_BSP_38
//=====

#include <iomanip.h>
#include <iostream.h>
#include <algorithm>

// nur bei Microsoft-Compiler
#include "stlminmax.h"

using namespace std;

bool mycompare (int a, int b)
{
    return (b<2*a)? true : false;
}

void main (void)
{
    cout << max (1, 2, mycompare) << endl;
}

```

}

## 14.26 max\_element

Sucht den Maximalwert in einer Folge

Syntax:

```
read_iterator max_element (read_iterator first,
                          read_iterator last);

read_iterator max_element (read_iterator first,
                          read_iterator last);
                          Vergleichsfunktion compare);
```

Typ:	Minimum/Maximum-Algorithmen (min/max algorithms)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren
Rückgabe:	<b>operator&lt;, operator==</b> Iterator, der auf das maximale Element verweist

Der Algorithmus sucht das größte Element in der angegebenen Folge **first** / **last**.

Der Unterschied der beiden **max\_Element**-Varianten besteht darin, dass die erste Form den Vergleich der Elemente mit dem Kleiner-als-Operator **operator<** durchführt. Für selbstdefinierte Klassen wird die entsprechende, überladene Operatorfunktion **operator<** benutzt.

Die zweite Form hingegen verwendet statt des Kleiner-als-Operators die Vergleichsfunktion **compare**. Dadurch sind auch andere Bewertungen realisierbar.

**Achtung:** das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.



```
//=====
// PROGRAMM: ALGORITHM_BSP_39
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<int> aFolge;
    vector<int>::iterator aI;

    aFolge.push_back(1);
    aFolge.push_back(234);
    aFolge.push_back(4);
    aFolge.push_back(16);
    aFolge.push_back(43);
    aFolge.push_back(23);
```

```
aI = max_element(aFolge.begin(), aFolge.end());  
  
cout << *aI << endl;  
}
```

```
//=====  
// PROGRAMM: ALGORITHM_BSP_40  
//=====  
  
#include <iomanip.h>  
#include <iostream.h>  
#include <vector>  
#include <algorithm>  
  
using namespace std;  
  
bool mycompare (int a, int b)  
{  
    return (a>b)? true : false;  
}  
  
void main (void)  
{  
    vector<int> aFolge;  
    vector<int>::iterator aI;  
  
    aFolge.push_back(1);  
    aFolge.push_back(234);  
    aFolge.push_back(4);  
    aFolge.push_back(16);  
    aFolge.push_back(43);  
    aFolge.push_back(23);  
  
    aI = max_element(aFolge.begin(), aFolge.end(), mycompare);  
  
    cout << *aI << endl;  
}
```



## 14.27 merge

Der generische Algorithmus **merge** vereinigt zwei sortierte Folgen zu einer neuen, dritten sortierten Folge.

Syntax:

```
void merge (read_iterator first1,
           read_iterator last1,
           read_iterator first2,
           read_iterator last2,
           write_iterator output);

void merge (read_iterator first1,
           read_iterator last1,
           read_iterator first2,
           read_iterator last2,
           write_iterator output,
           Vergleichsfunktion compare);
```

Typ:	Einsortierungsalgorithmus (merging algorithm)
Zeitbedarf:	$O(N)$ mit $N=(last1-first1)+ (last2-first2)$
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren Folgen sind gleichartig sortiert <b>operator&lt;, operator==, operator=</b>
Rückgabe:	keine

Der Algorithmus fasst die beiden sortierten Folgen **first1 / last1** und **first2 / last2** zu einer dritten, ebenfalls sortierten Folge zusammen.

Wenn beide Folgen die gleichen Elemente enthalten, werden die Werte der ersten Teilfolge als erste gespeichert.

Der Unterschied der beiden **merge**-Varianten besteht darin, dass die erste Form voraussetzt, dass die Folgen mit dem Kleiner-als-Operator **operator<** sortiert sind. Für selbstdefinierte Klassen wird die entsprechende, überladene Operatorfunktion **operator<** verwendet.

Die zweite Form hingegen benutzt statt des Kleiner-als-Operators die zu übergebene Vergleichsfunktion **compare**. Dadurch sind auch andere Sortierungen verarbeitbar.

**Achtung:** Die Vereinigung sich überschneidender Intervalle ist nicht definiert. Die Elemente **last1** und **last2** gehört nicht mehr zum Bereich der Folgen, wie bei vielen STL-Methoden zeigen die Iteratoren hinter den jeweils letzten Wert der Folge.

**Achtung:** die Folge, in welche die Ausgabe erfolgt, muss bereits hinreichend Platz zur Aufnahme der Elemente reserviert haben. Das Kopieren erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**.



```
//=====
// PROGRAMM: ALGORITHM_BSP_41
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<double> aVector1;
    vector<double> aVector2;
    vector<double> aVector3;

    aVector1.push_back (2.6); // erste Folge
    aVector1.push_back (4.0);
    aVector1.push_back (5.2);

    aVector2.push_back (1.0); // zweite Folge
    aVector2.push_back (2.0);
    aVector2.push_back (3.2);
    aVector2.push_back (4.2);

    aVector3.resize(aVector1.size()+aVector2.size());

    merge (aVector1.begin(), aVector1.end(),
           aVector2.begin(), aVector2.end(),
           aVector3.begin());

    for (int i=0; i<aVector3.size(); i++)
    {
        cout << aVector3[i] << endl;
    }
}
```

**14.28 min**

Gibt das Minimum zweier Werte zurück.

Syntax (Standard):

```
const T& min (const T& value1,
             const T& value2);

const T& min (const T& value1,
             const T& value2,
             Vergleichsfunktion compare);
```

Syntax (Microsoft):

Algorithmus ist nicht definiert (Stand: VC++ 6.0)

Typ:	Minimum/Maximum-Algorithmen (min/max algorithms)
Zeitbedarf:	konstant
Platzbedarf:	konstant
Voraussetzung:	<b>operator&lt;</b> , <b>operator==</b> , <b>operator=</b>
Rückgabe:	keine

Der Algorithmus gibt die Referenz auf den kleineren der beiden übergebenen Werte zurück, bzw. auf den ersten Wert, wenn die Vergleichsfunktion feststellt, dass die Werte gleich sind.

Der Unterschied der beiden **min** -Varianten besteht darin, dass die erste Form den Vergleich mit dem Kleiner-als-Operator **operator<** vornimmt. Für selbstdefinierte Klassen wird die entsprechende, überladene Operatorfunktion **operator<** verwendet.

Die zweite Form hingegen benutzt statt des Kleiner-als-Operators die zu übergebene Vergleichsfunktion **compare**. Dadurch sind (z.B. bei eigenen Klassen) auch vom **operator<** abweichende Bewertungen möglich.

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.

**Achtung:** Der Visual C++ Compiler definiert das Template **min** im Rahmen der STL nicht. Die entsprechenden Templates sind jedoch sehr einfach und können daher sehr leicht in einem eigenen Headerfile hinzugefügt werden:



```
//=====
// PROGRAMM: STLMINMAX.H
//=====

#ifndef _STLMINMAX_H_
#define _STLMINMAX_H_

template <class T>
inline const T& max (const T& a, const T& b)
{
    return a < b ? b : a;
}
```

```

}

template <class T, class Compare>
inline const T& max (const T& a, const T& b, Compare comp)
{
    return comp(a, b) ? b : a;
}

template <class T>
inline const T& min (const T& a, const T& b)
{
    return b < a ? b : a;
}

template <class T, class Compare>
inline const T& min (const T& a, const T& b, Compare comp)
{
    return comp(b, a) ? b : a;
}
#endif

```

```

//=====
// PROGRAMM: ALGORITHM_BSP_42
//=====

#include <iomanip.h>
#include <iostream.h>
#include <algorithm>

// nur bei Microsoft-Compiler
#include "stlminmax.h"

using namespace std;

void main (void)
{
    cout << min (100, 1000) << endl;
    cout << min (1.2, 1.11) << endl;
    cout << min ('A', 'X') << endl;
}

```



```

//=====
// PROGRAMM: ALGORITHM_BSP_43
//=====

#include <iomanip.h>
#include <iostream.h>
#include <algorithm>

// nur bei Microsoft-Compiler
#include "stlminmax.h"

using namespace std;

bool mycompare (int a, int b)
{
    return (b<2*a)? true : false;
}

void main (void)
{
    cout << max (1, 2, mycompare) << endl;
}

```



```
}
```

## 14.29 min\_element

Sucht den Minimalwert in einer Folge

Syntax:

```
read_iterator min_element (read_iterator first,
                          read_iterator last);

read_iterator min_element (read_iterator first,
                          read_iterator last);
                          Vergleichsfunktion compare);
```

Typ:	Minimum/Maximum-Algorithmen (min/max algorithms)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren
Rückgabe:	<b>operator&lt;, operator==</b> Iterator, der auf das maximale Element verweist

Der Algorithmus sucht das kleinste Element in der angegebenen Folge **first** / **last**.

Der Unterschied der beiden **min\_Element**-Varianten besteht darin, dass die erste Form den Vergleich der Elemente mit dem Kleiner-als-Operator **operator<** durchführt. Für selbstdefinierte Klassen wird die entsprechende, überladene Operatorfunktion **operator<** benutzt.

Die zweite Form hingegen verwendet statt des Kleiner-als-Operators die Vergleichsfunktion **compare**. Dadurch sind auch andere Bewertungen realisierbar.

**Achtung:** das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

```
//=====
// PROGRAMM: ALGORITHM_BSP_44
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<int> aFolge;
    vector<int>::iterator aI;

    aFolge.push_back(1);
    aFolge.push_back(234);
    aFolge.push_back(4);
    aFolge.push_back(16);
    aFolge.push_back(43);
    aFolge.push_back(23);
```



```

    aI = min_element(aFolge.begin(), aFolge.end());

    cout << *aI << endl;
}

```



```

//=====
// PROGRAMM: ALGORITHM_BSP_45
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

bool mycompare (int a, int b)
{
    return (a>b)? true : false;
}

void main (void)
{
    vector<int> aFolge;
    vector<int>::iterator aI;

    aFolge.push_back(1);
    aFolge.push_back(234);
    aFolge.push_back(4);
    aFolge.push_back(16);
    aFolge.push_back(43);
    aFolge.push_back(23);

    aI = min_element(aFolge.begin(), aFolge.end(), mycompare);

    cout << *aI << endl;
}

```

### 14.30 mismatch

Der generische Algorithmus **mismatch** durchsucht zwei Folgen nach zwei Elementen, die nicht miteinander übereinstimmen. Dazu werden die Elemente der beiden Folgen paarweise miteinander verglichen. Der Algorithmus gibt ein Iteratoren-**pair** zurück, sowie zwei Elemente nicht miteinander übereinstimmen oder die Position **last1** in der ersten Folge erreicht wurde.

Syntax:

```
pair<read_iterator, read_iterator>
    mismatch (read_iterator first1,
             read_iterator last1,
             read_iterator first2);

pair<read_iterator, read_iterator>
    mismatch (read_iterator first1,
             read_iterator last1,
             read_iterator first2,
             Vergleichsfunktion compare);
```

Typ:	Vergleichsalgorithmus (comparing algorithm)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren Folgen haben gleiche Länge
Rückgabe:	<b>operator==</b> Iterator-Paar, welches auf nicht übereinstimmende Elemente in beiden Folgen zeigt.

Der Unterschied der beiden **mismatch**-Varianten besteht darin, dass die erste Form den Vergleichsoperator **operator==** verwendet, um die Elemente der beiden Folgen zu vergleichen. Für selbstdefinierte Klassen wird die entsprechende, überladene Operatorfunktion **operator==** verwendet.

Die zweite Form hingegen benutzt statt des Vergleichsoperators die zu übergebene Vergleichsfunktion **compare**. Dadurch sind auch andere Vergleichsformen möglich.

**Achtung:** das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

```
//=====
// PROGRAMM: ALGORITHM_BSP_46
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
```



```

vector<int> aFolge1 (10);
vector<int> aFolge2 (10);
pair<vector<int>::iterator, vector<int>::iterator> aErgebnis;

for (int i=0; i<10; i++)
{
    aFolge1[i] = i;
    aFolge2[i] = i;
}
aFolge2[4] = 17;

aErgebnis = mismatch(aFolge1.begin(), aFolge1.end(), aFolge2.begin());

if (aErgebnis.first == aFolge1.end() &&
    aErgebnis.second == aFolge2.end())
{
    cout << "Die Folgen sind gleich" << endl;
}
else
{
    cout << "Die Folgen sind ungleich" << endl;
    cout << *(aErgebnis.first) << endl;
    cout << *(aErgebnis.second) << endl;
}
}

```



