1 C++ Standard-Library

Die C++ Standard-Library stellt einen Werkzeugkasten für die Programmiersprache C++ zur Verfügung. Die Library beruht auf Templates, mit deren Hilfe eine generische Programmierung ermöglicht wird. In der generischen Programmierung werden Funktionen und Klassen so allgemein gehalten, dass sie für verschiedene Datentypen genutzt werden können. Templates stellen nur gewisse Anforderungen an den Datentyp. Ein Template, welches zum Beispiel Werte aufsummiert, erwartet als Argument einen numerischen Datentyp wie zum Beispiel einen Integer-Wert.
Durch die Standardisierung der Funktionen und Klassen in der Library ist das ausführbare Programm selber leichter portier- und wartbar.

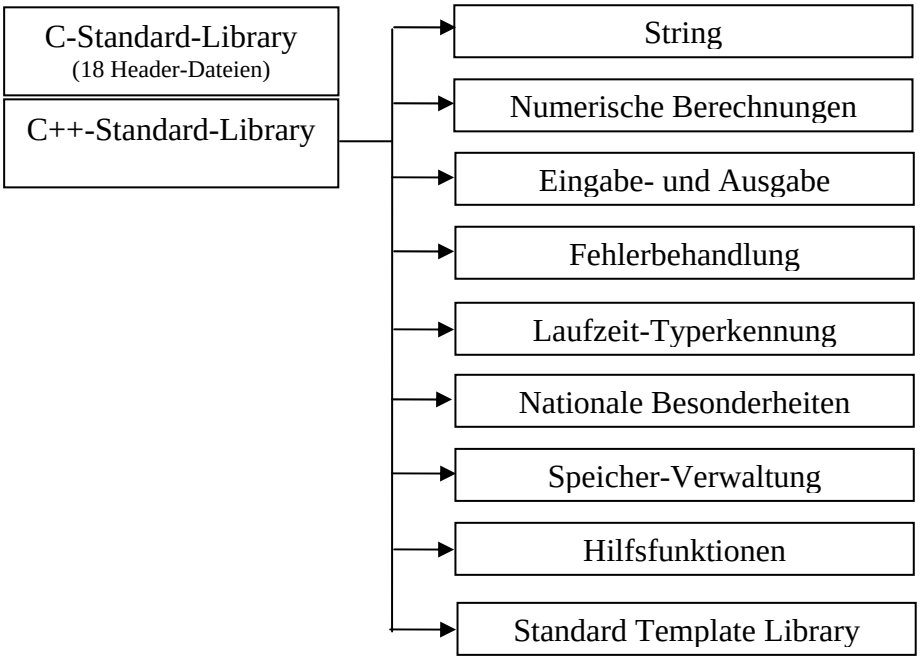


Abbildung 1 Bestandteile der C++-Standard-Library

1.1 Standard-Headerdateien

C-Headerdateien

C-Headerdatei	*Neue Form*	*Erläuterung*
<assert.h>	<cassert>	Ist der Ausdruck nicht wahr, bricht das Programm mit einer Fehlermeldung ab.
<ctype.h>	<cctype>	Klassifizierung von Zeichen.
<errno.h>	<cerrno>	Error-Fehlernummern für Bibliotheksfunktionen.
<float.h>	<cfloat>	Minimaler und maximaler Wert von Fließkommazahl-Typen.
<iso646.h>	<ciso646>	Operatormakros.
<limits.h>	<climits>	Minimaler und maximaler Wert von Ganzzahl-Typen.
<locale.h>	<clocale>	Kulturelle Besonderheiten abbilden.
<math.h>	<cmath>	Mathematische Funktionen.
<setjmp.h>	<csetjmp>	Ausführung von nicht-lokalen goto-Anweisungen.
<signal.h>	<csignal>	Legt eine Aktion fest, die bei einem bestimmten Signal ausgelöst wird. Fehlerroutinen.
<stdarg.h>	<cstdarg>	Variable Argumentlisten für Funktionen mit einer variablen Anzahl von Parametern.
<stddef.h>	<cstddef>	Definition von einfachen Typen und Makros wie zum Beispiel den NULL-Zeiger.
<stdio.h>	<cstdio>	Ein- und Ausgabe auf die Konsole oder in Dateien.
<stdlib.h>	<cstdlib>	Allgemeines wie Konstanten für den Programmabbruch.
<string.h>	<cstring>	C-Funktionen für nullterminierte Zeichenfelder.
<time.h>	<ctime>	Zeit- und Datumsfunktionen.
<wchar.h>	<cwchar>	Umwandlung von Strings zu Zahlen für den Unicode-Zeichensatz.
<wctype.h>	<cwctype>	Zeichenuntersuchung für den Unicode-Zeichensatz.

Für streng C-kompatible Programme werden die Bezeichnungen aus der ersten Spalte genutzt. Diese Headerdateien nutzen den globalen Namensraum im Gegensatz zu den Headerdateien in der zweiten Spalte. Wenn ein reines C++-Programm benötigt wird, sollten die Headerdateien in der zweiten Spalte genutzt werden. Diese Headerdateien nutzen den Namensraum std.

String-Klasse

C++-Headerdatei	*Erläuterung*
<string>	Erstellen von Strings. Manipulation von Strings.

Numerische Berechnungen

C++-Headerdatei	*Erläuterung*
<complex>	Komplexe Zahlen
<valarray>	Gleitkommaberechnungen mit großen Datenmengen, die auf eindimensionalen Zahlenfeldern basieren.
<limits>	Größeninformationen wie minimaler Wert etc.
<numeric>	STL (Standard Template Library). Funktionen zur Summenbildung, der Differenz und des Skalarprodukts.

Ein- und Ausgabe

C++-Headerdatei	*Erläuterung*
<istream>	Eingabe.
<ostream>	Ausgabe.
<iostream>	Ein- und Ausgabe auf dem Strandard-Streams cin (Standardeingabe), cout (Standardausgabe) und cerr (Standardfehlerausgabe)
<iomanip>	Manipulatoren für die Ein- und Ausgabe.
<ios>	Grundlage für die Ein- und Ausgabe.
<iofwd>	Grundlage für die Ein- und Ausgabe.
<fstream>	Zeichen aus Dateien lesen oder in Dateien schreiben.
<streambuf>	Ein- und Ausgabe auf gebufferten Streams.
<sstream>	Zeichenketten lesen und in String-Objekte schreiben.

Fehlerbehandlung

C++-Headerdatei	*Erläuterung*
<exception>	Exception Handling. Ausnahmebehandlung.
<stdexept>	Fehlerreports.

Laufzeittyperkennung

C++-Headerdatei	*Erläuterung*
<typeinfo>	Laufzeit-Typinformation als Text.

Nationale Besonderheiten

C++-Headerdatei	*Erläuterung*
<local>	Kulturelle Unterschiede der Zeichen.

Speicherverwaltung

C++-Headerdatei	*Erläuterung*
<memory>	Speicheranforderung und –freigabe.
<new>	Speicheranforderung.

