Die Standardbibliothek von C++

Programmierung

Alle Programme in diesem Buch wurden mit dem nicht kommerziellen und sehr leistungsfähigen GNU-C++-Compiler g++ getestet, der für unterschiedliche Plattformen angeboten wird.

PEARSON Education

ADDISON-WESLEY

Helmut Herold

Die Standardbibliothek von C++

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Bibliografische Information Der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar

10 9 8 7 6 5 4 3 2 1

07 06 05

ISBN 3-8273-2267-7-1

ein Imprint der Pearson Education Deutschland GmbH,
Martin-Kollar-Straße 10-12, D-81829 München/Germany

Einbandgestaltung: Marco Lindenbeck, webwo GmbH (mlindenbeck@webwo.de)
Lektorat: Sylvia Hasselbach, shasselbach@pearson.de
Korrektorat: Werner Siedenburg
Herstellung: Elisabeth Prümm, epruemm@pearson.de
Satz: LaTeX
Belichtung, Druck und Bindung: Bercker Graph. Vertrieb, Kevelaer
Printed in Germany

Inhaltsverzeichnis

Kapitel 1

Komplexität und Aufwand

Für einen effizienten Einsatz der C++-Standardbibliothek muss man die Komplexität deren Komponenten ebenso kennen wie den Aufwand, den sie im schlimmsten Fall (*worst case*) benötigen. In der folgenden Tabelle sind mögliche Komplexitäten gezeigt, die ein Algorithmus bzw. eine STL-Komponente für die Durchführung ihrer Aufgabe bei `n` Elementen benötigen kann. Die Zeiten sind dabei in dieser Tabelle nach ihrer Geschwindigkeit angeordnet, also die schnellste zuerst und die langsamste zuletzt, wobei `log` für den „Zweier-Logarithmus“ steht:

Komplexität	**kurze Beschreibung**
O(1)	(*konstant*) Jede Anweisung wird höchstens einmal ausgeführt. Dies ist der Idealzustand für einen Algorithmus. Ein solches Programm benötigt immer eine konstante Zeit, unabhängig von der Anzahl von Elementen.
O(log n)	(*logarithmisch*) Hier bewirkt z. B. eine vierfache Datenmenge doppelten, eine 8-fache Datenmenge 3-fachen und eine 1024-fache Datenmenge 10-fachen Ressourcenverbrauch.
O(n)	(*linear*) Speicher- oder Zeitverbrauch wachsen direkt proportional mit der Problemgröße `n`.
O(n log n)	(*n-logarithmisch*) liegt zwischen `n` (*linear*) und n^2 (*quadratisch*).
$O(n^2)$	(*quadratisch*) Solche Algorithmen lassen sich praktisch nur für kleine Probleme anwenden.

Bei der Vorstellung der entsprechenden STL-Komponenten, deren Komplexität von Wichtigkeit ist, wird diese in Form der O-Notation in Klammern angegeben.
Neben den in der vorherigen Tabelle vorgestellten Komplexitätsfunktioenen existieren natürlich noch weitere mögliche Komplexitäten von Algorithmen, die aber glücklicherweise kein Algorithmus in der Standardbibliothek aufweist, wie z. B.:

$O(n^3)$	(*kubisch*) Solche Algorithmen lassen sich in der Praxis nur für sehr kleine Problemgrößen anwenden.
$O(2^n)$	(*exponentiell*) Bei doppelter, dreifacher und 10-facher Datenmenge steigt der Ressourcenverbrauch auf das 4-, 8- bzw. 1024-fache. Solche Algorithmen sind praktisch kaum verwendbar.

Beispiel für einen logarithmischen und linearen Algorithmus ($O(\log n)$ und $O(n)$)

Programm 1.1 demonstriert den Unterschied zwischen einem logarithmischen und einem linearen Algorithmus, indem es einmal die Potenz zu einer Zahl mittels des Legendre-Algorithmus (`legendre_log()`) und einmal mit `n` Schleifendurchläufen (`hoch_linear()` ermittelt. Es gibt dabei jeweils die benötigten Schleifendurchläufe und die benötigten Zeiten aus.