```

//=====
// PROGRAMM: ALGORITHM_BSP_47
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

bool fnNear (int nEins, int nZwei)
{
    int nDiff = nEins - nZwei;
    return ((nDiff >= -1) && (nDiff <= 1));
}

void main (void)
{
    vector<int> aFolge1 (10);
    vector<int> aFolge2 (10);
    pair<vector<int>::iterator, vector<int>::iterator> aErgebnis;

    for (int i=0; i<10; i++)
    {
        aFolge1[i] = i;
        aFolge2[i] = i;
    }
    aFolge2[4] = 6; // Folge 1 enthält 4

    aErgebnis = mismatch(aFolge1.begin(), aFolge1.end(),
                        aFolge2.begin(), fnNear);

    if (aErgebnis.first == aFolge1.end() &&
        aErgebnis.second == aFolge2.end())
    {

```

```
        cout << "Die Folgen sind gleich" << endl;
    }
    else
    {
        cout << "Die Folgen sind ungleich" << endl;
        cout << *(aErgebnis.first) << endl;
        cout << *(aErgebnis.second) << endl;
    }

    aFolge2[4] = 5; // Folge 1 enthält 4
    aErgebnis = mismatch(aFolge1.begin(), aFolge1.end(),
                        aFolge2.begin(), fnNear);

    if (aErgebnis.first == aFolge1.end() &&
        aErgebnis.second == aFolge2.end())
    {
        cout << "Die Folgen sind gleich" << endl;
    }
    else
    {
        cout << "Die Folgen sind ungleich" << endl;
        cout << *(aErgebnis.first) << endl;
        cout << *(aErgebnis.second) << endl;
    }
}
```

### 14.31 next\_permutation

Der generische Algorithmus ordnet die Elemente einer Folge dergestalt um, dass sie die nächste, lexikographisch folgenden Permutation bilden.

Syntax:

```

bidirectional_iterator
    next_permutation (bidirectional_iterator first,
                     bidirectional_iterator last);

bidirectional_iterator
    next_permutation (bidirectional_iterator first,
                     bidirectional_iterator last);
                     Vergleichsfunktion compare);

```

Typ:	Permutationsalgorithmen (permutation algorithms)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren
Rückgabe:	<b>operator&lt;</b> , <b>operator==</b> , <b>operator=</b> <b>true</b> wenn eine neue Permutation gebildet werden konnte, <b>false</b> wenn es keine nächste Permutation gibt.

Der Algorithmus verändert die Reihenfolge der Elemente und bildet so die lexikographisch nächste Reihenfolge. Startet man mit der kleinsten Folge, können so alle möglichen Elementanordnungen (Permutationen) gebildet werden.

Der Unterschied der beiden **next\_permutation**-Varianten besteht darin, dass die erste Form den Vergleich der Elemente mit dem Kleiner-als-Operator **operator<** durchführt. Für selbstdefinierte Klassen wird die entsprechende, überladene Operatorfunktion **operator<** benutzt.

Die zweite Form hingegen verwendet statt des Kleiner-als-Operators die Vergleichsfunktion **compare**. Dadurch sind auch andere Bewertungen realisierbar.

**Achtung:** das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.

```

//=====
// PROGRAMM: ALGORITHM_BSP_48
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

```

```
void main (void)
{
    vector<int> aFolge;
    int i = 1;

    aFolge.push_back(1);
    aFolge.push_back(2);
    aFolge.push_back(3);
    aFolge.push_back(3);

    cout << setw(2) << i << ": "
         << aFolge[0] << " "
         << aFolge[1] << " "
         << aFolge[2] << " "
         << aFolge[3] << endl;

    while (next_permutation(aFolge.begin(), aFolge.end()))
    {
        i++;
        cout << setw(2) << i << ": "
             << aFolge[0] << " "
             << aFolge[1] << " "
             << aFolge[2] << " "
             << aFolge[3] << endl;
    }
}
```

## 14.32 nth\_element

Teilt die Elemente einer Folge am  $n$ -ten Element.

Syntax:

```
void nth_element (randomaccess_iterator first,
                 randomaccess_iterator nth,
                 randomaccess_iterator last);

void nth_element (randomaccess_iterator first,
                 randomaccess_iterator nth,
                 randomaccess_iterator last,
                 Vergleichsfunktion compare);
```

Typ:	Aufteilungsalgorithmen (partitioning algorithms)
Zeitbedarf:	$O(N)$ mit $N = (\text{last} - \text{first})$
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Random-Access-Iteratoren
Rückgabe:	<b>operator&lt;</b> , <b>operator==</b> , <b>operator=</b> <b>true</b> wenn eine neue Permutation gebildet werden konnte, <b>false</b> wenn es keine weitere Permutation gibt.

Iteratoren mit wahlfreiem Zugriff sind nur für die Container-Klassen **vector** und **deque** definiert.

Die Teilung erfolgt so, dass anschließend alle Elemente, die kleiner sind als der durch den Iterator **nth** referenzierte Wert, in der Folge links stehen, alle Werte größer oder gleich diesem Wert rechts davon.

Der Unterschied der beiden **nth\_element**-Varianten besteht darin, dass die erste Form den Vergleich der Elemente mit dem Kleiner-als-Operator **operator<** durchführt. Für selbstdefinierte Klassen wird die entsprechende, überladene Operatorfunktion **operator<** benutzt.

Die zweite Form hingegen verwendet statt des Kleiner-als-Operators die Vergleichsfunktion **compare**. Dadurch sind auch andere Sortierungen realisierbar.

**Achtung:** das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.

```
//=====
// PROGRAMM: ALGORITHM_BSP_49
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>
```

```
#include <conio.h>

using namespace std;

void main (void)
{
    vector<int> aFolge;
    int i = 1;

    aFolge.push_back(9);
    aFolge.push_back(3);
    aFolge.push_back(8);

    aFolge.push_back(5);
    aFolge.push_back(5);
    aFolge.push_back(4);

    aFolge.push_back(2);
    aFolge.push_back(7);
    aFolge.push_back(5);

    for (i=0; i<9; i++)
    {
        cout << aFolge[i] << endl;
    }
    cout << endl;

    nth_element (aFolge.begin(), aFolge.begin()+4, aFolge.end());

    for (i=0; i<9; i++)
    {
        cout << aFolge[i] << endl;
    }
    cout << endl;
}
```

### 14.33 partial\_sort

Sortiert die ersten  $n$  Elemente einer Folge.

Syntax:

```
void partial_sort (randomaccess_iterator first,
                  randomaccess_iterator middle,
                  randomaccess_iterator last);

void partial_sort (randomaccess_iterator first,
                  randomaccess_iterator middle,
                  randomaccess_iterator last,
                  Vergleichsfunktion compare);
```

Typ:	Sortieralgorithmen (sorting algorithms)
Zeitbedarf:	$(\mathbf{last} - \mathbf{first}) * \log N$
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Random-Access-Iteratoren
Rückgabe:	<b>operator&lt;, operator==, operator=</b> keine

Iteratoren mit wahlfreiem Zugriff sind nur für die Container-Klassen **vector** und **deque** definiert.

Sortiert werden nur die ersten  $n$  Elemente einer Folge, wobei  $n$  dem Ausdruck **middle – first** entspricht. Die restlichen Elemente der Folge **first / last** bleiben unsortiert. Die Werte werden aus der gesamten Folge entnommen, aber sowie die Sortierung die Position **middle** erreicht hat, wird die weitere Ausführung abgebrochen.

Der Unterschied der beiden **partial\_sort**-Varianten besteht darin, dass die erste Form den Vergleich der Elemente mit dem Kleiner-als-Operator **operator<** durchführt. Für selbstdefinierte Klassen wird die entsprechende, überladene Operatorfunktion **operator<** benutzt.

Die zweite Form hingegen verwendet statt des Kleiner-als-Operators die Vergleichsfunktion **compare**. Dadurch sind auch andere Sortierungen realisierbar.

**Achtung:** das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.

```
//=====
// PROGRAMM: ALGORITHM_BSP_50
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
```

```
#include <algorithm>
#include <conio.h>

using namespace std;

void main (void)
{
    vector<int> aFolge;
    int i = 1;

    aFolge.push_back(9);
    aFolge.push_back(3);
    aFolge.push_back(8);

    aFolge.push_back(5);
    aFolge.push_back(5);
    aFolge.push_back(4);

    aFolge.push_back(2);
    aFolge.push_back(7);
    aFolge.push_back(5);

    for (i=0; i<9; i++)
    {
        cout << aFolge[i] << endl;
    }
    cout << endl;

    partial_sort (aFolge.begin(), aFolge.begin()+4, aFolge.end());

    for (i=0; i<9; i++)
    {
        cout << aFolge[i] << endl;
    }
    cout << endl;
}
```

### 14.34 `partial_sort_copy`

Kopiert die ersten  $n$  Elemente einer Folge und sortiert das Ergebnis in eine andere Folge.

Syntax:

```
randomaccess_iterator partial_sort_copy
    (read_iterator first1,
     read_iterator last1,
     randomaccess_iterator first2,
     randomaccess_iterator last2);

randomaccess_iterator partial_sort_copy
    (read_iterator first1,
     read_iterator last1,
     randomaccess_iterator first2,
     randomaccess_iterator last2,
     Vergleichsfunktion compare);
```

Typ:	Sortieralgorithmen (sorting algorithms)
Zeitbedarf:	$(last - first) * \log N$
Platzbedarf:	konstant
Voraussetzung:	Ziel-Container unterstützt Random-Access-Iteratoren <b>operator&lt;</b> , <b>operator==</b> , <b>operator=</b>
Rückgabe:	Der kleinere der beiden Werte <b>last2</b> und <b>first2+n</b> wobei $n = last1 - first1$ ist

Iteratoren mit wahlfreiem Zugriff sind nur für die Container-Klassen **vector** und **deque** definiert.

Sortiert die Elemente **first1** / **last1** der ersten Folge und kopiert das Ergebnis in die Folge **first2** / **last2**.

Der Unterschied der beiden **partial\_sort\_copy**-Varianten besteht darin, dass die erste Form den Vergleich der Elemente mit dem Kleiner-als-Operator **operator<** durchführt. Für selbstdefinierte Klassen wird die entsprechende, überladene Operatorfunktion **operator<** benutzt.

Die zweite Form hingegen verwendet statt des Kleiner-als-Operators die Vergleichsfunktion **compare**. Dadurch sind auch andere Sortierungen realisierbar.

**Achtung:** Die Elemente **last1** und **last2** gehören nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigen sie hinter den letzten Wert der jeweiligen Folge.

**Achtung:** die Folge, in welche die Ausgabe erfolgt, muss bereits hinreichend Platz zur Aufnahme der Elemente reserviert haben. Das Kopieren erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**.



```
//=====
// PROGRAMM: ALGORITHM_BSP_51
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>
#include <conio.h>

using namespace std;

void main (void)
{
    vector<int> aFolge1;
    vector<int> aFolge2(5);
    int i = 1;

    aFolge1.push_back(9);
    aFolge1.push_back(3);
    aFolge1.push_back(8);
    aFolge1.push_back(5);
    aFolge1.push_back(5);
    aFolge1.push_back(4);
    aFolge1.push_back(2);
    aFolge1.push_back(7);

    for (i=0; i<8; i++)
    {
        cout << aFolge1[i] << endl;
    }
    cout << endl;

    partial_sort_copy (aFolge1.begin(), aFolge1.begin()+4,
                      aFolge2.begin(), aFolge2.begin()+4);

    for (i=0; i<8; i++)
    {
        cout << aFolge1[i] << endl;
    }
    cout << endl;

    for (i=0; i<4; i++)
    {
        cout << aFolge2[i] << endl;
    }
    cout << endl;
}
```

### 14.35 partial\_sum

Der generische Algorithmus **partial\_sum** bildet die forlaufende „Summe“ aller Datenelemente einer gegebenen Teilfolge und speichert das Ergebnis in einer anderen Folge ab:

Syntax:

```
write_iterator partial_sum (read_iterator first,
                           read_iterator last,
                           write_iterator output);

write_iterator partial_sum (read_iterator first,
                           read_iterator last,
                           write_iterator output,
                           binär_Operation binop);
```

Typ:	Berechnungsalgorithmus (math algorithm)
Zeitbedarf:	konstant
Platzbedarf:	linear
Voraussetzung:	Container unterstützt Iteratoren <b>operator+</b> , <b>operator=</b>
Rückgabe:	Iterator, der hinter das letzte Element der Summen-Bildung zeigt

Die Elemente werden dabei über das folgende Schema miteinander verknüpft:

$$\text{summe} = \text{summe} \text{ binop1 } * \text{iterator}$$

$$* \text{output} = \text{summe}$$

Der Unterschied der beiden **partial\_sum**-Varianten besteht darin, dass die erste Form den Plus-Operator zur Addition verwendet, bzw. die entsprechende, überladene Operatorfunktion **operator+** für selbstdefinierte Klassen.

Die zweite Form hingegen benutzt statt des Plus-Operators die zu übergebene Binäroperation **binop**. Dadurch ist z.B. auch ein multiplizieren der Folge möglich.

**Achtung:** Das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** die Folge, in welche die Ausgabe erfolgt, muss bereits hinreichend Platz zur Aufnahme der Elemente reserviert haben. Das Kopieren erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**.