Hilfsfunktionen

C++-Headerdatei	*Erläuterung*
<utility>	Fügt zwei beliebige Werte zu einem Paar zusammen. Elemente bestehen aus einem Wert und einem Schlüssel. Der Schlüssel dient der eindeutigen Identifizierung.
<functional>	Konstruiert zum Beispiel das Komplement auf ein Prädikat.

1.2 Standard Template Library

Die Standard Template Library besteht aus folgenden Komponenten:

- Container nehmen Elemente beliebigen Datentyps auf.
- Iteratoren stellen Zeiger dar, mit denen auf ein bestimmtes Element im Container zugegriffen werden kann. Ein Container kann abhhängig von der Art des Iterators durchlaufen werden
- Algorithmen verarbeiten die in einem Container abgelegten Elemente. Algorithmen bieten Funktionalitäten wie zum Beispiel das Sortieren und Finden von Elementen in Containern an.

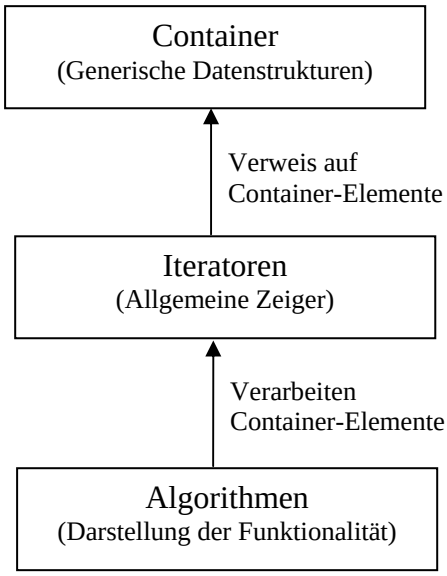
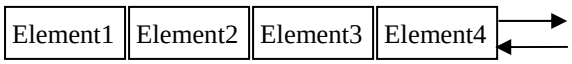
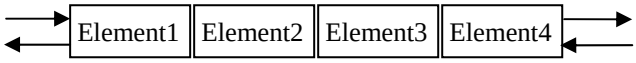
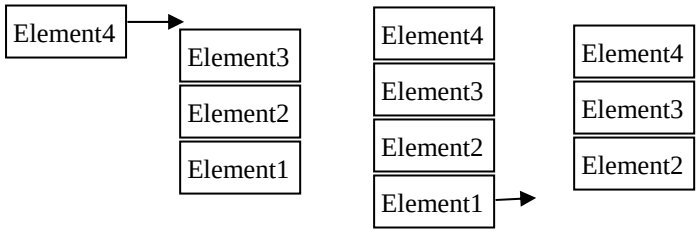
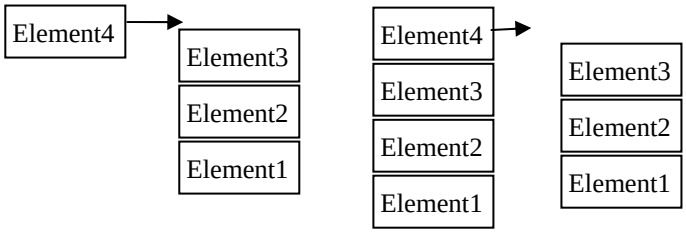
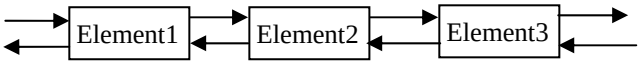


Abbildung 2 Zusammenarbeit der verschiedenen Komponenten der STL

Container

Container nehmen andere Objekte auf. Jeder der Container ist als Template realisiert und kann jeden beliebigen Datentyp aufnehmen. Die Elemente eines Containers können aus einem einfachen Datentyp wie zum Beispiel Integer, Strings oder Klassen bestehen.
Die Container werden in sequentielle und assoziative Container unterteilt. Sequentielle Container werden linear im Speicher abgelegt. Die Elemente in dem Container sind unsortiert. Das Anfügen oder Entfernen von Elementen am Anfang oder Ende wird sehr schnell ausgeführt. Eine Suche in einem unsortierten Container ist sehr langsam.
Elemente in einem assoziativen Container werden in einem Binärbaum sortiert abgelegt. Binärbäume können sehr schnell durchsucht werden. Ein Anfügen oder Entfernen von Elementen in einem Baum erfolgt sehr langsam.

Sequentielle	***Erläuterung***
<bitset>	Vektoren aus Bits.
<vector>	Ähnelt einem Array. Ein wahlfreier Zugriff auf die Elemente ist möglich. Die Anzahl seiner Elemente ist dynamisch veränderbar. Ein Löschen und Hinzufügen ist nur am Ende möglich 
<deque>	Zweiendige Warteschlangen. Daten können am Anfang und Ende hinzugefügt oder entfernt werden. 
<queue>	Warteschlangen (First in – First out - Speicher). 
<stack>	Stacks (Last In – First Out - Speicher). 
<list>	Doppelt verkettete Listen. Sequentieller Zugriff auf die Elemente. 

Als Beispiel wird der Container <vector> definiert:

```
#include <vector>
using std::vector;

int main(){
     // Vektor definieren
     vector<int> vect;

     // Vektor mit 5 Elementen definieren
     vector<int> vectElement(5);

     // Vektor mit 5 Elementen.
     // Jedes Element hat den Wert 10.
     vector<int> vectInhalt(5, 10);

     // Vektor duplizieren
     vector<int> vectCopy(vect);

     //Ein Vektor wird mit einem Array initialisiert
     int feld[] = {1, 2, 3, 4, 5};
     vector<int> vectFeld(feld, feld+sizeof(feld)/sizeof(int));
}
```

Die Anzahl der abgelegten Elemente in einem Standard-Container wird folgendermaßen ermittelt:

```
cout << "Größe des Feldes: " << vectFeld.size()<< endl;
```

Mit Hilfe von `reserve()` kann die Größe eines Vektors festgelegt werden:

```
vectFeld.reserve(20);
```

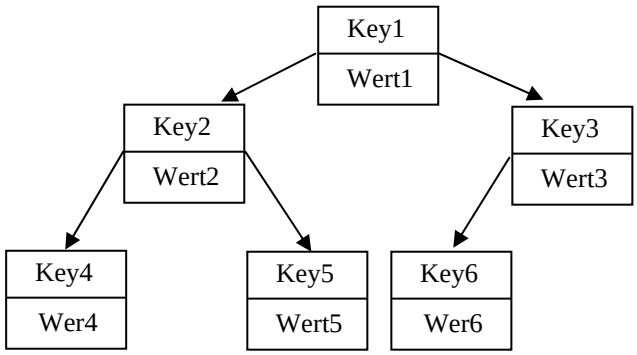
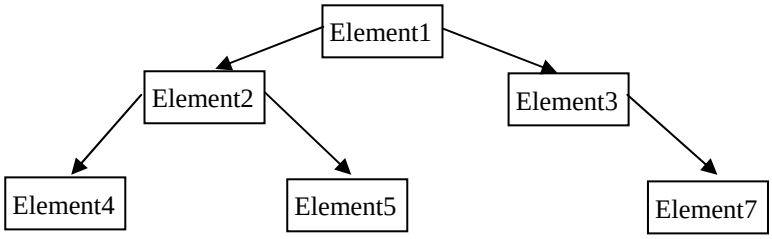
Wenn neuer Speicher angefordert wird, werden alle Element-Referenzen und –Zeiger sowie Iteratoren ungültig.