Programm 1.1 – `algolinlog.cpp`:
Potenzieren mit Legendre-Algorithmus und mit n Schleifendurchläufen

```
#include <ctime>
#include <cmath>
#include <iomanip>
#include <iostream>
using namespace std;

#define AUSGABE(x) \
   cout << setw(10) << x << " (" << setw(6) \
        << (((double)clock()-start) / CLOCKS_PER_SEC) << ") |";
double legendre_log(double a, int b) {
  double x = a, z = 1;
  int    y = b, i = 0;
  clock_t start = clock();
   while (y>0) {
      if (y%2 != 0) // y ungerade ?
         z = z * x;
      y = y / 2;
      x = x * x;
      i++;
   }
   AUSGABE(i)
   return z;
}
double hoch_linear(double a, int b) {
  double  x = 1;
  clock_t start = clock();
   for (int i=1; i<=b; i++)
      x = x * a;
   AUSGABE(b) cout << endl;
   return x;
}
int main(void) {
   cout << "    Potenz ||     Legendre (Sek) |       Linear (Sek) |" << endl
        << "-----------++--------------------+--------------------+" << endl;
   for (double i = ::pow(2,20); i <= ::pow(2,30); i*=2) {
      cout << setw(10) << int(i) << " ||";
      legendre_log(1.3, int(i));
      hoch_linear(1.3, int(i));
   }
}
```

Programm 1.1 liefert z. B. die folgende Ausgabe:

```
    Potenz ||     Legendre (Sek) |       Linear (Sek) |
-----------++--------------------+--------------------+
   1048576 ||        21 (     0) |   1048576 (  0.34) |
   2097152 ||        22 (     0) |   2097152 (  0.69) |
   4194304 ||        23 (     0) |   4194304 (  1.38) |
   8388608 ||        24 (     0) |   8388608 (  2.77) |
  16777216 ||        25 (     0) |  16777216 (  5.54) |
  33554432 ||        26 (     0) |  33554432 ( 11.07) |
  67108864 ||        27 (     0) |  67108864 (  22.8) |
 134217728 ||        28 (     0) | 134217728 ( 45.38) |
 268435456 ||        29 (     0) | 268435456 ( 88.81) |
 536870912 ||        30 (     0) | 536870912 (177.51) |
1073741824 ||        31 (     0) |1073741824 (355.06) |
```

Beispiel zu n-logarithmischen und quadratischen Algorithmus $O(n \log n)$ und $O(n^2)$

Programm 1.2 demonstriert den Unterschied zwischen einem n-logarithmischen und einem quadratischen Algorithmus, indem es aus einem immer größer werdenden Bereich einmal die Primzahlen nach dem *Sieb des Eratosthenes* (Komplexität $O(n^2)$) und dann mit einem effizienteren Algorithmus (Komplexität $O(n \log n)$) ermittelt.

Programm 1.2 – `algoquadlog.cpp`:
Primzahlen nach dem Sieb des Eratosthenes und effizienter

```
#include <ctime>
#include <cmath>
#include <iomanip>
#include <iostream>
using namespace std;

#define MAX 100000

#define AUSGABE(x) \
   cout << setw(12) << x << " (" << setw(6) \
        << (((double)clock()-start) / CLOCKS_PER_SEC) << ") |";

void eratosthenes(int a[], int n) {
   unsigned  z=0;
   clock_t   start = clock();

   for (int i=2; i < n ; i++)
      for (int j=2 ; j<n ; j++, z++)
         if (j >= 2*i && j%i == 0)
         a[j] = 0;
   AUSGABE(z)
   //... Ausgabe der Primzahlen auskommentiert
   // for (int z=2; z < n ; z++)
   //    if (a[z])
   //       cout << setw(7) << z;
}
```

```
void prim_nlogn(int a[], int n) {
   int     prim[MAX], i, c=1, p, k, m, z=0;
   clock_t start = clock();

   // cout << setw(7) <<  2;
   m = n/2 - 3;
   for (i=0; i<=m; i++)
      prim[i] = 1;
   for (i=0; i<=m; i++) {
      if (z++,prim[i]) {
         p = i+i+3;
         k = i+p;
         c++;
         // cout << setw(7) <<  p;
         while (z++,k<=m) {
            prim[k] = 0;
            k += p;
         }
      }
   }
   AUSGABE(z) cout << endl;
}
int main(void) {
  int array[MAX], i;
   cout << "    Anzahl   ||   Eratosthenes (Sek) |        n log n (Sek) |" << endl
        << "-------------++----------------------+----------------------+" << endl;
   for (double s = ::pow(2,10); s <= ::pow(2,16); s*=2) {
      cout << setprecision(2) << setw(12) << int(s) << " ||";
      for (i=1; i<=s ; i++)
         array[i] = i;
      eratosthenes(array, int(s));
      for (i=1; i<=s ; i++)
         array[i] = i;
      prim_nlogn(array, int(s));
   }
}
```