```
//=====
// PROGRAMM: ALGORITHM_BSP_52
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>
#include <numeric>
```

```
using namespace std;

void main (void)
{
    vector<double> aVect;
    vector<double> aErg(4);
    int i;

    aVect.push_back (1.0);
    aVect.push_back (2.0);
    aVect.push_back (3.0);
    aVect.push_back (4.0);
    aVect.push_back (5.0);
    aVect.push_back (6.0);
    aVect.push_back (7.0);
    aVect.push_back (8.0);

    partial_sum (aVect.begin(), aVect.begin()+4, aErg.begin());

    for (i=0; i<aVect.size(); i++)
    {
        cout << aVect[i] << endl;
    }
    cout << endl;

    for (i=0; i<aErg.size(); i++)
    {
        cout << aErg[i] << endl;
    }
    cout << endl;
}
```

## 14.36 partition

Dieser generische Algorithmus teilt – unter Verwendung einer Bewertungsfunktion – die Elemente einer Folge in zwei Gruppen auf.

Syntax:

```
bidirectional_iterator
    partition (bidirectional_iterator first,
              bidirectional_iterator last,
              unär_Bewertung func);
```

Typ:	Aufteilungsalgorithmen (partitioning algorithms)
Zeitbedarf:	$O(N)$ mit $N = \text{last} - \text{first}$
Platzbedarf:	Linear
Voraussetzung:	Container unterstützt Iteratoren
Rückgabe:	<b>operator=</b> Iterator, der auf das erste Element der zweiten Folge zeigt

Die Elemente des Intervalls werden von **partition** dergestalt sortiert, dass alle Elemente, für welche die unäre Bewertungsfunktion **true** zurückgibt vor denen stehen, für die **false** zurückgegeben wird.

**Achtung:** Das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.

```
//=====
// PROGRAMM: ALGORITHM_BSP_53
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

bool greater500 (double x)
{
    return x > 500.0 ? true : false;
}

void main (void)
{
    vector<double> aVect;
    int i;

    aVect.push_back (134.0);
    aVect.push_back (235.0);
    aVect.push_back (983.0);
    aVect.push_back (774.0);
    aVect.push_back (345.0);
    aVect.push_back (677.0);
```

```
aVect.push_back (676.0);
aVect.push_back (338.0);

partition (aVect.begin(), aVect.end(), greater500);

for (i=0; i<aVect.size(); i++)
{
    cout << aVect[i] << endl;
}
cout << endl;

for (i=0; i<aVect.size(); i++)
{
    cout << aVect[i] << endl;
}
cout << endl;
}
```

## 14.37 pop\_heap

Der generische Algorithmus **pop\_heap**, entnimmt das erste Element aus einem **heap**.

Syntax:

```
void pop_heap (randomaccess_iterator first,
              randomaccess_iterator last);

void pop_heap (randomaccess_iterator first,
              randomaccess_iterator last,
              Vergleichsfunktion compare);
```

Typ:	Heapalgorithmen (heap algorithms)
Zeitbedarf:	2 x log (last – first)
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Random-Access-Iteratoren
Rückgabe:	<b>operator&lt;, operator==, operator=</b> Keine

Iteratoren mit wahlfreiem Zugriff sind nur für die Container-Klassen **vector** und **deque** definiert, so dass auch nur aus diesen Containern heraus ein **heap** erzeugt werden kann. Der **pop\_heap** Algorithmus tauscht, beginnend mit dem angegebenen Intervall **first** / **last** das erste und das letzte Element und macht anschließend den Bereich **first** / **last-1** zum neuen **heap**.

Der Unterschied der beiden **pop\_heap**-Varianten besteht darin, dass die erste Form für die Anordnung im **heap** den Kleiner-als-Operator **operator<** verwendet. Für selbstdefinierte Klassen wird die entsprechende, überladene Operatorfunktion **operator<** benutzt.

Die zweite Form hingegen ruft statt des Kleiner-als-Operators die zu übergebene Vergleichsfunktion **compare** auf. Dadurch sind auch andere Sortierungen auf dem **heap** realisierbar.

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.

```
//=====
// PROGRAMM: ALGORITHM_BSP_54
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<int> aFolge;
```

```
int i;

aFolge.push_back(123);
aFolge.push_back(2);
aFolge.push_back(345);
aFolge.push_back(23);
aFolge.push_back(4);
aFolge.push_back(-4);

for (i=0; i<6; i++)
{
    cout << "Folge: " << aFolge[i] << endl;
}
cout << endl;

make_heap (aFolge.begin(), aFolge.end());

for (i=aFolge.size(); i>=1; i--)
{
    cout << "Heap: " << aFolge[0] << endl;
    pop_heap (aFolge.begin(), aFolge.begin()+i);
}
}
```

## 14.38 prev\_permutation

Der generische Algorithmus ordnet die Elemente einer Folge dergestalt um, dass sie die vorherige, lexikographisch vorangehende Permutation bilden.

Syntax:

```
bool prev_permutation (bidirectional_iterator first,
                      bidirectional_iterator last);

bool prev_permutation (bidirectional_iterator first,
                      bidirectional_iterator last);
                      Vergleichsfunktion compare);
```

Typ:	Permutationsalgorithmen (permutation algorithms)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren
Rückgabe:	<b>operator&lt;, operator==, operator=</b> <b>true</b> wenn eine neue Permutation gebildet werden konnte, <b>false</b> wenn es keine weitere Permutation gibt.

Der Algorithmus verändert die Reihenfolge der Elemente und bildet so die lexikographisch vorangehende Reihenfolge. Startet man mit der größten Folge, können so alle möglichen Elementanordnungen (Permutationen) gebildet werden.

Der Unterschied der beiden **prev\_permutation**-Varianten besteht darin, dass die erste Form den Vergleich der Elemente mit dem Kleiner-als-Operator **operator<** durchführt. Für selbstdefinierte Klassen wird die entsprechende, überladene Operatorfunktion **operator<** benutzt.

Die zweite Form hingegen verwendet statt des Kleiner-als-Operators die Vergleichsfunktion **compare**. Dadurch sind auch andere Bewertungen realisierbar.

**Achtung:** das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.



**Achtung:** Die Implementation der Rückgabe bei der STL-Implementation von HP (im Microsoft-Compiler enthalten) arbeitet leider nicht entsprechend der STL-Definition, weshalb man, setzt man das next\_permutation-Beispiel für prev\_permutation um, in eine Endlosschleife gerät. Offenbar wird das Erreichen der letzten Permutationsform nicht korrekt erkannt. Die Implementation von Roguewave (im Borland-Compiler enthalten) zeigt dieses Verhalten nicht.

```
//=====
// PROGRAMM: ALGORITHM_BSP_55_1
// MICROSOFT
```

```
//=====
#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>
#include <conio.h>

using namespace std;

void main (void)
{
    vector<int> aFolge;
    int i = 1;

    aFolge.push_back(3);
    aFolge.push_back(3);
    aFolge.push_back(2);
    aFolge.push_back(1);

    cout << setw(2) << i << ": "
         << aFolge[0] << " "
         << aFolge[1] << " "
         << aFolge[2] << " "
         << aFolge[3] << endl;

    // while (prev_permutation(aFolge.begin(), aFolge.end()))
    for (int j=0; i<12; j++)
    {
        prev_permutation(aFolge.begin(), aFolge.end());
        i++;
        cout << setw(2) << i << ": "
             << aFolge[0] << " "
             << aFolge[1] << " "
             << aFolge[2] << " "
             << aFolge[3] << endl;
    }
}
```

```
//=====
// PROGRAMM: ALGORITHM_BSP_55_2
// BOLAND
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>
#include <conio.h>

using namespace std;

void main (void)
{
    vector<int> aFolge;
    int i = 1;

    aFolge.push_back(3);
    aFolge.push_back(3);
    aFolge.push_back(2);
    aFolge.push_back(1);
```

```
cout << setw(2) << i << ": "  
    << aFolge[0] << " "  
    << aFolge[1] << " "  
    << aFolge[2] << " "  
    << aFolge[3] << endl;  
  
while (prev_permutation(aFolge.begin(), aFolge.end()))  
{  
    i++;  
    cout << setw(2) << i << ": "  
        << aFolge[0] << " "  
        << aFolge[1] << " "  
        << aFolge[2] << " "  
        << aFolge[3] << endl;  
}  
}
```

## 14.39 push\_heap

Der generische Algorithmus **push\_heap** legt das letzte Element auf einem **heap** ab.

Syntax:

```
void push_heap (randomaccess_iterator first,
               randomaccess_iterator last);

void push_heap (randomaccess_iterator first,
               randomaccess_iterator last,
               Vergleichsfunktion compare);
```

Typ:	Heapalgorithmen (heap algorithms)
Zeitbedarf:	$2 \times \log(\text{last} - \text{first})$
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Random-Access-Iteratoren
Rückgabe:	<b>operator&lt;, operator==, operator=</b> Keine

Iteratoren mit wahlfreiem Zugriff sind nur für die Container-Klassen **vector** und **deque** definiert, so dass auch nur aus diesen Containern heraus ein **heap** erzeugt werden kann. Der **push\_heap** Algorithmus fügt, beginnend mit dem Intervall **first** / **last-1** das Element **last** so in den Heap ein, dass anschließend **first** / **last** ein Heap ist.

Der Unterschied der beiden **push\_heap**-Varianten besteht darin, dass die erste Form für die Anordnung im **heap** den Kleiner-als-Operator **operator<** verwendet. Für selbstdefinierte Klassen wird die entsprechende, überladene Operatorfunktion **operator<** benutzt.

Die zweite Form hingegen ruft statt des Kleiner-als-Operators die zu übergebene Vergleichsfunktion **compare** auf. Dadurch sind auch andere Sortierungen auf dem **heap** realisierbar.

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.

```
//=====
// PROGRAMM: ALGORITHM_BSP_56
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<int> aFolge;
    int i;
```

```
aFolge.push_back(123);
aFolge.push_back(2);
aFolge.push_back(345);
aFolge.push_back(23);
aFolge.push_back(4);
aFolge.push_back(-4);

for (i=0; i<aFolge.size(); i++)
{
    cout << "Folge: " << aFolge[i] << endl;
}
cout << endl;

make_heap (aFolge.begin(), aFolge.end());
aFolge.push_back(8888);
push_heap (aFolge.begin(), aFolge.end());

for (i=aFolge.size(); i>=1; i--)
{
    cout << "Heap: " << aFolge[0] << endl;
    pop_heap (aFolge.begin(), aFolge.begin()+i);
}
cout << endl;

for (i=0; i<aFolge.size(); i++)
{
    cout << "Folge: " << aFolge[i] << endl;
}
}
```

## 14.40 random\_shuffle

Ordnet die Elemente einer Folge in einer zufälligen Reihenfolge an.

Syntax:

```
void random_shuffle (randomaccess_iterator first,
                    randomaccess_iterator last);

void random_shuffle (randomaccess_iterator first,
                    randomaccess_iterator last,
                    Zufallszahlengenerator generator);
```

Typ:	Mischalgorithmen (shuffle algorithms)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Random-Access-Iteratoren
Rückgabe:	<b>operator=</b> Keine

Iteratoren mit wahlfreiem Zugriff sind nur für die Container-Klassen **vector** und **deque** definiert.

Der Unterschied der beiden **random\_shuffle**-Varianten besteht darin, dass die erste Form die Vertauschung der Elemente anhand des internen Zufallszahlengenerators durchführt. Die Vertauschung selbst wird durch Zuweisung mit dem Zuweisungs-Operator **operator=** durchführt. Für selbstdefinierte Klassen wird die entsprechende, überladene Operatorfunktion **operator=** benutzt.

Die zweite Form hingegen verwendet statt des internen Zufallszahlengenerators den Generator **generator**. Dadurch sind auch andere Formen realisierbar.

**Achtung:** Das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.