Die Maximalgröße des Vektors kann mit `capacity()` ermittelt werden.

```
cout << "Maximalgröße: "  << vectFeld.capacity() << endl;
```

Einem bestimmten Elemente des Containers kann ein Wert zugewiesen oder der Wert eines Elements kann ausgegeben werden.

```
vector<int> myVect(10);
unsigned int index;

// Über ein Index wird einem Vector-Element ein Wert zugewiesen
myVect[0] = 10;

// Mit Hilfe von push_back wird ein Element am Ende angefügt.
   myVect.push_back(5);

// Dem ersten Element (begin()) wird ein Wert zugewiesen.
myVect.insert(myVect.begin(),1);

// Dem letzten Element plus eins (end()) wird
// ein Wert zugewiesen.
myVect.insert(myVect.end(), 6);

// Die Elemente werden ausgegeben.
for (index = 0; index < myVect.size(); index++){
      cout << "Vektor Nr. " << index << ": " << myVect[index];
}
```

Assoziative	***Erläuterung***
<map>	Elemente, die aus einem Schlüssel und einen dazugehörigen Wert bestehen. Jeder Schlüssel ist einmal vorhanden. Die Elemente werden nach dem Schlüssel sortiert. Es können Tabellen, die auf Listen basieren oder binäre Bäume abgebildet werden. 
<multimap>	Maps mit möglichen Mehrfachvorkommen. Elemente, die aus einem Schlüssel und einen dazugehörigen Wert bestehen. Ein Schlüssel kann mehrmals vorkommen. Die Elemente werden nach dem Schlüssel sortiert.
<set>	Sortierte Menge von Objekten. Der Schlüsselwert kommt nur einmal vor. 
<multiset>	Sortierte Menge von Objekten, die mehrmals vorkommen können.

Beispiel für die Nutzung von Schlüssel-Wert-Paaren:

```
#include <map>
#include <iostream>
#include <iomanip>
#include < utility>
#include <string>

using namespace std;

int main(){
    map<string, string>adressbook;

     adressbook["aue"] = "aue@rrzn.uni-hannover.de";

     // pair fasst zwei Werte zu einem Objekt zusammen
     // Der Datentyp der Werte wird in eckige Klammern gesetzt.
     // Hier: String, string (Schlüssel, Wert)
    pair<string, string>paar1("firmaXYZ","firmaXYZ@myFirma.de");
    adressbook.insert(paar1);

    // Hier wird ein Datentyp für Schlüssel-Wert-Paare angelegt.
    typedef pair<string, string>element;
    element paar2("institutXYZ","institutXYZ@myInstitut.de");
    adressbook.insert(paar2);

    cout << "firma XYZ hat folgende E-Mail-Adresse ";
    cout << adressbook["firmaXYZ"];

}
```

`pair <Datentyp, Datentyp> Objektname` ist ein Standard-Datentyp, der in der Header-Datei <utility> definiert ist. Dieser Datentyp erstellt ein Objekt, welches zwei Werte zusammenfasst. Beispiel für zwei zusammengehörige Werte sind Schlüssel-Wert-Kombinationen. In eckige Klammern wird für den Wert und den Schlüssel der Datentyp, getrennt durch Kommatas angegeben. Als Datentyp können alle einfachen Datentypen in C++ oder Objekte genutzt werden. Objekte, die als Datentyp für `pair` genutzt werden, müssen folgende Voraussetzungen erfüllen:

- Die Objekte müssen kopierbar sein.
- Das Verhalten des Zuweisungsoperators muss definiert sein.
- Sie müssen vergleichbar sein.

Mit Hilfe der Methode `myPair.first` wird auf das erste Element und mit `myPair.second` wird auf das zweite Element zugegriffen.

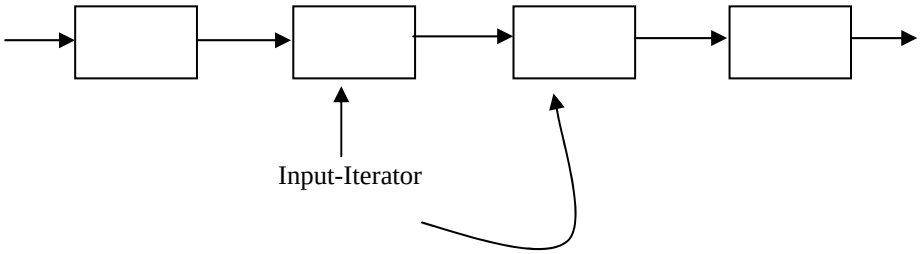
Beispiel für die Nutzung von Mengen:

```
#include <set>
#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

int main(){
    const int Anzahl = 3;
    const char* Obst[Anzahl] = {"Banane", "Apfel", "Erdbeere"};
    string slogan("Verkauf von Obst und Gemüse");
    set<string> setGemuese;
    set<char> setSlogan;

    // Zuweisung eines Arrays
    set<string> setObst(Obst, Obst + Anzahl);

    // Elemente werden als Strings behandelt.
    setGemuese.insert("Weisskohl");
    setGemuese.insert("Kartoffeln");

    // Zuweisung eines Strings an den Container set.
    // Es wird jedes Zeichen als ein Element betrachtet.
    setSlogan.insert(slogan.begin(), slogan.end());
}
```

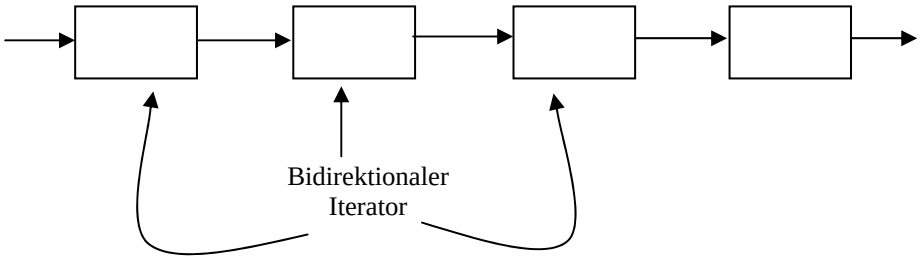
Iterator

Iteratoren arbeiten ähnlich wie Zeiger. Iteratoren sind Zeiger auf Elemente in einem beliebigen Container.
Die unten aufgelisteten Iteratoren können genutzt werden:

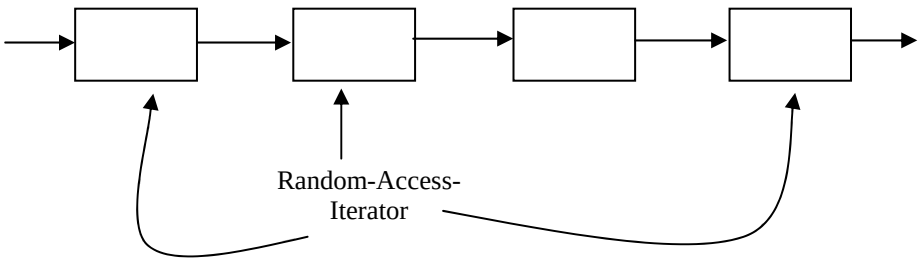
- Input-Iteratoren erlauben nur einen Lesezugriff auf die Elemente in einem Container. Der Zeiger kann sich nur vorwärts bewegen. Einmal gelesene Elemente können nicht erneut gelesen werden.