Programm 1.2 liefert z. B. die folgende Ausgabe:

```
    Anzahl   ||   Eratosthenes (Sek) |        n log n (Sek) |
-------------++----------------------+----------------------+
        1024 ||     1044484 (     0) |        1399 (     0) |
        2048 ||     4186116 (  0.03) |        2895 (     0) |
        4096 ||    16760836 (  0.11) |        5965 (     0) |
        8192 ||    67076100 (  0.42) |       12248 (     0) |
       16384 ||   268369924 (   1.7) |       25085 (     0) |
       32768 ||  1073610756 (   6.6) |       51275 (     0) |
       65536 ||  4294705156 (    27) |      104611 (     0) |
```

Es handelt sich bei dem Algorithmus in der Funktion `prim_nlogn()` um einen n-logarithmischen Algorithmus, da sein Zeitaufwand etwas größer als linear ist.

Kapitel 2

Strings

In C ist ein String (Zeichenkette) ein Array von Zeichen, dessen Ende durch ein `0`-Byte gekennzeichnet ist. In C++ wird über die Klasse `string` ein eigener Datentyp angeboten, der sich in zwei wesentlichen Punkten von den 0-terminierten `char*`-Strings in C unterscheidet:

1. Die Klasse `string` stellt nützliche Methoden zur Verfügung, um die jeweilige Zeichenkette bearbeiten zu können. Zudem wird in einem `string`-Objekt interne Information über die gespeicherte Zeichenkette gehalten, wie z. B. deren Inhalt, deren Startadresse, deren aktuelle und maximal mögliche Länge. Wird die maximal mögliche Länge überschritten, wird der Puffer, in dem sich die Zeichenkette befindet, automatisch verlängert.
2. Zeichenketten in `string`-Objekten werden nicht mit `0`-Byte abgeschlossen.

Damit wird dem C++-Programmierer einige Verantwortung abgenommen, so dass einige der häufigsten Fehler beim Programmieren in C vermieden werden können:

- Kein Überschreiben fremden Speicherplatzes außerhalb des reservierten Puffers mehr.
- Kein fehlerhafter Zugriff über nicht initialisierte oder falsche Zeiger mehr.
- Keine ungültige Zeiger mehr, nachdem der Speicherplatz für die Zeichenkette freigegeben wurde.

Bei Verwendung der `string`-Klasse muss man folgende Headerdatei inkludieren:

```
#include <string>
```

2.1 Vordefinierte Konstanten und Datentypen für Strings

Nachfolgend werden Operationen vorgestellt, die auf `string`-Objekte durchgeführt werden können. Dabei gilt Folgendes:

- `npos` (eigentlich `string::npos`) ist als `-1` definiert.
- Der Datentyp `size_type` entspricht dem Datentyp `size_t` (entspricht meist `int` oder `long`).

2.2 Erzeugen und Initialisieren von Strings

Die Klasse `string` stellt die folgenden Konstruktoren zur Verfügung:

`string()` $O(1)$
: legt ein `string`-Objekt mit einem leeren String an.