```
//=====
// PROGRAMM: ALGORITHM_BSP_57
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>
#include <time.h>

using namespace std;

//-----
// diese Klasse realisiert einen (sehr einfachen) Generator
// für Zufallszahlen
//-----
```

```
class NewRandomGenerator
{
    public:
        unsigned long operator() (unsigned long nBereich);
};

unsigned long NewRandomGenerator::operator() (unsigned long nBereich)
{
    return time(0) % nBereich;
}

//-----
// Hauptprogramm
//-----
void main (void)
{
    vector<int> aFolge;
    int i;

    aFolge.push_back(1);
    aFolge.push_back(2);
    aFolge.push_back(3);
    aFolge.push_back(4);
    aFolge.push_back(5);
    aFolge.push_back(6);

    random_shuffle (aFolge.begin(), aFolge.end(), NewRandomGenerator());

    for (i=0; i<aFolge.size(); i++)
    {
        cout << aFolge[i] << endl;
    }
}
```

## 14.41 remove

Der generische Algorithmus **remove** entfernt alle Elemente einer Folge, die mit einem vorgegebenen Wert **value** übereinstimmen.

Syntax:

```
forward_iterator remove (forward_iterator first,
                        forward_iterator last,
                        const T& value);
```

Typ:	Löschalgorithmen (removing algorithms)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren
Rückgabe:	Iterator der hinter das letzte Element der reduzierten Folge zeigt, also auf <b>last - n</b> wobei <b>n</b> die Anzahl der gelöschten Einträge ist.

**Achtung:** Das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** Der Algorithmus **remove** ändert die Größe der verarbeiteten Folge nicht, so dass die Elemente zwischen **last - n** und **last** als undefiniert zu betrachten sind.

```
//=====
// PROGRAMM: ALGORITHM_BSP_58
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<int> aFolge;
    vector<int>::iterator aI;
    vector<int>::iterator aErg;
    int i;

    aFolge.push_back(1);
    aFolge.push_back(2);
    aFolge.push_back(1);
    aFolge.push_back(3);
    aFolge.push_back(1);
    aFolge.push_back(4);

    for (i=0; i<aFolge.size(); i++)
    {
        cout << "Folge: " << aFolge[i] << endl;
    }
    cout << endl;

    aErg = remove (aFolge.begin(), aFolge.end(), 1);
```

```
for (aI=aFolge.begin(); aI<aErg; aI++)
{
    cout << (*aI) << endl;
}
cout << endl;

for (i=0; i<aFolge.size(); i++)
{
    cout << "Folge: " << aFolge[i] << endl;
}
}
```

## 14.42 remove\_copy

Der generische Algorithmus **remove\_copy** kopiert alle Elemente einer Folge, die mit einem vorgegebenen Wert **value** nicht übereinstimmen in eine neue Folge.

Syntax:

```
write_iterator remove_copy (read_iterator first,
                           read_iterator last,
                           write_iterator output,
                           const T& value);
```

Typ:	Löschalgorithmen (removing algorithms)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren
Rückgabe:	Iterator der hinter das letzte Element der Ergebnisfolge zeigt.

Der Algorithmus **remove\_copy** verändert die Ausgangsfolge nicht.

**Achtung:** Das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** die Folge, in welche die Ausgabe erfolgt, muss bereits hinreichend Platz zur Aufnahme der Elemente reserviert haben. Das Kopieren erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**.

```
//=====
// PROGRAMM: ALGORITHM_BSP_59
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<int> aFolge1;
    vector<int> aFolge2(10);
    vector<int>::iterator aErg;
    int i;

    aFolge1.push_back(1);
    aFolge1.push_back(2);
    aFolge1.push_back(1);
    aFolge1.push_back(3);
    aFolge1.push_back(1);
    aFolge1.push_back(4);

    for (i=0; i<aFolge1.size(); i++)
    {
        cout << "Folge1: " << aFolge1[i] << endl;
```

```
}
cout << endl;

aErg = remove_copy (aFolge1.begin(), aFolge1.end(),
                   aFolge2.begin(), 1);

for (i=0; i<aFolge1.size(); i++)
{
    cout << "Folge1: " << aFolge1[i] << endl;
}
cout << endl;

//-----
// Hier wäre aFolge2 immer noch 10 Einträge lang
// daher auf tatsächliche Größe gemäß remove_copy
// reduzieren
//-----
aFolge2.resize(aErg-aFolge2.begin());
for (i=0; i<aFolge2.size(); i++)
{
    cout << "Folge2: " << aFolge2[i] << endl;
}
cout << endl;
}
```

### 14.43 remove\_copy\_if

Der generische Algorithmus **remove\_copy\_if** kopiert alle Elemente einer Folge, mit Ausnahme derjenigen Elemente, die mit einer Bewertungsfunktion aussortiert werden.

Syntax:

```
write_iterator remove_copy_if (read_iterator first,
                              read_iterator last,
                              write_iterator output,
                              Vergleichsfunktion compare);
```

Typ:	Löschalgorithmen (removing algorithms)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren
Rückgabe:	<b>operator&lt;, operator==, operator=</b> Iterator der hinter das letzte Element der Ergebnisfolge zeigt.

Der Algorithmus **remove\_copy\_if** verändert die Ausgangsfolge nicht.

**Achtung:** Das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** die Folge, in welche die Ausgabe erfolgt, muss bereits hinreichend Platz zur Aufnahme der Elemente reserviert haben. Das Kopieren erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**.

```
//=====
// PROGRAMM: ALGORITHM_BSP_60
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

bool fnUngerade (int x)
{
    return x%2 ? true : false;
}

void main (void)
{
    vector<int> aFolge1;
    vector<int> aFolge2(10);
    vector<int>::iterator aErg;
    int i;

    aFolge1.push_back(1);
    aFolge1.push_back(2);
    aFolge1.push_back(1);
    aFolge1.push_back(3);
```

```
aFolge1.push_back(1);
aFolge1.push_back(4);

for (i=0; i<aFolge1.size(); i++)
{
    cout << "Folge1: " << aFolge1[i] << endl;
}
cout << endl;

aErg = remove_copy_if (aFolge1.begin(), aFolge1.end(),
                      aFolge2.begin(), fnUngerade);

for (i=0; i<aFolge1.size(); i++)
{
    cout << "Folge1: " << aFolge1[i] << endl;
}
cout << endl;

//-----
// Hier wäre aFolge2 immer noch 10 Einträge lang
// daher auf tatsächliche Größe gemäß remove_copy
// reduzieren
//-----
aFolge2.resize(aErg-aFolge2.begin());
for (i=0; i<aFolge2.size(); i++)
{
    cout << "Folge2: " << aFolge2[i] << endl;
}
cout << endl;
}
```

## 14.44 remove\_if

Der generische Algorithmus **remove\_if** entfernt, unter Zuhilfenahme einer Bewertungsfunktion, Elemente aus einer Folge.

Syntax:

```
forward_iterator remove_if (forward_iterator first,
                           forward_iterator last,
                           Vergleichsfunktion compare);
```

Typ:	Löschalgorithmen (removing algorithms)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren <b>operator&lt;, operator==</b>
Rückgabe:	Iterator der hinter das letzte Element der reduzierten Folge zeigt, also auf <b>last – n</b> wobei <b>n</b> die Anzahl der gelöschten Einträge ist.

**Achtung:** Das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** Der Algorithmus **remove\_if** ändert die Größe der verarbeiteten Folge nicht, so dass die Elemente zwischen **last – n** und **last** als undefiniert zu betrachten sind.



```
//=====
// PROGRAMM: ALGORITHM_BSP_61
//=====

#include <iomanip>
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

bool fnGerade (int x)
{
    return x%2 ? false : true;
}

void main (void)
{
    vector<int> aFolge;
    vector<int>::iterator aI;
    vector<int>::iterator aErg;
    int i;

    aFolge.push_back(10);
    aFolge.push_back(2);
    aFolge.push_back(112);
    aFolge.push_back(3);
    aFolge.push_back(123);
    aFolge.push_back(4);

    for (i=0; i<aFolge.size(); i++)
```

```
{
    cout << "Folge: " << aFolge[i] << endl;
}
cout << endl;

aErg = remove_if (aFolge.begin(), aFolge.end(), fnGerade);

for (aI=aFolge.begin(); aI<aErg; aI++)
{
    cout << (*aI) << endl;
}
cout << endl;

for (i=0; i<aFolge.size(); i++)
{
    cout << "Folge: " << aFolge[i] << endl;
}
}
```

## 14.45 replace

Der generische Algorithmus **replace** ersetzt alle Elemente einer Folge, die mit einem vorgegebenen Wert **value** übereinstimmen, durch einen anderen Wert.

Syntax:

```
void replace (forward_iterator first,
             forward_iterator last,
             const T& oldvalue,
             const T& newvalue);
```

Typ:	Ersetzungsalgorithmen (replacing algorithms)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren
Rückgabe:	<b>operator=</b> keine

**Achtung:** Das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.

```
//=====
// PROGRAMM: ALGORITHM_BSP_62
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<int> aFolge;
    int i;

    aFolge.push_back(1);
    aFolge.push_back(2);
    aFolge.push_back(1);
    aFolge.push_back(3);
    aFolge.push_back(1);
    aFolge.push_back(4);

    for (i=0; i<aFolge.size(); i++)
    {
        cout << "Folge: " << aFolge[i] << endl;
    }
    cout << endl;

    replace (aFolge.begin(), aFolge.end(), 1, 1000);
```

```
for (i=0; i<aFolge.size(); i++)
{
    cout << "Folge: " << aFolge[i] << endl;
}
}
```

## 14.46 replace\_copy

Der generische Algorithmus **replace\_copy** kopiert alle Elemente einer Folge, wobei die mit einem vorgegebenen Wert **value** übereinstimmenden Elemente durch andere Elemente ersetzt werden.

Syntax:

```
write_iterator replace_copy (read_iterator first,
                             read_iterator last,
                             write_iterator output,
                             const T& oldvalue,
                             const T& newvalue);
```

Typ:	Ersetzungsalgorithmen (replacing algorithms)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren
Rückgabe:	Iterator der hinter das letzte Element der Ergebnisfolge zeigt.

**Achtung:** Das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** die Folge, in welche die Ausgabe erfolgt, muss bereits hinreichend Platz zur Aufnahme der Elemente reserviert haben. Das Kopieren erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**.

```
//=====
// PROGRAMM: ALGORITHM_BSP_63
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<int> aFolge1;
    int i;

    aFolge1.push_back(1);
    aFolge1.push_back(2);
    aFolge1.push_back(1);
    aFolge1.push_back(3);
    aFolge1.push_back(1);
    aFolge1.push_back(4);

    vector<int> aFolge2(aFolge1.size());

    for (i=0; i<aFolge1.size(); i++)
    {
        cout << "Folge1: " << aFolge1[i] << endl;
```

```
}
cout << endl;

replace_copy (aFolge1.begin(), aFolge1.end(),
             aFolge2.begin(), 1, 1000);

for (i=0; i<aFolge2.size(); i++)
{
    cout << "Folge2: " << aFolge2[i] << endl;
}
cout << endl;
}
```

## 14.47 replace\_copy\_if

Der generische Algorithmus **replace\_copy\_if** kopiert alle Elemente einer Folge, wobei die mit einer unären Bewertungsfunktion aussortierten Elemente durch andere Elemente ersetzt werden.

Syntax:

```
write_iterator replace_copy_if (read_iterator first,
                               read_iterator last,
                               write_iterator output,
                               unäre_Bewertung func,
                               const T& newvalue);
```

Typ:	Ersetzungsalgorithmen (replacing algorithms)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren
Rückgabe:	<b>operator&lt;, operator==, operator=</b> Iterator der hinter das letzte Element der Ergebnisfolge zeigt.

**Achtung:** Das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** die Folge, in welche die Ausgabe erfolgt, muss bereits hinreichend Platz zur Aufnahme der Elemente reserviert haben. Das Kopieren erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**.