- Output-Iteratoren erlauben nur einen Schreibzugriff auf die Elemente in einem Container. Der Zeiger wird nur vorwärts bewegt. Der Zeiger arbeitet sequentiell.
- Forward-Iteratoren sind eine Mischung aus Input- und Output-Iteratoren. Der Zeiger hat sequentiell einen Schreib- und Lesezugriff auf die Elemente in einem Container.
- Bidirektionale Iteratoren können lesend und schreibend auf die Elemente zugreifen. Der Zeiger kann vorwärts und rückwärts die Elemente in einem Container durchlaufen. Der Iterator wird für den Container <list> benötigt.



- Random-Access-Iteratoren erlauben einen wahlfreien Zugriff. Die Elemente können nicht nur vorwärts und rückwärts durchlaufen werden, sondern es kann jedes Element auch direkt angesprochen werden. Der Iterator wird zum Beispiel für die Container <deque> oder <vector> benötigt.



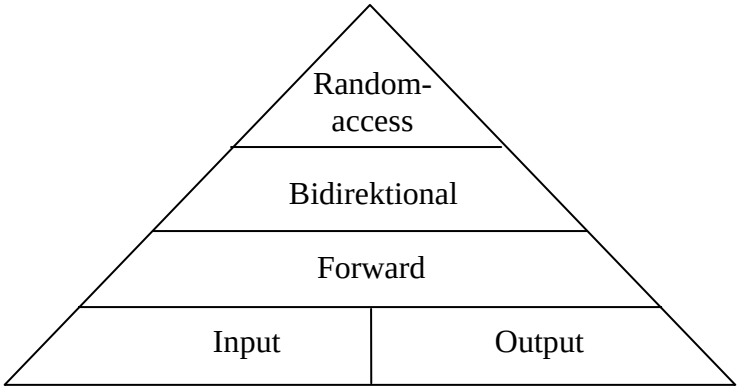


Abbildung 3: Hirarchie der Iteratoren untereinander

Die nachfolgende Tabelle enthält alle Aktionen, die mit Iteratoren möglich sind.

Aktion	***Code***
Aktuelles Element lesen	`*iter`
Member des akt. Elements lesen	`iter->member`
Wert schreiben	`*iter = wert`
Zuweisung	`iter1 = iter2`
Ein Element vorwärts	`++iter`
Ein Element zurück	`--iter`
Zugriff auf Element + n	`iter[n]`
Iterator um n erhöhen / vermindern	`iter + n` `iter - n`

Iteratoren vergleichen	`iter1 == iter2`
	`iter1 != iter2`
	`iter1 < iter1`
	`iter1 <= iter2`
	`iter1 > iter2`
	`iter1 >= iter2`

Standard-Konstruktor	`DATENTYP()`
Kopier-Kontruktor	`DATENTYP(iter)`

Aber nicht alle Iteratoren können alle Aktionen ausführen. Iteratoren, die sehr hoch in der Hirarchie stehen, können mehr Aktionen ausführen als Iteratoren, die sehr weit unten in der Hirarchie stehen. Iteratoren erben Aktionen von Iteratoren, die in der Hirarchie unter ihnen stehen.

Aktion	***Input***	***Output***	***Forward***	***Bi-direktional***	***Random-Access***
Aktuelles Element lesen	☑		☑	☑	☑
Member des akt. Elements lesen	☑		☑	☑	☑
Wert schreiben		☑	☑	☑	☑
Zuweisung			☑	☑	☑
Ein Element vorwärts	☑	☑	☑	☑	☑
Ein Element zurück				☑	☑
Zugriff auf Element + n					☑
Iterator um n erhöhen / vermindern					☑
Iteratoren vergleichen	☑ (Vergleich auf Gleichheit und Ungleichheit)		☑	☑	☑
Standard-Konstruktor			☑	☑	☑
Kopier-Kontruktor	☑	☑	☑	☑	☑

Mit Hilfe von Iteratoren kann bestimmten Elementen des Containers <vector> Werte zugewiesen werden.

```
// Definition eines Containers mit 5 Elementen
   vector<int> myVect(5);
// Definition eines allgemeinen Zeigers auf einen Container
vector<int>::iterator ptr;

// Dem Zeiger wird eine Position zugewiesen.
// Hier: Der Zeiger zeigt auf das 3.Element im Container.0
ptr = myVect.begin() + 2;

// Dieser Position wird ein Wert zugewiesen.
myVect.insert(ptr,3);

// Mit Hilfe des Dereferenzierungsoperators werden
// die Werte ausgegeben.
for (ptr = myVect.begin(); ptr < myVect.end(); ptr++){
            cout << "Vektor-Element: " << *ptr  << endl;
}
```

Mit Hilfe von konstanten Iteratoren kann nur lesend auf die Elemente eines Containers zugegriffen werden.

```
template <typename TDATA> void Ausgabe(const TDATA &container)
{
     typename TDATA::const_iterator ptrIter;

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

Dem Template wird eine Referenz auf einen konstanten Container übergeben. Wenn der Container konstant ist, muss der Iterator auf diesen Container auch konstant sein. Der Datentyp des Iterators wird über eine `typedef`-Anweisung festgelegt.

Mit Hilfe des Reverse-Iterators kann ein Container in umgekehrter Reihenfolge ausgegeben werden.

```
template <typename TDATA> void Ausgabe(TDATA &container)
{
     typename TDATA::reverse_iterator ptrIter;

     for (ptrIter = container.rbegin(); ptrIter < container.rend();
         ++ptrIter)
     {
            cout << *ptrIter << endl;
     }
}
```

Mit Hilfe von Iteratoren kann auf Schlüssel-Wert-Paare zugegriffen werden.

```
#include <map>
#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

int main(){
    map<string, string>adressbook;
    map<string, string>::iterator itptr;

    adressbook["aue"] = "aue@rrzn.uni-hannover.de";
    adressbook["firmaXYZ"] = "firmaXYZ@myFirma.de";
    adressbook["institutXYZ"] = "institutXYZ@myFirma.de";

    itptr = adressbook.find("firmaXYZ");

    if (itptr != adressbook.end()){
            // first = Zugriff auf den Schlüssel eines Elements
            cout << itptr->first << " hat die E-Mail-Adresse ";
            // second = Zugriff auf den Wert eines Elements
            cout << itptr->second << endl;
    }

    cout << "Anzahl der Einträge: " << adressbook.count("firmaXYZ") << endl;
}
```

Mit Hilfe des `ostream_iterators` können alle Elemente eines Containers auf dem Bildschirm ausgegeben werden.