`string obj(const string& s)` $O(n)$

`string obj = s` $O(n)$
: legen beide ein `string`-Objekt an, das mit `string`-Objekt `s` initialisiert wird.

`string(const char *str)` $O(n)$

`string = str` $O(n)$
: legen beide ein `string`-Objekt an, das mit dem String `str` initialisiert wird.

`string(const char *str, size_type n)` $O(n)$
: legen ein `string`-Objekt an, das mit den ersten `n` Zeichen des Strings `str` initialisiert wird.

`string(const string& s, size_type pos, size_type n=npos)` $O(n)$
: legt ein `string`-Objekt an, das mit einem Teilstring aus `string`-Objekt `s` initialisiert wird. Der Teilstring startet dabei ab dem Zeichen an der Position `pos` und hat `n` Zeichen. Der Defaultwert `npos` legt hier fest, dass der Teilstring die restlichen Zeichens des Strings ab Position `pos` enthält.

`string(size_type n, const char& z)` $O(n)$
: legt ein `string`-Objekt mit einem String an, der `n` Mal das Zeichen `z` enthält.

`string(InputIterator begin, InputIterator end)` $O(n)$
: initialisiert das Objekt mit den Zeichen, die sich im Bereich [`begin`,`end`) befinden.

Programm 2.1 – `stringinit.cpp`:
Initialisieren von string-Objekten

```
#include  <string>
#include  <iostream>
using namespace std;

int main(void) {
   string leer;  // Ein Leerstring der Länge 0 ("")

   string str1("Ich bin ein String");       // Übergabe C-Zeichenkette
   string str2 = "Ich bin auch ein String"; // Übergabe C-Zeichenkette
   cout << "str1 : " << str1 << endl << "str2 : " << str2 << endl;
   string str3(str1);    // Übergabe eines anderen string-Objekts
   string str4 = str2;   // Übergabe eines anderen string-Objekts
   cout << "str3 : " << str3 << endl << "str4 : " << str4 << endl;
   string str5(20, '-'); // String mit 20 Strichen anlegen
   string str6("I am a String", 10); // Ersten 10 Zeichen übernehmen
```

```
   cout << "str5 : " << str5 << endl << "str6 : " << str6 << endl;
   string str7(str1, 4, 11); // Von str1 ab 4. Pos. 11 Zeichen übernehmen
   string str8(str2, 8);     // Von str2 ab 8. Pos. alle Zeichen übernehmen
   cout << "str7 : " << str7 << endl << "str8 : " << str8 << endl;
   string str9 (str1.begin(), str1.end()); // ganzen str1 übernehmen
   string str10(str1.begin()+4, str1.end()-7); // Teilstring von str1 übernehmen
   cout << "str9 : " << str9 << endl << "str10: " << str10 << endl;
}
```

Programm 2.1 liefert die folgende Ausgabe:

```
str1 : Ich bin ein String
str2 : Ich bin auch ein String
str3 : Ich bin ein String
str4 : Ich bin auch ein String
str5 : --------------------
str6 : I am a Str
str7 : bin ein Str
str8 : auch ein String
str9 : Ich bin ein String
str10: bin ein
```

2.3 Zuweisungen an Strings

`string`-Objekte können einander mit dem Zuweisungsoperator = zugewiesen werden, wobei dieser wie folgt überladen ist:

```
string& operator=(const string& s)
string& operator=(const char *str)
string& operator=(char z)
```

Auf der rechten Seite kann dabei also ein anderes `string`-Objekt, eine C-Zeichenkette oder aber auch nur ein einfaches Zeichen angegeben werden, wie es in Programm 2.2 gezeigt ist.