```
//=====
// PROGRAMM: ALGORITHM_BSP_64
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

bool fnUngerade (int x)
{
    return x%2 ? true : false;
}

void main (void)
{
    vector<int> aFolge1;
    int i;

    aFolge1.push_back(1);
    aFolge1.push_back(2);
    aFolge1.push_back(1);
    aFolge1.push_back(3);
    aFolge1.push_back(1);
    aFolge1.push_back(4);
```

```
vector<int> aFolge2(aFolge1.size());

for (i=0; i<aFolge1.size(); i++)
{
    cout << "Folge1: " << aFolge1[i] << endl;
}
cout << endl;

replace_copy_if (aFolge1.begin(), aFolge1.end(),
                aFolge2.begin(), fnUngerade, 1000);

for (i=0; i<aFolge2.size(); i++)
{
    cout << "Folge2: " << aFolge2[i] << endl;
}
cout << endl;
}
```

## 14.48 replace\_if

Der generische Algorithmus **replace\_if** ersetzt, unter Zuhilfenahme einer unären Bewertungsfunktion, Elemente aus einer Folge durch einen anderen Wert.

Syntax:

```
void replace_if (forward_iterator first,
                forward_iterator last,
                unäre_Bewertung func,
                const T& newvalue);
```

Typ:	Ersetzungsalgorithmen (replacing algorithms)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren <b>operator&lt;, operator==, operator=</b>
Rückgabe:	keine

**Achtung:** Das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.



```
//=====
// PROGRAMM: ALGORITHM_BSP_65
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

bool fnGerade (int x)
{
    return x%2 ? false : true;
}

void main (void)
{
    vector<int> aFolge;
    int i;

    aFolge.push_back(10);
    aFolge.push_back(2);
    aFolge.push_back(112);
    aFolge.push_back(3);
    aFolge.push_back(123);
    aFolge.push_back(4);

    for (i=0; i<aFolge.size(); i++)
    {
        cout << "Folge: " << aFolge[i] << endl;
    }
}
```

```
}
cout << endl;

replace_if (aFolge.begin(), aFolge.end(), fnGerade, 1000);

for (i=0; i<aFolge.size(); i++)
{
    cout << "Folge: " << aFolge[i] << endl;
}
}
```

## 14.49 reverse

Der generische Algorithmus **reverse** kehrt die Reihenfolge der in einer Folge enthaltenen Elemente um.

Syntax:

```
void reverse (bidirectional_iterator first,
             bidirectional_iterator last);
```

Typ:	Ersetzungsalgorithmen (replacing algorithms)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren <b>operator&lt;, operator==, operator=</b>
Rückgabe:	keine

**Achtung:** Das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.



```
//=====
// PROGRAMM: ALGORITHM_BSP_66
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<int> aFolge;
    int i;

    for (i=0; i<10; i++)
    {
        aFolge.push_back(i);
        cout << "Folge: " << aFolge[i] << endl;
    }
    cout << endl;

    reverse (aFolge.begin(), aFolge.end());

    for (i=0; i<aFolge.size(); i++)
    {
        cout << "Folge: " << aFolge[i] << endl;
    }
}
```

## 14.50 reverse\_copy

Der generische Algorithmus **reverse** kopiert die Elemente einer Ausgangsfolge in eine neue Folge und kehrt dabei die Reihenfolge der enthaltenen Elemente um.

Syntax:

```
void reverse_copy (bidirectional_iterator first,
                  bidirectional_iterator last
                  write_iterator output);
```

Typ:	Ersetzungsalgorithmen (replacing algorithms)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren <b>operator&lt;, operator==, operator=</b>
Rückgabe:	keine

**Achtung:** Das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** die Folge, in welche die Ausgabe erfolgt, muss bereits hinreichend Platz zur Aufnahme der Elemente reserviert haben. Das Kopieren erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**.

```
//=====
// PROGRAMM: ALGORITHM_BSP_67
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<int> aFolge;
    int i;

    for (i=0; i<10; i++)
    {
        aFolge.push_back(i);
        cout << "Folge: " << aFolge[i] << endl;
    }
    cout << endl;

    vector<int> aFolge2(aFolge.size());

    reverse_copy (aFolge.begin(), aFolge.end(), aFolge2.begin());

    for (i=0; i<aFolge2.size(); i++)
    {
        cout << "Folge2: " << aFolge2[i] << endl;
    }
}
```

}

## 14.51 rotate

Der generische Algorithmus **rotate** rotiert die Elemente einer Folge um einen angegebenen Punkt.

Syntax:

```
void rotate (forward_iterator first,
            forward_iterator middle,
            forward_iterator last);
```

Typ:	Rotierungsalgorithmen (rotating algorithms)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren
Rückgabe:	<b>operator&lt;, operator==, operator=</b> keine

Die Folge **first / last** wird um insgesamt **middle – first** Plätze nach rechts rotiert.

**Achtung:** Das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.



```
//=====
// PROGRAMM: ALGORITHM_BSP_68
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<int> aFolge;
    int i;

    for (i=0; i<10; i++)
    {
        aFolge.push_back(i);
        cout << "Folge: " << aFolge[i] << endl;
    }
    cout << endl;

    rotate (aFolge.begin(), aFolge.begin()+3, aFolge.end());

    for (i=0; i<aFolge.size(); i++)
    {
        cout << "Folge: " << aFolge[i] << endl;
    }
}
```

## 14.52 rotate\_copy

Der generische Algorithmus **rotate\_copy** kopiert alle Elemente einer Folge, wobei die Elemente einer Folge um einen angegebenen Punkt rotiert werden.

Syntax:

```
iterator rotate_copy (forward_iterator first,
                    forward_iterator middle,
                    forward_iterator last,
                    write_iterator output);
```

Typ:	Rotierungsalgorithmen (rotating algorithms)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren <b>operator&lt;, operator==, operator=</b>
Rückgabe:	Iterator der hinter das letzte Element der Ergebnisfolge zeigt.

Die Kopie der Folge **first / last** wird um insgesamt **middle – first** Plätze nach rechts rotiert.

**Achtung:** Das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** die Folge, in welche die Ausgabe erfolgt, muss bereits hinreichend Platz zur Aufnahme der Elemente reserviert haben. Das Kopieren erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**.

```
//=====
// PROGRAMM: ALGORITHM_BSP_69
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<int> aFolge;
    int i;

    for (i=0; i<10; i++)
    {
        aFolge.push_back(i);
        cout << "Folge: " << aFolge[i] << endl;
    }
    cout << endl;

    vector<int> aFolge2(aFolge.size());

    rotate_copy (aFolge.begin(), aFolge.begin()+3, aFolge.end(),
                aFolge2.begin());
```

```
for (i=0; i<aFolge2.size(); i++)
{
    cout << "Folge2: " << aFolge2[i] << endl;
}
}
```

## 14.53 search

Der generische Algorithmus **search** sucht innerhalb einer Folge nach einer gegebenen Teilfolge.

Syntax:

```
forward_iterator search (forward_iterator first1,
                        forward_iterator last1,
                        forward_iterator first2,
                        forward_iterator last2);
```

Syntax:

```
forward_iterator search (forward_iterator first1,
                        forward_iterator last1,
                        forward_iterator first2,
                        forward_iterator last2,
                        binäre_Bewertung binfunc);
```

Typ:	Suchalgorithmen (searching algorithms)
Zeitbedarf:	quadratisch
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren
Rückgabe:	Iterator, der auf die gesuchte Teilfolge zeigt (falls die Teilfolge gefunden wurde) oder auf <b>last1</b> (falls die Teilfolge nicht gefunden wurde).

Der Unterschied der beiden **search**-Varianten besteht darin, dass die erste Form den Vergleich der Elemente mit dem Vergleichsoperator **operator==** durchführt. Für selbstdefinierte Klassen wird die entsprechende, überladene Operatorfunktion **operator==** benutzt.

Die zweite Form hingegen verwendet statt des Vergleichsoperators die binäre Bewertungsfunktion **binfunc**. Dadurch sind auch andere Vergleichsformen realisierbar.

**Achtung:** Die Elemente **last1** und **last2** gehören nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigen sie hinter den letzten Wert der jeweiligen Folge.

```
//=====
// PROGRAMM: ALGORITHM_BSP_70
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
```

```

vector<int> aFolge1;
vector<int> aFolge2;
vector<int>::iterator aErg;
int i;

for (i=0; i<10; i++)
{
    aFolge1.push_back(i);
    cout << "Folge1: " << aFolge1[i] << endl;
}
cout << endl;

for (i=4; i<7; i++)
{
    aFolge2.push_back(i);
}

for (i=0; i<aFolge2.size(); i++)
{
    cout << "Folge2: " << aFolge2[i] << endl;
}
cout << endl;

aErg = search (aFolge1.begin(), aFolge1.end(),
              aFolge2.begin(), aFolge2.end());

if (aErg == aFolge1.end())
{
    cout << "keine Teilfolge gefunden" << endl;
}
else
{
    cout << "Teilfolge gefunden an Index " <<
          (aErg - aFolge1.begin()) << endl;
}
}

```



```

//=====
// PROGRAMM: ALGORITHM_BSP_71
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

bool fnEqual (int a, int b)
{
    bool bErg = false;
    int nDiff = a-b;

    if ((nDiff >= -1) && (nDiff <= 1))
    {
        bErg = true;
    }
    return bErg;
}

void main (void)
{

```

```
vector<int> aFolge1;
vector<int> aFolge2;
vector<int>::iterator aErg;
int i;

for (i=0; i<10; i++)
{
    aFolge1.push_back(i);
    cout << "Folge1: " << aFolge1[i] << endl;
}
cout << endl;

for (i=4; i<7; i++)
{
    aFolge2.push_back(i);
}

for (i=0; i<aFolge2.size(); i++)
{
    cout << "Folge2: " << aFolge2[i] << endl;
}
cout << endl;

aErg= search (aFolge1.begin(), aFolge1.end(),
             aFolge2.begin(), aFolge2.end(),
             fnEqual);

if (aErg == aFolge1.end())
{
    cout << "keine Teilfolge gefunden" << endl;
}
else
{
    cout << "Teilfolge gefunden an Index " <<
        (aErg - aFolge1.begin()) << endl;
}
}
```

## 14.54 set\_difference

Der generische Algorithmus **set\_difference** erzeugt eine neue Folge, welche die Differenzmenge zweier Folgen darstellt. D.h. die neue Folge enthält alle Elemente, die in nur einer der beiden Quellenfolgen enthalten sind.

Syntax:

```
write_iterator set_difference (read_iterator first1,
                             read_iterator last1,
                             read_iterator first2,
                             read_iterator last2,
                             write_iterator output);
```

Syntax:

```
write_iterator set_difference (read_iterator first1,
                             read_iterator last1,
                             read_iterator first2,
                             read_iterator last2,
                             write_iterator output,
                             Vergleichsfunktion compare);
```

Typ:	Mengenalgorithmen (set algorithms)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren
Rückgabe:	<b>operator&lt;, operator==, operator=</b> Iterator der hinter das letzte Element der Ergebnisfolge zeigt.

Der Unterschied der beiden **set\_difference**-Varianten besteht darin, dass die erste Form den Vergleichsoperator **operator==** zur Bewertung heranzieht, bzw. die entsprechende, überladene Operatorfunktion **operator==** für selbstdefinierte Klassen.

Die zweite Form hingegen benutzt statt des Vergleichsoperators die zu übergebene Vergleichsfunktion **compare**. Dadurch sind auch andere Vergleichsformen zur Differenzbildung möglich.

**Achtung:** Das Ergebnis für sich überschneidende Folgenbereiche ist nicht definiert.

**Achtung:** Die Elemente **last1** und **last2** gehören nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigen sie hinter den letzten Wert der jeweiligen Folge.

**Achtung:** die Folge, in welche die Ausgabe erfolgt, muss bereits hinreichend Platz zur Aufnahme der Elemente reserviert haben. Das Kopieren erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**.