```
#include <iostream>
#include <iomanip>
#include <set>
#include <iterator>
#include <string>

using namespace std;

int main(){
    const int Anzahl = 4;
    const char* Obst[Anzahl] = {"Banane", "Apfel", "Erdbeere", "Karotte"};
    const char* Gemuese[Anzahl] = {"Weisskohl", "Karotte", "Kartoffel"};

    set<string> setObst(Obst, Obst + Anzahl);
    set<string> setGemuese(Gemuese, Gemuese + Anzahl - 1);
    set<string> waren;

    cout << "Menge Obst: ";
    copy(setObst.begin(), setObst.end(),
            // Eingebunden in der Header-Datei <iterator>
            // ostream_iterator<DATENTYP> iter(ostream,Trennzeichen);
            // Der komplette Inhalt des Containers wird
            // auf dem Bildschirm angezeigt.
            ostream_iterator<string>(cout, " "));
    cout << endl;
    cout << "Menge Gemüse: ";
    copy(setGemuese.begin(), setGemuese.end(),
            ostream_iterator<string>(cout, " "));
    cout << endl;
}
```

Mit Hilfe von `ostream_iterator<DATENTYP> iter(ostream,Trennzeichen)` kann ein Container vollständig in eine Datei oder auf dem Bildschirm ausgegeben werden. Dem Iterator-Typ folgt der Datentyp der zu übertragenen Daten. Anschließend wird ein beliebiger Name für den Iterator vergeben. Dem Iterator werden der Ausgabestream und ein Trennzeichen für die Elemente des Containers übergeben. Das Gegenstück ist `istream_iterator<DTYP> iter(istream).`

Algorithmen

Mit Hilfe von Algorithmen können die Elemente in einem Container bearbeitet werden. Die zur Verfügung gestellten Algorithmen lassen sich in folgende Kategorien einteilen:

Nicht-modifizierende	***Beschreibung***
… verändern nicht die Werte der Elemente oder deren Reihenfolge. … können auf alle Standard-Container angewandt werden.	
adjacent_find()	Sucht nach benachbarten Elementen, deren Werte gleich sind.
count()	Zählt die Anzahl der Elemente, die einen bestimmten Wert besitzen.
count_if()	Zählt die Anzahl der Elemente, die einer bestimmten Bedingung entsprechen.
equal()	Vergleicht zwei Bereiche auf Gleichheit.
find()	Sucht nach Elementen mit einem bestimmten Wert.
find_if()	Sucht nach Elementen, die eine bestimmte Bedingung erfüllen.
find_end()	Sucht nach dem letzten Auftreten eines Elements.
find_first_of()	Sucht nach dem ersten Auftreten eines Elements.
for_each()	Für jedes Element wird eine Operation durchgeführt.
lexicographical_compare	Zwei Bereiche werden lexikographisch miteinander verglichen.
max_element()	Liefert das größte Element.
min_element()	Liefert das kleinste Element.
mismatch()	Liefert die Position zurück, an der zwei Container verschieden sind.
search()	Sucht nach einem übereinstimmenden Bereich.
search_n()	Sucht nach dem ersten n-fachen Auftreten eines Elements.

Modifizierende	***Beschreibung***
… ändern die Werte von Elementen. … können nicht auf assoziative Container wie map und set angewandt werden.	
copy()	Kopiert einen Bereich.
copy_backward()	Kopiert einen Bereich. Das letzte Element des Bereichs wird als erstes kopiert.
for_each()	Führt für jedes Element eine Operation durch.
generate()	Über eine definierte Operation werden Elemente erzeugt.
generate_n()	Erzeugt für n-Elemente neue Elemente.
merge()	Vereinigt zwei Bereiche.
replace()	Ersetzt in einem Bereich bestimmte Werte der Elemente.
replace_if()	Ersetzt in einem Bereich Elemente, die einer bestimmten Bedingung genügen.
replace_copy_if()	Ersetzt in einem Bereich bestimmte Werte der Elemente, die einer Bedingung entsprechen und kopiert diese.
swap_ranges()	Vertauscht zwei Bereiche.
fill()	Ersetzt in einem Bereich die Elemente durch andere.
fill_n()	Ersetzt n-Elemente durch ein anderes Element.
transform()	Verändert die Werte von Elementen, aber die Elemente nicht.

Löschende	***Beschreibung***
… entfernen Elemente aus einem Container. … überschreiben die zu löschenden Elemente mit den nachfolgenden Elementen. … verkleinern nicht den Speicherplatz eines Containers. … können nicht auf assoziative Container wie map und set angewandt werden.	
remove()	Entfernt ein Element mit einem bestimmten Wert.
remove_if()	Entfernt ein Element, welches einer bestimmten Bedingung genügt.
remove_copy_if()	Kopiert alle Elemente eines Bereichs, die nicht einer bestimmten Bedingung genügen.
unique()	Entfernt unmittelbar aufeinander folgende gleiche Werte.
unique_copy()	Kopiert alle Elemente. Unmittelbar aufeinander folgende gleiche Werte werden nur einmal kopiert.

Mutierende	***Beschreibung***
… ändern die Reihenfolge der Elemente (Permutation) in einem Container. … sollten auf sortierte Elemente angewandt werden. … können nicht auf assoziative Container wie map und set angewandt werden.	
next_permutation()	Bildet die nächste Kombination der Elemente.
prev_permutation()	Bildet die vorherige Kombination der Elemente.
partition()	Bringt Elemente, die einer bestimmten Bedingung entsprechen, an den Anfang des Containers.
stable_paration()	Bringt Elemente, die einer bestimmten Bedingung entsprechen, an den Anfang des Containers. Die relative Anordnung der Elemente bleibt unverändert.
random_shuffle()	Mischt die Elemente zufällig.
reverse()	Kehrt die Reihenfolge der Elemente um.
reverse_copy()	Kopiert die Elemente in umgekehrter Reihenfolge.
rotate()	Verschiebt die Elemente.
rotate_copy()	Kopiert die Elemente, in einer verschobenen Reihenfolge.

Zum Beispiel die Buchstaben a, b, c haben insgesamt sechs Permutationen:
abc, acb, bac, bca, cab, cba.
Die Anzahl von Permutationen wird wie folgt berechnet: n! = Anzahl. 3! = 6.
Um alle Permutationen mit Hilfe eines Algorithmus zu berechnen, muss der Algorithmus solange aufgerufen werden, bis dieser false liefert.

Algorithmen	***Beschreibung***
… zur Erstellung eines Heaps.	
make_heap()	Konvertiert einen Bereich in einen Heap.
pop_heap()	Entfernt ein Element vom Heap und fügt es am Ende des Containers wieder ein.
push_heap()	Fügt ein Element zum Heap hinzu.
sort_heap()	Konvertiert ein Heap in eine sortierte sequenzielle Reihenfolge.

Algorithmen	***Beschreibung***
… zum Sortieren von Elementen in einem Container.	
nth_element()	Sortiert die Elemente so, dass die n kleinsten / größten Elemente am Anfang des Containers stehen.
partial_sort()	Sortiert die Elemente solange, bis n Elemente in sortierter Reihenfolge vorliegen.
partial_sort_copy()	Kopiert die Elemente solange um, bis n Elemente in sortierter Reihenfolge vorliegen.
partition()	Die Elemente, die einer bestimmten Bedingung genügen, werden an den Anfang des Containers gesetzt.
stable_partition()	Die Elemente, die einer bestimmten Bedingung genügen, werden an den Anfang des Containers gesetzt. Die Elemente behalten innerhalb des Containers ihre relative Reihenfolge.
sort()	Sortiert Elemente.
stable_sort()	Sortiert Elemente. Die Elemente behalten die ihre Reihenfolge.

Algorithmen	***Beschreibung***
… die auf sortierte Bereiche angewendet werden.	
binary_search()	Durchsucht einen Bereich nach einem bestimmten Wert.
includes()	Überprüft, ob alle Elemente eines Bereichs in einem anderen Bereich vorkommen.
lower_bound()	Sucht nach dem ersten Element das größer gleich einen bestimmten Wert ist.
upper_bound()	Sucht nach dem ersten Element das größer als ein bestimmter Wert ist.
equal_range()	Kombination aus lower_bound() und upper_bound().
merge()	Fügt Elemente aus zwei Bereichen zusammen.
set_union	Bildet aus zwei Bereichen die Vereinigungsmenge.
set_intersection()	Bildet aus zwei Bereichen die Schnittmenge.
set_difference()	Bildet aus zwei Bereichen die Differenzmenge.
set_symmetric_difference()	Zwei Bereiche werden mit einem exklusiv Oder verbunden. D. h. es werden nur Elemente übernommen, die nicht im ersten und zweiten Bereich vorkommen.
inplace_merge()	Fügt zwei sortierte Bereiche in einem Container zusammen.

Algorithmen	***Beschreibung***
… zur Berechnung von Werten.	
accumulate()	Führt Berechnungen über alle Elemente eines Bereichs aus. Standard: Addition aller Elemente (Ergebnis = wert1 + wert2 + wert3).
adjacent_difference()	Führt Berechnungen mit benachbarten Elementen eines Bereichs durch. Standard: Differenzbildung (Ergebnis1 = wert1; Ergebnis2 = wert1 - wert2; Ergebnis3 = wert3 – wert2).
inner_product()	Führt Berechnungen über alle Elemente aus zwei Bereichen durch. Standard: Multiplikation der Elemente (Ergebnis = (cont1El1 * Cont2El1) * (cont1El2 * Cont2El2))
partial_sum()	Führt eine fortlaufende Berechnung mit allen Elementen eines Bereichs durch. Standard: fortlaufende Summe (Ergebnis1 = wert1; Ergebnis2 = wert1+ wert2; Ergebnis3 = wert1 + wert2 + wert3).

Als Beispiel wird in einem <vector> ein bestimmter Wert gesucht und der <vector> anschließend sortiert.

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

using namespace std;

int main(){
    vector<int> myVect(5);
     vector<int>::iterator ptr;
     int count;

     count = 5;
     //
     for (ptr = myVect.begin(); ptr < myVect.end(); ptr++){
            *ptr = count;
            if ((count % 2) != 0){
           count = count * 5 ;
            }
            else {
                   count = count % 2;
            }
    }

    ptr = find(myVect.begin(), myVect.end(), 5);

    if(ptr == myVect.end()){
     cout << "Die Ziffer 5 ist nicht vorhanden." << endl;
    }
    else {
     cout << "Die Ziffer 5 ist vorhanden" << endl;
    }

     sort(myVect.begin(), myVect.end());

     for (ptr = myVect.begin(); ptr < myVect.end(); ptr++){
            cout << "Vektor-Element: " << *ptr  << endl;
     }
}
```