Programm 2.2 – `stringzuw.cpp`:
Zuweisen von string-Objekten

```
#include  <string>
using namespace std;


int main(void) {
   string str1("Ich bin String1");
   string str2, str3;
   string str4;
   // string str5 = 'z'; // Initialisierung mit einem Zeichen nicht mögl.

   str2 = str1;            // Zuweisen von str1 an str2
   str3 = "I am String3";  // Zuweisen einer C-Zeichenkette an str3
   str4 = 'z';  // Zuweisen eines Zeichens an str4
}
```

Aus Programm 2.2 wird ersichtlich, dass man einem `string`-Objekt zwar ein Zeichen zuweisen kann, aber man es nicht mit einem Zeichen initialisieren kann.
Neben der Zuweisung mittels des Zuweisungsoperators `=` ist auch eine Zuweisung mittels der Methode `assign()` möglich:

`string& assign(const string& s)`
`string& assign(const char *str)`
: weisen das `string`-Objekt `s` bzw. den C-String `str` zu.

`string& assign(const char *str, size_type n)`
: weist die ersten `n` Zeichen des Strings `str` zu.

`string& assign(const string& s, size_type p, size_type n)`
: weist `n` Zeichen ab der Position `p` des Strings aus `string`-Objekt `s` zu.

`string& assign(size_type n, const char& z)`
: weist einen String zu, der `n` Mal das Zeichen `z` enthält.

`string& assign(InputIterator begin, InputIterator end)`
: weist die Zeichen zu, die sich im Bereich von [`begin`,`end`) befinden.

Programm 2.3 – `stringzuw2.cpp`:
Zuweisen mittels der Methode assign()

```
#include <iostream>
#include <string>
using namespace std;
int main(void) {
  string str1;
  string str2("Hase und Igel");
   str1.assign("Katz und Maus");              cout << str1 << endl;
   str1.assign(str2);                         cout << str1 << endl;
   str1.assign("Katz und Maus", 8);           cout << str1 << endl;
   str1.assign(str2, 5, 3);                   cout << str1 << endl;
   str1.assign("Katz und Maus", 5, 3);        cout << str1 << endl;
   str1.assign(30, 'X');                      cout << str1 << endl;
   str1.assign(str2.begin(), str2.end());     cout << str1 << endl;
   str1.assign(str2.begin()+5, str2.end()-2); cout << str1 << endl;
}
```

Programm 2.3 liefert die folgende Ausgabe:

```
Katz und Maus
Hase und Igel
Katz und
und
und
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Hase und Igel
und Ig
```

2.4 Ermitteln der Länge eines Strings

```
size_type length() const
size_type size() const
```

Programm 2.4 – `stringlen.cpp`:
Länge von Strings ermitteln

```
#include <iostream>
#include <string>
using namespace std;
int main(void) {
  string string1("Ich bin ein String");
  string string2;
   cout << string1 << ": " << string1.size()   << endl;
   cout << string1 << ": " << string1.length() << endl;
   cout << string2 << ": " << string2.size()   << endl;
   cout << string2 << ": " << string2.length() << endl;
}
```

Programm 2.4 liefert die folgende Ausgabe:

```
Ich bin ein String: 18
Ich bin ein String: 18
: 0
: 0
```

2.5 Prüfen, ob String leer ist

```
bool empty() const
```

Programm 2.5 – `stringempty.cpp`:
Prüfen, ob String leer ist

```
#include <iostream>
#include <string>
using namespace std;
int main(void) {
  string string1("Ich bin ein String");
  string string2;
   if (!string1.empty())    cout << "string1 ist nicht leer" << endl;
   if (string1.size() != 0) cout << "string1 ist nicht leer" << endl;
   if (string2.empty())       cout << "string2 ist leer" << endl;
   if (string2.length() == 0) cout << "string2 ist leer" << endl;
}
```

Programm 2.5 liefert die folgende Ausgabe:

```
string1 ist nicht leer
string1 ist nicht leer
string2 ist leer
string2 ist leer
```

2.6 Konvertieren von string-Objekte in C-Zeichenketten

In Programm 2.2 wurde bei der zweiten Zuweisung die C-Zeichenkette implizit in ein `string`-Objekt konvertiert. Die umgekehrte Konvertierung von einem `string`-Objekt in eine C-Zeichenkette wird nicht implizit durchgeführt.
Hier muss man eine der folgenden Methoden explizit aufrufen:

`const char *c_str()`
`const char *data()`
: liefern beide einen Zeiger auf einen normalen C-String, der bei `c_str()` 0-terminiert ist. Da der zurückgegebene Zeiger auf `const` zeigt, können die Zeichen in diesem C-String nicht geändert werden.