```
//=====
// PROGRAMM: ALGORITHM_BSP_72
//=====
#include <iomanip.h>
```

```

#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

// nur bei Microsoft-Compiler
#include "stlminmax.h"

void main (void)
{
    vector<int> aFolge1;
    vector<int> aFolge2;
    vector<int>::iterator aI;
    int i;

    for (i=0; i<10; i++)
    {
        aFolge1.push_back(i);
        cout << "Folge1: " << aFolge1[i] << endl;
    }
    cout << endl;

    for (i=0; i<7; i++)
    {
        aFolge2.push_back(i);
        cout << "Folge2: " << aFolge2[i] << endl;
    }
    cout << endl;

    //-----
    // man muss damit rechnen, das die grössere Folge eventuell
    // komplett kopiert wird, wenn die Folgen keine gemeinsamen
    // Elemente besitzen
    //-----
    vector<int> aErg (max (aFolge1.size(), aFolge2.size()));

    aI = set_difference (aFolge1.begin(), aFolge1.end(),
                        aFolge2.begin(), aFolge2.end(),
                        aErg.begin());

    //-----
    // Hier wäre aErg noch zu gross
    //-----
    aErg.resize(aI-aErg.begin());
    for (i=0; i<aErg.size(); i++)
    {
        cout << "Ergebnisfolge: " << aErg[i] << endl;
    }
}

```

```

//=====
// PROGRAMM: ALGORITHM_BSP_73
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

```

```
// nur bei Microsoft-Compiler
#include "stlminmax.h"

bool fnEqual (int a, int b)
{
    bool bErg = false;
    int nDiff = a-b;

    if ((nDiff >= -1) && (nDiff <= 1))
    {
        bErg = true;
    }
    return bErg;
}

void main (void)
{
    vector<int> aFolge1;
    vector<int> aFolge2;
    vector<int>::iterator aI;
    int i;

    for (i=0; i<10; i++)
    {
        aFolge1.push_back(i);
        cout << "Folge1: " << aFolge1[i] << endl;
    }
    cout << endl;

    for (i=0; i<7; i++)
    {
        aFolge2.push_back(i);
        cout << "Folge2: " << aFolge2[i] << endl;
    }
    cout << endl;

    //-----
    // man muss damit rechnen, das die grössere Folge eventuell
    // komplett kopiert wird, wenn die Folgen keine gemeinsamen
    // Elemente besitzen
    //-----
    vector<int> aErg (max (aFolge1.size(), aFolge2.size()));

    aI = set_difference (aFolge1.begin(), aFolge1.end(),
                       aFolge2.begin(), aFolge2.end(),
                       aErg.begin(), fnEqual);

    //-----
    // Hier wäre aErg noch zu gross
    //-----
    aErg.resize(aI-aErg.begin());
    for (i=0; i<aErg.size(); i++)
    {
        cout << "Ergebnisfolge: " << aErg[i] << endl;
    }
}
```

## 14.55 set\_intersection

Der generische Algorithmus **set\_intersection** erzeugt eine neue Folge, welche die Schnittmenge zweier Folgen darstellt. D.h. die neue Folge enthält alle Elemente, die in beiden Quellenfolgen enthalten sind.

Syntax:

```
write_iterator set_intersection (read_iterator first1,
                               read_iterator last1,
                               read_iterator first2,
                               read_iterator last2,
                               write_iterator output);
```

Syntax:

```
write_iterator set_intersection
                               (read_iterator first1,
                               read_iterator last1,
                               read_iterator first2,
                               read_iterator last2,
                               write_iterator output,
                               Vergleichsfunktion compare);
```

Typ:	Mengenalgorithmen (set algorithms)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren
Rückgabe:	<b>operator&lt;, operator==, operator=</b> Iterator der hinter das letzte Element der Ergebnisfolge zeigt.

Der Unterschied der beiden **set\_intersection**-Varianten besteht darin, dass die erste Form den Vergleichsoperator **operator==** zur Bewertung heranzieht, bzw. die entsprechende, überladene Operatorfunktion **operator==** für selbstdefinierte Klassen.

Die zweite Form hingegen benutzt statt des Vergleichsoperators die zu übergebene Vergleichsfunktion **compare**. Dadurch sind auch andere Vergleichsformen zur Differenzbildung möglich.

**Achtung:** Das Ergebnis für sich überschneidende Folgenbereiche ist nicht definiert.

**Achtung:** Die Elemente **last1** und **last2** gehören nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigen sie hinter den letzten Wert der jeweiligen Folge.

**Achtung:** die Folge, in welche die Ausgabe erfolgt, muss bereits hinreichend Platz zur Aufnahme der Elemente reserviert haben. Das Kopieren erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**.

```
//=====
// PROGRAMM: ALGORITHM_BSP_74
//=====
```

```

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

// nur bei Microsoft-Compiler
#include "stlminmax.h"

void main (void)
{
    vector<int> aFolge1;
    vector<int> aFolge2;
    vector<int>::iterator aI;
    int i;

    for (i=0; i<10; i++)
    {
        aFolge1.push_back(i);
        cout << "Folge1: " << aFolge1[i] << endl;
    }
    cout << endl;

    for (i=0; i<4; i++)
    {
        aFolge2.push_back(i);
        cout << "Folge2: " << aFolge2[i] << endl;
    }
    cout << endl;

    //-----
    // man muss damit rechnen, das die grössere Folge eventuell
    // komplett kopiert wird, wenn die Folgen keine gemeinsamen
    // Elemente besitzen
    //-----
    vector<int> aErg (max (aFolge1.size(), aFolge2.size()));

    aI = set_intersection (aFolge1.begin(), aFolge1.end(),
                          aFolge2.begin(), aFolge2.end(),
                          aErg.begin());

    //-----
    // Hier wäre aErg noch zu gross
    //-----
    aErg.resize(aI-aErg.begin());
    for (i=0; i<aErg.size(); i++)
    {
        cout << "Ergebnisfolge: " << aErg[i] << endl;
    }
}

```

```

//=====
// PROGRAMM: ALGORITHM_BSP_75
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

```

```
// nur bei Microsoft-Compiler
#include "stlminmax.h"

bool fnEqual (int a, int b)
{
    bool bErg = false;
    int nDiff = a-b;

    if ((nDiff >= -1) && (nDiff <= 1))
    {
        bErg = true;
    }
    return bErg;
}

void main (void)
{
    vector<int> aFolge1;
    vector<int> aFolge2;
    vector<int>::iterator aI;
    int i;

    for (i=0; i<10; i++)
    {
        aFolge1.push_back(i);
        cout << "Folge1: " << aFolge1[i] << endl;
    }
    cout << endl;

    for (i=0; i<4; i++)
    {
        aFolge2.push_back(i);
        cout << "Folge2: " << aFolge2[i] << endl;
    }
    cout << endl;

    //-----
    // man muss damit rechnen, das die grössere Folge eventuell
    // komplett kopiert wird, wenn die Folgen keine gemeinsamen
    // Elemente besitzen
    //-----
    vector<int> aErg (max (aFolge1.size(), aFolge2.size()));

    aI = set_intersection (aFolge1.begin(), aFolge1.end(),
                          aFolge2.begin(), aFolge2.end(),
                          aErg.begin(), fnEqual);

    //-----
    // Hier wäre aErg noch zu gross
    //-----
    aErg.resize(aI-aErg.begin());
    for (i=0; i<aErg.size(); i++)
    {
        cout << "Ergebnisfolge: " << aErg[i] << endl;
    }
}
```

## 14.56 set\_symmetric\_difference

Der generische Algorithmus **set\_symmetric\_difference** erzeugt eine neue, sortierte Folge, welche die Differenzmenge zweier Folgen darstellt. D.h. die neue Folge enthält alle Elemente, die in nur einer der beiden Quellenfolgen enthalten sind.

Syntax:

```
write_iterator set_symmetric_difference
                (read_iterator first1,
                 read_iterator last1,
                 read_iterator first2,
                 read_iterator last2,
                 write_iterator output);
```

Syntax:

```
write_iterator set_symmetric_difference
                (read_iterator first1,
                 read_iterator last1,
                 read_iterator first2,
                 read_iterator last2,
                 write_iterator output,
                 Vergleichsfunktion compare);
```

Typ:	Mengenalgorithmen (set algorithms)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren
Rückgabe:	<b>operator&lt;</b> , <b>operator==</b> , <b>operator=</b> Iterator der hinter das letzte Element der Ergebnisfolge zeigt.

Der Unterschied der beiden **set\_symmetric\_difference**-Varianten besteht darin, dass die erste Form eine Sortierung mit dem Kleiner-als-Operator **operator<** voraussetzt, bzw. eine Sortierung mit der entsprechenden, überladenen Operatorfunktion **operator<** für selbstdefinierte Klassen.

Die zweite Form hingegen benutzt statt des Kleiner-als-Operators die zu übergebene Vergleichsfunktion **compare**. Dadurch sind auch andere Vergleichsformen zur sortierten Differenzbildung möglich.

**Achtung:** Das Ergebnis für sich überschneidende Folgenbereiche ist nicht definiert.

**Achtung:** Die Elemente **last1** und **last2** gehören nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigen sie hinter den letzten Wert der jeweiligen Folge.

**Achtung:** die Folge, in welche die Ausgabe erfolgt, muss bereits hinreichend Platz zur Aufnahme der Elemente reserviert haben. Das Kopieren erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**.



```
//=====
// PROGRAMM: ALGORITHM_BSP_76
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

// nur bei Microsoft-Compiler
#include "stlminmax.h"

void main (void)
{
    vector<int> aFolge1;
    vector<int> aFolge2;
    vector<int>::iterator aI;
    int i;

    for (i=0; i<10; i++)
    {
        aFolge1.push_back(i);
        cout << "Folge1: " << aFolge1[i] << endl;
    }
    cout << endl;

    for (i=0; i<7; i++)
    {
        aFolge2.push_back(i);
        cout << "Folge2: " << aFolge2[i] << endl;
    }
    cout << endl;

    //-----
    // man muss damit rechnen, das die grössere Folge eventuell
    // komplett kopiert wird, wenn die Folgen keine gemeinsamen
    // Elemente besitzen
    //-----
    vector<int> aErg (max (aFolge1.size(), aFolge2.size()));

    aI = set_symmetric_difference (aFolge1.begin(), aFolge1.end(),
                                   aFolge2.begin(), aFolge2.end(),
                                   aErg.begin());

    //-----
    // Hier wäre aErg noch zu gross
    //-----
    aErg.resize(aI-aErg.begin());
    for (i=0; i<aErg.size(); i++)
    {
        cout << "Ergebnisfolge: " << aErg[i] << endl;
    }
}
```

## 14.57 set\_union

Der generische Algorithmus **set\_union** erzeugt eine neue sortierte Folge, welche die Vereinigungsmenge zweier Folgen darstellt. D.h. die neue Folge enthält alle Elemente, die in einer der beiden Quellenfolgen enthalten sind.

Syntax:

```
write_iterator set_union (read_iterator first1,
                          read_iterator last1,
                          read_iterator first2,
                          read_iterator last2,
                          write_iterator output);
```

Syntax:

```
write_iterator set_union (read_iterator first1,
                          read_iterator last1,
                          read_iterator first2,
                          read_iterator last2,
                          write_iterator output,
                          Vergleichsfunktion compare);
```

Typ:	Mengenalgorithmen (set algorithms)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren
Rückgabe:	<b>operator&lt;, operator==, operator=</b> Iterator der hinter das letzte Element der Ergebnisfolge zeigt.

Ist ein Element in beiden Folgen enthalten, so wird das Element aus der ersten Quellmenge in die Zielmenge eingestellt.

Der Unterschied der beiden **set\_union**-Varianten besteht darin, dass die erste Form eine Sortierung mit dem Kleiner-als-Operator **operator<** voraussetzt, bzw. eine Sortierung mit der entsprechenden, überladenen Operatorfunktion **operator<** für selbstdefinierte Klassen.

Die zweite Form hingegen benutzt statt des Kleiner-als-Operators die zu übergebene Vergleichsfunktion **compare**. Dadurch sind auch andere Vergleichsformen zur sortierten Differenzbildung möglich.

**Achtung:** Das Ergebnis für sich überschneidende Folgenbereiche ist nicht definiert.

**Achtung:** Die Elemente **last1** und **last2** gehören nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigen sie hinter den letzten Wert der jeweiligen Folge.

**Achtung:** die Folge, in welche die Ausgabe erfolgt, muss bereits hinreichend Platz zur Aufnahme der Elemente reserviert haben. Das Kopieren erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**.