Mit Hilfe von Algorithmen können Schnittmengen etc. auf Mengen gebildet werden.

```
#include <iostream>
#include <iomanip>
#include <set>
#include <iterator>
#include <string>

using namespace std;

int main(){
    const int Anzahl = 4;
    const char* Obst[Anzahl] = {"Banane", "Apfel", "Erdbeere", "Karotte"};
    const char* Gemuese[Anzahl] = {"Weisskohl", "Karotte", "Kartoffel"};

    set<string> setObst(Obst, Obst + Anzahl);
    set<string> setGemuese(Gemuese, Gemuese + Anzahl - 1);
    set<string> waren;

               ;
    cout << endl;
    cout << "Union: ";
    set_union(setObst.begin(), setObst.end(), setGemuese.begin(),
                  setGemuese.end(), ostream_iterator<string>(cout, " "));
    cout << endl;
    cout << "Intersection: ";
    set_intersection(setObst.begin(), setObst.end(), setGemuese.begin(),
                     setGemuese.end(), ostream_iterator<string>(cout, " "));
    cout << endl;
    cout << "Difference: ";
    set_difference(setObst.begin(), setObst.end(), setGemuese.begin(),
                         setGemuese.end(), inserter(waren, waren.begin()));
    copy(waren.begin(), waren.end(), ostream_iterator<string>(cout, " "));
    cout << endl;
}
```

Vorteile

In der Standard Template Library sind selber Datenstrukturen unabhängig von einem Datentyp in Form von Templates implementiert.

Zum Beispiel können doppelt verkettete Listen mit Hilfe des Containers <list> sehr einfach erstellt und durch die vorhandenen Methoden verändert werden.

```
#include <list>
#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

int main(){
    list<int> zahlen;
    list<int>::iterator iter;

    // Liste füllen
    zahlen.push_back(5);
    zahlen.push_back(3);
    zahlen.push_back(1);
    zahlen.push_back(5);
    zahlen.push_back(2);

    // Ausgabe: 5 3 1 5 2
    for(iter=zahlen.begin(); iter!=zahlen.end(); iter++){
     cout << *iter << " ";
    }
    cout << endl;

    //Liste drehen
    zahlen.reverse();
    // Ausgabe: 2 5 1 3 5
    for(iter=zahlen.begin(); iter!=zahlen.end(); iter++){
     cout << *iter << " ";
    }
    cout << endl;

    // Ein Element mit dem Wert 5 löschen
    iter = find(zahlen.begin(), zahlen.end(), 5);
    if (iter != zahlen.end()){
         zahlen.remove(*iter);
    }
    // Ausgabe: 2 1 3 5
    for(iter=zahlen.begin(); iter!=zahlen.end(); iter++){
     cout << *iter << " ";
    }
    cout << endl;


}
```