```
//=====
// PROGRAMM: ALGORITHM_BSP_77
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<int> aFolge1;
    vector<int> aFolge2;
    vector<int>::iterator aI;
    int i;

    for (i=0; i<10; i++)
    {
        aFolge1.push_back(i);
        cout << "Folge1: " << aFolge1[i] << endl;
    }
    cout << endl;

    for (i=0; i<12; i++)
    {
        aFolge2.push_back(i);
        cout << "Folge2: " << aFolge2[i] << endl;
    }
    cout << endl;

    //-----
    // man muss damit rechnen, das die Ergebnismenge eventuell
    // der Summe der Quellmengen entspricht
    //-----
    vector<int> aErg (aFolge1.size() + aFolge2.size());

    aI = set_union (aFolge1.begin(), aFolge1.end(),
                   aFolge2.begin(), aFolge2.end(),
                   aErg.begin());

    //-----
    // Hier wäre aErg eventuell noch zu gross
    //-----
    aErg.resize(aI-aErg.begin());
    for (i=0; i<aErg.size(); i++)
    {
        cout << "Ergebnisfolge: " << aErg[i] << endl;
    }
}
```

## 14.58 sort

Der generische Algorithmus **sort** sortiert eine Folge.

Syntax:

```
void sort (randomaccess_iterator first,
          randomaccess_iterator last);

void sort (randomaccess_iterator first,
          randomaccess_iterator last,
          Vergleichsfunktion compare);
```

Typ:	Sortieralgorithmen (sorting algorithms)
Zeitbedarf:	$O(N \log N)$
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Random-Access-Iteratoren
Rückgabe:	<b>operator&lt;</b> , <b>operator==</b> , <b>operator=</b> keine

Iteratoren mit wahlfreiem Zugriff sind nur für die Container-Klassen **vector** und **deque** definiert.

Der Unterschied der beiden **sort**-Varianten besteht darin, dass die erste Form den Vergleich der Elemente mit dem Kleiner-als-Operator **operator<** durchführt. Für selbstdefinierte Klassen wird die entsprechende, überladene Operatorfunktion **operator<** benutzt.

Die zweite Form hingegen verwendet statt des Kleiner-als-Operators die Vergleichsfunktion **compare**. Dadurch sind auch andere Sortierungen realisierbar.

**Achtung:** das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.

```
//=====
// PROGRAMM: ALGORITHM_BSP_78
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>
#include <conio.h>

using namespace std;

void main (void)
{
    vector<int> aFolge;
    int i = 1;
```

```
aFolge.push_back(9);
aFolge.push_back(3);
aFolge.push_back(8);

aFolge.push_back(5);
aFolge.push_back(5);
aFolge.push_back(4);

aFolge.push_back(2);
aFolge.push_back(7);
aFolge.push_back(5);

for (i=0; i<9; i++)
{
    cout << aFolge[i] << endl;
}
cout << endl;

sort (aFolge.begin(), aFolge.end());

for (i=0; i<9; i++)
{
    cout << aFolge[i] << endl;
}
cout << endl;
}
```

## 14.59 sort\_heap

Der generische Algorithmus **sort\_heap** sortiert einen **heap**, wo bei die **heap**-Eigenschaft verloren geht.

Syntax:

```
void sort_heap (randomaccess_iterator first,
               randomaccess_iterator last);

void sort_heap (randomaccess_iterator first,
               randomaccess_iterator last,
               Vergleichsfunktion compare);
```

Typ:	Heapalgorithmen (heap algorithms)
Zeitbedarf:	$O \log(n)$
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Random-Access-Iteratoren
Rückgabe:	<b>operator&lt;, operator==, operator=</b> Keine

Iteratoren mit wahlfreiem Zugriff sind nur für die Container-Klassen **vector** und **deque** definiert, so dass auch nur aus diesen Containern heraus ein **heap** erzeugt werden kann. Der **sort\_heap** Algorithmus stellt die Sortierung innerhalb der **heap**-Folge um.

Der Unterschied der beiden **sort\_heap**-Varianten besteht darin, dass die erste Form für die Sortierung den Kleiner-als-Operator **operator<** verwendet. Für selbstdefinierte Klassen wird die entsprechende, überladene Operatorfunktion **operator<** benutzt.

Die zweite Form hingegen ruft statt des Kleiner-als-Operators die zu übergebene Vergleichsfunktion **compare** auf. Dadurch sind auch andere Sortierungen auf dem **heap** realisierbar.

**Achtung:** nach Aufruf des **sort\_heap**-Algorithmus ist die **heap**-Eigenschaft verloren gegangen!

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.



```
//=====
// PROGRAMM: ALGORITHM_BSP_79
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
```

```
vector<int> aFolge;
int i;

aFolge.push_back(123);
aFolge.push_back(2);
aFolge.push_back(345);
aFolge.push_back(23);
aFolge.push_back(4);
aFolge.push_back(-4);

for (i=0; i<aFolge.size(); i++)
{
    cout << "Folge: " << aFolge[i] << endl;
}
cout << endl;

make_heap (aFolge.begin(), aFolge.end());
aFolge.push_back(8888);
push_heap (aFolge.begin(), aFolge.end());
sort_heap (aFolge.begin(), aFolge.end());

for (i=0; i<aFolge.size(); i++)
{
    cout << "Folge: " << aFolge[i] << endl;
}
}
```

## 14.60 stable\_partition

Dieser generische Algorithmus teilt – unter Verwendung einer Bewertungsfunktion – die Elemente einer Folge in zwei Gruppen auf. Der Algorithmus ist im Gegensatz zu **partition** stabil, d.h. die Elemente behalten ihre relative Reihenfolge zueinander bei.

Syntax:

```
bidirectional_iterator partition
    (bidirectional_iterator first,
     bidirectional_iterator last,
     unäre_Bewertung func);
```

Typ:	Aufteilungsalgorithmen (partitioning algorithms)
Zeitbedarf:	$O(N \log N)$ wenn nicht genügend freier Speicherplatz vorhanden ist, sonst $O(N)$
Platzbedarf:	Konstant wenn nicht genügend freier Speicherplatz vorhanden ist, sonst $N$
Voraussetzung:	Container unterstützt Iteratoren
Rückgabe:	Iterator, der auf das erste Element der zweiten Folge zeigt

Die Elemente des Intervalls werden von **partition** dergestalt sortiert, dass alle Elemente, für welche die unäre Bewertungsfunktion **true** zurückgibt vor denen stehen, für die **false** zurückgegeben wird.

**Achtung:** Das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.

```
//=====
// PROGRAMM: ALGORITHM_BSP_80
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

bool greater500 (double x)
{
    return x > 500.0 ? true : false;
}

void main (void)
{
    vector<double> aVect;
    int i;

    aVect.push_back (134.0);
    aVect.push_back (235.0);
```

```
aVect.push_back (983.0);
aVect.push_back (774.0);
aVect.push_back (345.0);
aVect.push_back (677.0);
aVect.push_back (676.0);
aVect.push_back (338.0);

stable_partition (aVect.begin(), aVect.end(), greater500);

for (i=0; i<aVect.size(); i++)
{
    cout << aVect[i] << endl;
}
cout << endl;

for (i=0; i<aVect.size(); i++)
{
    cout << aVect[i] << endl;
}
cout << endl;
}
```

## 14.61 stable\_sort

Der generische Algorithmus **stable\_sort** sortiert eine Folge. Der Algorithmus ist im Gegensatz zu **sort** stabil, d.h. Elemente mit gleichem Wert behalten ihre relative Reihenfolge zueinander bei.

Syntax:

```
void stable_sort (randomaccess_iterator first,
                 randomaccess_iterator last);
```

```
void stable_sort (randomaccess_iterator first,
                 randomaccess_iterator last,
                 Vergleichsfunktion compare);
```

Typ:	Sortieralgorithmen (sorting algorithms)
Zeitbedarf:	$O(N \log N)^2$ wenn nicht genügend freier Speicherplatz vorhanden ist, sonst $O(N \log N)$
Platzbedarf:	Konstant wenn nicht genügend freier Speicherplatz vorhanden ist, sonst $N$
Voraussetzung:	Container unterstützt Random-Access-Iteratoren <b>operator&lt;</b> , <b>operator==</b> , <b>operator=</b>
Rückgabe:	keine

Iteratoren mit wahlfreiem Zugriff sind nur für die Container-Klassen **vector** und **deque** definiert.

Der Unterschied der beiden **stable\_sort**-Varianten besteht darin, dass die erste Form den Vergleich der Elemente mit dem Kleiner-als-Operator **operator<** durchführt. Für selbstdefinierte Klassen wird die entsprechende, überladene Operatorfunktion **operator<** benutzt.

Die zweite Form hingegen verwendet statt des Kleiner-als-Operators die Vergleichsfunktion **compare**. Dadurch sind auch andere Sortierungen realisierbar.

**Achtung:** das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.

```
//=====
// PROGRAMM: ALGORITHM_BSP_81
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>
#include <conio.h>

using namespace std;
```

```
void main (void)
{
    vector<int> aFolge;
    int i = 1;

    aFolge.push_back(9);
    aFolge.push_back(3);
    aFolge.push_back(8);

    aFolge.push_back(5);
    aFolge.push_back(5);
    aFolge.push_back(4);

    aFolge.push_back(2);
    aFolge.push_back(7);
    aFolge.push_back(5);

    for (i=0; i<9; i++)
    {
        cout << aFolge[i] << endl;
    }
    cout << endl;

    stable_sort (aFolge.begin(), aFolge.end());

    for (i=0; i<9; i++)
    {
        cout << aFolge[i] << endl;
    }
    cout << endl;
}
```

## 14.62 swap

Der generische Algorithmus **swap** tauscht die Inhalte zweier Objekte miteinander aus.

Syntax:

```
void swap (T& one,
          T& two);
```

Typ:	Tauschalgorithmus (swapping algorithm)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	<b>operator=</b>
Rückgabe:	keine

Der Algorithmus verwendet den Zuweisungs-Operator **operator=**, um die Werte, auf welche die Iteratoren verweisen, auszutauschen. Dies ist insbesondere für selbstdefinierte Klassen relevant. Ist hier der **operator=** so überladen, dass nicht alle Zustände des Objektes zugewiesen werden, so gehen diese fehlenden Eigenschaft auch bei Zuweisung über **swap** verloren.

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.



```
//=====
// PROGRAMM: ALGORITHM_BSP_82
//=====

#include <iomanip.h>
#include <iostream.h>
#include <algorithm>

using namespace std;

void main (void)
{
    int a = 10, b = 12;

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

    swap (a, b);

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
}
```

## 14.63 swap\_ranges

Der generische Algorithmus **swap** tauscht die Inhalte zweier paralleler (d.h. gleich langer) Bereiche miteinander aus.

Syntax:

```
void swap_ranges (forward_iterator first1,
                 forward_iterator last1,
                 forward_iterator first2);
```

Typ:	Tauschalgorithmus (swapping algorithm)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	<b>operator=</b>
Rückgabe:	keine

Der Algorithmus tauscht jeweils die Elemente aus, auf welche die Iteratoren zeigen, beginnend mit den elementen, auf die durch **first1** und **first2** verwiesen wird.

Der Algorithmus verwendet den Zuweisungs-Operator **operator=**, um die Werte, auf welche die Iteratoren verweisen, auszutauschen. Dies ist insbesondere für selbstdefinierte Klassen relevant. Ist hier der **operator=** so überladen, dass nicht alle Zustände des Objektes zugewiesen werden, so gehen diese fehlenden Eigenschaft auch bei Zuweisung über **swap\_ranges** verloren.

**Achtung:** das Element **last1** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.

```
//=====
// PROGRAMM: ALGORITHM_BSP_83
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<double> aVect1;
    vector<double> aVect2;
    int i;

    aVect1.push_back (134.0);
    aVect1.push_back (235.0);
    aVect1.push_back (983.0);
    aVect1.push_back (774.0);
    aVect1.push_back (345.0);
    aVect1.push_back (677.0);
```



```
aVect2.push_back (1.0);
aVect2.push_back (2.0);
aVect2.push_back (3.0);
aVect2.push_back (4.0);
aVect2.push_back (5.0);
aVect2.push_back (6.0);

for (i=0; i<aVect1.size(); i++)
{
    cout << "aVect1: " << aVect1[i] << endl;
}
cout << endl;

for (i=0; i<aVect2.size(); i++)
{
    cout << "aVect2: " << aVect2[i] << endl;
}
cout << endl;

swap_ranges (aVect1.begin(), aVect1.end(), aVect2.begin());

for (i=0; i<aVect1.size(); i++)
{
    cout << "aVect1: " << aVect1[i] << endl;
}
cout << endl;

for (i=0; i<aVect2.size(); i++)
{
    cout << "aVect2: " << aVect2[i] << endl;
}
cout << endl;
}
```

## 14.64 transform

Der generische Algorithmus **transform** wandelt eine Folge – mittels einer übergebenen Transformationsfunktion – um, indem die Transformationsfunktion für jedes Element aufgerufen wird.

Syntax:

```
write_iterator transform (read_iterator first1,
                        read_iterator last1,
                        write_iterator output,
                        unäre_Operation func);
```

Syntax:

```
write_iterator transform (read_iterator first1,
                        read_iterator last1,
                        read_iterator first2,
                        write_iterator output,
                        binäre_Operation func);
```

Typ:	Transformierungsalgorithmen (transforming algorithms)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren
	<b>operator=</b>
Rückgabe:	Iterator der hinter das letzte Element der Ergebnisfolge zeigt.

Der Algorithmus **transform** führt die Funktion **func** auf jedem Element zwischen **first1** und **last1** aus und speichert das Ergebnis in einer anderen Folge ab.

Der Unterschied der beiden **transform** -Varianten besteht darin, dass die erste Form eine unäre Funktion verwendet (z.B. Umkehrung des Vorzeichens).

Die zweite Form hingegen benutzt statt dessen eine binäre Funktion (z.B. Addition des Wertes der zweiten Quellfolge).

**Achtung:** das Element **last1** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** die Folge, die für die zweite Form des **transform**-Algorithmus herangezogen wird, muss die gleiche Länge besitzen wie die erste Quellfolge.

**Achtung:** die Folge, in welche die Ausgabe erfolgt, muss bereits hinreichend Platz zur Aufnahme der Elemente reserviert haben. Das Kopieren erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**.

```
//=====
// PROGRAMM: ALGORITHM_BSP_84
//=====

#include <iomanip.h>
#include <iostream.h>
```



```

#include <vector>
#include <algorithm>

using namespace std;

int fnChangeVZ (int x)
{
    return -x;
}

void main (void)
{
    vector<int> aVector;
    int i;

    aVector.push_back (1);
    aVector.push_back (2);
    aVector.push_back (3);

    vector<int> aErg(aVector.size());

    transform (aVector.begin(), aVector.end(),aErg.begin(), fnChangeVZ);

    for (i=0; i<aVector.size(); i++)
    {
        cout << "aVector: " << aVector[i] << endl;
    }
    cout << endl;

    for (i=0; i<aErg.size(); i++)
    {
        cout << "aErg: " << aErg[i] << endl;
    }
}

```

```

//=====
// PROGRAMM: ALGORITHM_BSP_85
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

int fnSum (int x, int y)
{
    return x + y;
}

void main (void)
{
    vector<int> aVector1;
    vector<int> aVector2;
    int i;

    aVector1.push_back (1);
    aVector1.push_back (2);
    aVector1.push_back (3);

    aVector2.push_back (10);

```

```
aVector2.push_back (20);
aVector2.push_back (30);

vector<int> aErg(aVector1.size());

transform (aVector1.begin(), aVector1.end(), aVector2.begin(),
          aErg.begin(), fnSum);

for (i=0; i<aVector1.size(); i++)
{
    cout << "aVector1: " << aVector1[i] << endl;
}
cout << endl;

for (i=0; i<aVector2.size(); i++)
{
    cout << "aVector2: " << aVector2[i] << endl;
}
cout << endl;

for (i=0; i<aErg.size(); i++)
{
    cout << "aErg: " << aErg[i] << endl;
}
}
```

## 14.65 unique

Der generische Algorithmus **unique** entfernt alle doppelten Elemente einer sortierten Folge, so dass von jedem Element am Ende genau ein Exemplar übrig bleibt.

Syntax:

```
forward_iterator unique (forward_iterator first,
                        forward_iterator last);

forward_iterator unique (forward_iterator first,
                        forward_iterator last,
                        binäre_Bewertung func);
```

Typ:	Filteralgorithmen (filtering algorithms)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren Folge muss sortiert sein <b>operator&lt;, operator==, operator=</b>
Rückgabe:	Iterator der hinter das letzte Element der reduzierten Folge zeigt, also auf <b>last – n</b> wobei <b>n</b> die Anzahl der gelöschten Einträge ist.

Der Unterschied der beiden **unique**-Varianten besteht darin, dass die erste Form den Vergleichsoperator **operator==** verwendet, um die Elemente der beiden Folgen zu vergleichen. Für selbstdefinierte Klassen wird die entsprechende, überladene Operatorfunktion **operator==** verwendet.

Die zweite Form hingegen benutzt statt des Vergleichsoperators die zu übergebene binäre Bewertungsfunktion **func**. Dadurch sind auch andere Vergleichsformen möglich.

**Achtung:** Das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** Der Algorithmus **unique** ändert die Größe der verarbeiteten Folge nicht, so dass die Elemente zwischen **last – n** und **last** als undefiniert zu betrachten sind.

**Achtung:** Die Zuweisung erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**. Der **operator=** muss daher korrekt überladen sein.

```
//=====
// PROGRAMM: ALGORITHM_BSP_86
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;
```

```

void main (void)
{
    vector<int> aFolge (10);
    vector<int>::iterator aI;
    vector<int>::iterator aErg;

    for (int i=0; i<10; i++)
    {
        aFolge[i] = i;
    }
    aFolge.push_back(7);
    aFolge.push_back(5);
    aFolge.push_back(3);

    sort (aFolge.begin(), aFolge.end());

    for (i=0; i<aFolge.size(); i++)
    {
        cout << "aFolge: " << aFolge[i] << endl;
    }
    cout << endl;

    aErg = unique (aFolge.begin(), aFolge.end());

    for (aI=aFolge.begin(); aI<aErg; aI++)
    {
        cout << "aI: " << (*aI) << endl;
    }
    cout << endl;

    for (i=0; i<aFolge.size(); i++)
    {
        cout << "aFolge: " << aFolge[i] << endl;
    }
}

```

```

//=====
// PROGRAMM: ALGORITHM_BSP_87
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

bool fnNear (int nEins, int nZwei)
{
    int nDiff = nEins - nZwei;
    return ((nDiff >= -1) && (nDiff <= 1));
}

void main (void)
{
    vector<int> aFolge (5);
    vector<int>::iterator aI;
    vector<int>::iterator aErg;

    for (int i=0; i<5; i++)
    {
        aFolge[i] = i*2;
    }
}

```



```
}
aFolge.push_back(3);
aFolge.push_back(4);
aFolge.push_back(5);

sort (aFolge.begin(), aFolge.end());

for (i=0; i<aFolge.size(); i++)
{
    cout << "aFolge: " << aFolge[i] << endl;
}
cout << endl;

aErg = unique (aFolge.begin(), aFolge.end(), fnNear);

for (aI=aFolge.begin(); aI<aErg; aI++)
{
    cout << "aI: " << (*aI) << endl;
}
cout << endl;

for (i=0; i<aFolge.size(); i++)
{
    cout << "aFolge: " << aFolge[i] << endl;
}
}
```

## 14.66 unique\_copy

Der generische Algorithmus **unique** kopiert eine sortierte Folge und entfernt dabei alle doppelten Elemente, so dass in der Zielfolge von jedem Element am Ende genau ein Exemplar übrig bleibt.

Syntax:

```
write_iterator unique_copy (read_iterator first,
                           read_iterator last,
                           iterator output);

write_iterator unique_copy (read_iterator first,
                           read_iterator last,
                           write_iterator output,
                           binäre_Bewertung func);
```

Typ:	Filteralgorithmen (filtering algorithms)
Zeitbedarf:	linear
Platzbedarf:	konstant
Voraussetzung:	Container unterstützt Iteratoren Folge muss sortiert sein <b>operator&lt;, operator==, operator=</b>
Rückgabe:	Iterator der hinter das letzte Element der reduzierten Folge zeigt, also auf <b>last - n</b> wobei <b>n</b> die Anzahl der gelöschten Einträge ist.

Der Unterschied der beiden **unique**-Varianten besteht darin, dass die erste Form den Vergleichsoperator **operator==** verwendet, um die Elemente der beiden Folgen zu vergleichen. Für selbstdefinierte Klassen wird die entsprechende, überladene Operatorfunktion **operator==** verwendet.

Die zweite Form hingegen benutzt statt des Vergleichsoperators die zu übergebene binären Bewertungsfunktion **func**. Dadurch sind auch andere Vergleichsformen möglich.

**Achtung:** Das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

**Achtung:** die Folge, in welche die Ausgabe erfolgt, muss bereits hinreichend Platz zur Aufnahme der Elemente reserviert haben. Das Kopieren erfolgt über den Zuweisungsoperator **operator=**, nicht über eine Methode wie z.B. **push\_back**.

```
//=====
// PROGRAMM: ALGORITHM_BSP_88
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;
```

```

void main (void)
{
    vector<int> aFolge (10);
    vector<int>::iterator aErg;
    int i;

    for (i=0; i<10; i++)
    {
        aFolge[i] = i;
    }
    aFolge.push_back(7);
    aFolge.push_back(5);
    aFolge.push_back(3);

    sort (aFolge.begin(), aFolge.end());

    for (i=0; i<aFolge.size(); i++)
    {
        cout << "aFolge: " << aFolge[i] << endl;
    }
    cout << endl;

    vector<int> aZiel (aFolge.size());
    aErg = unique_copy (aFolge.begin(), aFolge.end(), aZiel.begin());
    aZiel.resize(aErg - aZiel.begin());

    for (i=0; i<aZiel.size(); i++)
    {
        cout << "aZiel: " << aZiel[i] << endl;
    }
    cout << endl;
}

```



```

//=====
// PROGRAMM: ALGORITHM_BSP_89
//=====

#include <iomanip>
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

bool fnNear (int nEins, int nZwei)
{
    int nDiff = nEins - nZwei;
    return ((nDiff >= -1) && (nDiff <= 1));
}

void main (void)
{
    vector<int> aFolge (5);
    vector<int>::iterator aErg;

    for (int i=0; i<5; i++)
    {
        aFolge[i] = i*2;
    }
    aFolge.push_back(3);
    aFolge.push_back(4);
    aFolge.push_back(5);
}

```

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

for (i=0; i<aFolge.size(); i++)
{
    cout << "aFolge: " << aFolge[i] << endl;
}
cout << endl;

vector<int> aZiel (aFolge.size());
aErg = unique_copy (aFolge.begin(), aFolge.end(), aZiel.begin(),
                    fnNear);
aZiel.resize(aErg - aZiel.begin());

for (i=0; i<aZiel.size(); i++)
{
    cout << "aZiel: " << aZiel[i] << endl;
}
cout << endl;
}
```

## 14.67 upper\_bound

Der generische Algorithmus **upper\_bound** ermittelt die obere Grenze eines Bereiches. Die häufigste Anwendung innerhalb der STL ist die Bestimmung der Position, an denen ein Element sortiert eingefügt werden kann.

Syntax:

```
forward_iterator upper_bound (forward_iterator first,
                             forward_iterator last,
                             const T& value);

forward_iterator upper_bound (forward_iterator first,
                             forward_iterator last,
                             const T& value,
                             Vergleichsfunktion compare);
```

Typ:	Begrenzungsalgorithmus (bounding algorithm)
Zeitbedarf:	$O \log (n)$ bei Iteratoren mit wahlfreiem Zugriff sonst $O(n)$
Platzbedarf:	Konstant
Voraussetzung:	Container unterstützt Iteratoren Folge muss sortiert sein <b>operator&lt;, operator==</b>
Rückgabe:	Iterator, welcher auf die unterste Grenze verweist, an denen ein bestimmtes Element eingefügt werden kann.

Der Unterschied der beiden **upper\_bound**-Varianten besteht darin, dass die erste Form die Sortierung mit dem Kleiner-als-Operator **operator<** voraussetzt. Für selbstdefinierte Klassen wird die entsprechende, überladene Operatorfunktion **operator<** benutzt.

Die zweite Form hingegen setzt statt des Kleiner-als-Operators die Sortierung mit der Vergleichsfunktion **compare** voraus. Dadurch sind auch andere Sortierungen realisierbar.

**Achtung:** das Element **last** gehört nicht mehr zum Vergleichsbereich, wie bei vielen STL-Methoden zeigt es hinter den letzten Wert der Folge.

```
//=====
// PROGRAMM: ALGORITHM_BSP_90
//=====

#include <iomanip.h>
#include <iostream.h>
#include <vector>
#include <algorithm>

using namespace std;

void main (void)
{
    vector<int> aFolge;
    vector<int>::iterator aLower;

    aFolge.push_back(0);
```

```
aFolge.push_back(0);
aFolge.push_back(1);
aFolge.push_back(1);
aFolge.push_back(2);
aFolge.push_back(2);

aLower = upper_bound (aFolge.begin(), aFolge.end(), 1);
cout << "Eine 1 kann eingefügt werden an Index: "
      << aLower - aFolge.begin() << endl;
}
```