Die Implementierung einer doppelt verkettenden Liste ohne die Standard-Template-Library würde folgendermaßen aussehen:

```
#include <iostream>
#include <iomanip>
#include <new>

using namespace std;

struct zahlen{
       int wert;
       struct zahlen *next; /* Zeiger auf das nachfolgende Element*/
       struct zahlen *previous; /* Zeiger auf das vorhergehende Element*/
};

struct zahlen *next;
struct zahlen *anfang;
struct zahlen *ende;

void list_init(){
   /*
     Zeiger auf das nächste Elemente; Ende der Kette
     <=> struct zahlen *next = NULL;
     Zeiger auf das erste Element <=> struct zahlen *anfang = NULL;
     Zeiger auf das letzte Element <=> struct zahlen *ende = NULL;
  */
   next = anfang = ende = 0;

}

void anhaengen(int wert){
     struct zahlen *ptr_zahlen, *ptr_temp;

     /* Kein Element in der Liste vorhanden */
     if (anfang == 0){
        /* Speicherplatz für die Struktur reservieren */
        anfang = new zahlen;

        if (anfang == 0){
            cout << "Kein Speicherplatz zur Verfuegung.";
            cout << endl;
        }
        anfang->wert = wert;
        anfang->next = 0;
        ende = anfang;
        ende->previous = 0; /* = anfang->previous = NULL*/
     }
     else{
          ptr_zahlen = anfang; /* Zeiger auf das erste Element */
          while(ptr_zahlen->next != 0){
            ptr_zahlen = ptr_zahlen->next;
          }

          ptr_zahlen->next = new zahlen;
          if (ptr_zahlen->next == 0){
            cout << "Kein Speicherplatz zur Verfuegung.";
            cout << endl;
          }
          ptr_temp = ptr_zahlen;
          ptr_zahlen = ptr_zahlen->next;
          ptr_zahlen->wert = wert;
          ptr_zahlen->next = 0;
          ende = ptr_zahlen;
          ptr_zahlen->previous = ptr_temp;

     }
```

```
}

void loesche(int wert){
     struct zahlen *ptr_zahlen, *ptr_temp, *ptr_tmp;

     /* In der Liste sind Elemente vorhanden */
     if (anfang != 0){

        /* Das erste Element ist das gesuchte Element */
        if (anfang->wert == wert){
           ptr_zahlen = anfang->next;

           if (ptr_zahlen == 0){
              delete anfang;
              anfang = 0;
              ende = 0;
           }
           else{
                ptr_zahlen->previous = 0;
                delete anfang;
                anfang = ptr_zahlen;
           }
        }
        /* Das zu löschende Element ist das letzte */
        else if (ende->wert == wert){
             ptr_zahlen = ende->previous;
             ptr_zahlen->next = 0;
             ptr_temp = ende;
             ende = ptr_zahlen;
             delete ptr_temp;
             ptr_temp = 0;
        }
        else{
             /* Suche nach dem gewünschten Element */
             ptr_zahlen = anfang;

             while(ptr_zahlen != 0){
                   ptr_temp = ptr_zahlen->next;

                   /* Falls gefunden, wird der Speicher freigegeben */
                   if (ptr_temp->wert == wert){
                      ptr_zahlen->next = ptr_temp->next;
                      ptr_tmp = ptr_temp->next;
                      ptr_tmp->previous = ptr_zahlen;
                      delete ptr_temp;
                      ptr_temp = 0;
                      break;
                   }

                   ptr_zahlen = ptr_temp;
             }
        }
     }
}
```

```
void vertausche(){
     struct zahlen *ptr_rechts, *ptr_links;
     int temp;

     if (anfang != ende) {
            ptr_rechts = anfang;
         ptr_links = ende;

             while(ptr_links != ptr_rechts){
                     temp = ptr_rechts->wert;
                     ptr_rechts->wert = ptr_links->wert;
                   ptr_links->wert = temp;
                   ptr_links = ptr_links->previous;
                     ptr_rechts = ptr_rechts->next;
             }
     }

}

void ausgabe(){
     struct zahlen *ptr_zahlen;

     ptr_zahlen = anfang;

     while(ptr_zahlen != 0){
            cout << ptr_zahlen->wert << " ";
            ptr_zahlen = ptr_zahlen->next;
     }
     cout << endl;

}

int main(){
    // Liste initialisieren
    list_init();
    // Elemente in die Liste einfügen
    anhaengen(5);
    anhaengen(3);
    anhaengen(1);
    anhaengen(5);
    anhaengen(2);
    ausgabe();
    // Elemente vertauschen; Liste wird von hinten nach vorn gelesen
    vertausche();
    ausgabe();
    // Der Wert 5 wird gelöscht
    loesche(5);
    ausgabe();
}
```

1.3 Strings

Zeichenketten werden in C als Feld vom Typ char dargestellt. In der Headerdatei <string.h> (gleichwertig mit <cstring>) werden dem Nutzer viele Funktionen für das Durchsuchen, Verketten, Kopieren etc. von Strings zur Verfügung gestellt. Der Entwickler muss aber zum Beispiel Tests auf Bereichsüberschreitungen beim Einfügen von Elementen entwickeln.
Wenn der Datentyp `string` aus der Standard-Template-Library genutzt wird, wird der notwendige Speicher automatisch reserviert und falls nötig erweitert. Um den Datentyp zu nutzen muss die Headerdatei <string> eingebunden werden. Es wird der Namensraum std genutzt.

Eine Variable vom Datentyp string wird folgendermaßen definiert:

- `std::string buchstaben;`
  Eine Variable vom Datentyp string wird definiert. Der Name der Variablen ist frei wählbar. Die Variable ist leer. Sie enthält keine Zeichen.
- `std::string alphabet("abc");`
  Hier wird die Variable gleichzeitig mit einer Zeichenkette initialisiert. Eine Zeichenkette wird immer in Anführungszeichen eingefasst.
- `std::string zeichen(10, ' ');`
  Es wird Platz für 10 Zeichen reserviert. Die Variable wird mit 10 Leerzeichen initialisiert. Es kann jedes beliebige Zeichen aus der ASCII-Tabelle genutzt werden. Ein Zeichen wird immer in Apostroph gefasst.

Mit Hilfe des Gleichheitszeichens kann einer Variablen vom Datentyp string

- eine Zeichenkette (`alphabet = "ABCDE"`),
- eine andere Zeichenkette (`zeichen = alphabet`) oder
- ein einzelnes Zeichen (`buchstaben = 'A'`)

zugewiesen werden.

Das erste Element in einem String hat den Index 0. Mit Hilfe der Methode `.at(Index)` kann auf ein Zeichen innerhalb des Strings zugegriffen werden. Zum Beispiel `zeichen.at(3)` liefert 'D' zurück. Gleichzeitig testet die Funktion den übergebenen Index auf eine mögliche Bereichsüberschreitung.

Die Länge eines Strings kann mit Hilfe der Methode `.size()` ermittelt werden. `zeichen.size()` liefert den Wert 5 zurück.

Der Datentyp string kann in ein C-String mit Hilfe der Methode `.c_str()` konvertiert werden. Die Methode liefert einen Zeiger vom Datentyp const char* und somit ist der Inhalt des Strings nicht direkt über diesen Zeiger veränderbar. Der Zeiger kann mit Hilfe der Typkonvertierung `const_char<char*>` in einen nicht-konstanten Zeiger vom Datentyp char umgewandelt werden.

```
char *cZeichen;
cZeichen = const_cast<char*>(zeichen.c_str());
```

Der umgewandelte String kann auch in ein Feld vom Datentyp char kopiert werden und dort weiterverarbeitet werden.

```
char charFeld[40];
strcpy(charFeld,zeichen.c_str());
```

Variablen vom Datentyp string können mit Hilfe der Methode `.compare()` verglichen werden. Folgende Möglichkeiten sind vorhanden:

- Eine Variable vom Datentyp string wird mit einer anderen Variablen verglichen (`zeichen.compare(alphabet)`).
- Eine Variable vom Datentyp string wird mit einer Zeichenkette verglichen (`zeichen.compare("abc")`).
- Es werden Teilzeichenketten miteinander verglichen. Zum Beispiel: `zeichen.compare(2,2, alphabet,3,1)`. In diesem Beispiel werden zwei Zeichen ab dem zweiten Zeichen aus dem String zeichen mit einem Zeichen ab dem dritten Zeichen aus dem String alphabet verglichen.

Die Methode liefert als Rückgabewert

- … eine Null, wenn die Zeichenketten gleich sind.
- … einen positiven Wert, wenn String1 lexikografisch String2 folgt.
- .. einen negativen Wert, wenn String1 lexikografisch vor String2 liegt.

Ein lexikografischer Vergleich berücksichtigt die Position der Zeichen in der ASCII-Tabelle. Die einzelnen Zeichen liegen in der Reihenfolge
'0' < … < '9' < 'A' < … < 'Z' < 'a' … < 'z' vor.
Wenn zum Beispiel String1 den Wert „Walross“ und String2 den Wert „Eisbär“ besitzt, liegt String1 im Alphabet hinter String2.
Strings können auch mit Hilfe der Vergleichsoperatoren ==, !=, <, >, <=, => verglichen werden.

Mit Hilfe des Operanden += kann ein Zeichen an ein String angehängt werden.
zeichen += alphabet;
zeichen += "zxy";
zeichen += 'Z';
Wenn Teilstrings angehängt werden sollen, kann die Methode .append() genutzt werden.

```
// Die Zeichenkette alphabet wird vollständig angehängt
zeichen.append(alphabet);

// Von der Zeichenkette wug werden zwei Zeichen
// ab dem zweiten Zeichen angehängt.
 zeichen.append("wug",1,2);
```

Zeichen werden immer am Ende einer Zeichenkette angehängt.

Mit Hilfe der Methode `.Insert()` können Zeichen an einer beliebigen Position in eine andere Zeichenkette eingefügt werden.
An einer bestimmten Position wird eine konstante Zeichenkette oder eine Variable vom Datentyp string eingefügt.

```
pos = zeichen.size() - 1;
zeichen.insert(pos, "ZY");
```

An einer bestimmten Position im String wird eine bestimmte Anzahl von Zeichen eingefügt. Das Zeichen und die Häufigkeit sind frei wählbar.

```
zeichen.insert(pos, 3, '-');
```

Strings können mit Hilfe des Operators + verkettet werden. Dabei können nicht nur Strings mit Strings verkettet werden, sondern auch Strings mit einem Feld vom Datentyp char.

```
char charFeld[40];
zeichen = charFeld + alphabet;
zeichen = zeichen + "ZYX";
```

Innerhalb einer Zeichenkette kann nach einem Teilstring gesucht werden.
Für das Beispiel wurden folgende Variablen definiert.

```
// Text, der durchsucht wird
string text = "die Edda-Saga, die zwischen  ...";
// Text, der in der Variablen text gesucht wird
string suche = "die";
// An welcher Position kommmt der Suchtext vor
int pos = 0;
```

Mit Hilfe der Methode `.find(Suchstring)` wird ein String vom ersten bis zum letzten Zeichen nach einem Teilstring durchsucht.

```
pos = text.find(suche);
cout << pos << endl; // ==  0
```

Mit Hilfe der Methode `.rfind(Suchstring)` wird ein String vom letzten bis zum ersten Zeichen nach einem Teilstring durchsucht.

```
pos = text.rfind(suche);
cout << pos << endl; // == 15
```

Beide Methoden liefern das erste Auftreten des Teilstrings zurück. Wenn alle Position eines Teilstrings in einem String ermittelt werden sollen, muss mit einer Schleife gearbeitet werden.
Bei der Suche nach einem Teilstring wird wie bei `.compare()` die Groß- und Kleinschreibung beachtet.

Wenn ein String nach einem bestimmten Zeichen durchsucht warden soll, können die Methoden `find_first_of()`, `find_first_not_of`, `find_last_of()` und `find_last_not_of()` genutzt werden.

```
string lower = "abcdefghijklmnopqrstuvwxyz";
string upper = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
string zahl = "1234567890";
string satzende = "?!.";
string anyChar = "abcdERFR3456";
int pos = 0;
pos = anyChar.find_first_of(lower);           // ==  0
pos = anyChar.find_last_of(satzende);         // == -1
pos = anyChar.find_first_not_of(upper);// == 0
pos = anyChar.find_last_not_of(zahl);   // == 7
```

Zeichen können in einem String ersetzt werden. Mit Hilfe der Methode .replace() kann innerhalb eines Strings ein oder mehrere Zeichen ab einer bestimmten Position durch ein oder mehrere Zeichen ersetzt. Die Zeichen in einem String können durch eine Konstante oder Variable vom Datentyp string ersetzt werden. Es kann auch ein Feld vom Datentyp char genutzt werden.

```
string text = "Guten Morgen, Frau ...";
string suche = "Frau";
const char* ersetze = "Herr";
int pos = 0;
int anzahl = 0;
pos = text.find(suche);
anzahl = suche.size();
text.replace(pos, anzahl, ersetze, strlen(ersetze));
```

Mit Hilfe von `.clear()` kann ein String vollständig gelöscht werden. Ob der vom String belegte Speicher freigegeben wird, ist implementierungsabhängig.
Für das Löschen von Teilstrings kann `.erase()` eingesetzt werden. Der erste Parameter gibt immer die Startposition an, ab der zu löschen ist. Mit einem zweiten Parameter kann die Anzahl der zu löschenden Zeichen angegeben werden.
Mit Hilfe der Methode `.empty()` wird abgefragt, ob ein String leer ist.

```
string text = "Der Eisbaer in Sibirien ...";
string suche = "Sibirien";
int anzahl = 0;
int pos = 0;
int posClear = 0;
if (!(text.empty())){
            anzahl = text.size();
            pos = text.find(suche);
            posClear = anzahl - pos;
            text.erase(pos, posClear);
 }
```

Teilstrings können in andere Zeichenketten hineinkopiert werden.
Die Methode `.substr()` kopiert einen String vollständig in einen anderen.
Der Methode kann als erstes Argument die Anfangsposition und als zweites Argument die Anzahl der zu kopierenden Zeichen übergeben werden. Wenn nur die Anfangsposition angegeben wird, wird ein String ab einer bestimmten Position vollständig in einen anderen kopiert.

```
string text = "Der Eisbaer in Sibirien ...";
string suche = "Eisbaer";
string tiere;
int anzahl = 0;
int pos = 0;
int posClear = 0;
if (!(text.empty())){
        anzahl = suche.size();
        pos = text.find(suche);
        tiere = text.substr(pos, anzahl);
}
```