`size_type copy(char *str, size_type n, size_type pos=0) const`
: kopiert `n` Zeichen des Strings ab Position `pos` nach `str` und liefert die Anzahl der kopierten Zeichen als Rückgabewert. Um alle Zeichen eines Strings zu kopieren, kann man `string::npos` als zweites Argument angeben. Sollte `n` größer als die Zahl der noch vorhandenen Zeichen sein, werden die restlichen Zeichen des Strings kopiert. Wichtig ist, dass die kopierte Zeichenkette nicht mit `0` terminiert wird; siehe auch Programm 2.6.

Programm 2.6 – `stringkonv.cpp`:
Konvertieren von string-Objekten in C-Zeichenketten

```
#include  <string>
#include  <iostream>
using namespace std;

int main(void) {
   string str("Ich bin ein String");

   // const char *cstring = str; // nicht möglich
   const char *cstring1 = str.c_str();             cout << cstring1 << endl;
   const char *cstring2 = str.data();              cout << cstring2 << endl;
   cout << "-------------------------" << endl;
   char *cstring3 = new char[str.length()+1];
   strcpy(cstring3+10, "xxxxx");                   cout << cstring3 << endl;
   str.copy(cstring3, 10);                         cout << cstring3 << endl;
   cstring3[str.copy(cstring3, 10)] = 0;           cout << cstring3 << endl;
   cstring3[str.copy(cstring3, string::npos)] = 0; cout << cstring3 << endl;
   cstring3[str.copy(cstring3, 7, 4)] = 0;         cout << cstring3 << endl;
   cout << "-------------------------" << endl;
   char zeichk[30];
   memset( zeichk, '\0', 30 ); // Speicher vorher ausnullen
   str.copy( zeichk, string::npos );
   cout << zeichk << endl;
}
```

Programm 2.6 liefert die folgende Ausgabe:

```
Ich bin ein String
Ich bin ein String
-------------------------

Ich bin eixxxxx
Ich bin ei
Ich bin ein String
bin ein
-------------------------
Ich bin ein String
```

2.7 Zugriff auf einzelne Zeichen

Auf einzelne Zeichen in einem `string`-Objekt kann lesend bzw. schreibend auf zwei Arten zugegriffen werden:

- mittels des Indexoperators `[]`: Hier wird keine Überprüfung durchgeführt, ob der angegebene Index innerhalb des erlaubten Bereichs liegt.
- mittels der `string`-Methode `at()`, die eine Referenz liefert: Sollte sich der angegebene Index nicht innerhalb des erlaubten Bereichs befinden, wird hier die Exception `out_of_range` ausgelöst, die man abfangen kann.

Es ist noch zu erwähnen, dass bei `string`-Objekten nicht wie in C mittels Zeiger-Dereferenzierung (Operator `*`) auf einzelne Zeichen zugegriffen werden kann.

Programm 2.7 – `stringzugr.cpp`:
Zugriff auf einzelne Zeichen in string-Objekten

```
#include <iostream>
#include <string>
using namespace std;

int main(void) {
   string str("Ein Text");
   str[7] = 'l';     // jetzt: "Ein Texl"

   if (str[5] == 'e')
      str[5] = 's';  // jetzt: "Ein Tsxl"
   // *str = 'x';    // Nicht möglich, da operator* nicht für string def.
   str.at(4) = str.at(0);  // jetzt: "Ein Esxl"
   if (str.at(6) == 'x')
      str.at(6) = 'e';  // jetzt: "Ein Esel"
   cout << str << endl;

   // str[8] = 'c'; // keine Überprüfung
   // str.at(8) = 'c'; // Überprüfung und Programmabbruch durch Exception
}
```

Programm 2.7 liefert die folgende Ausgabe:

```
Ein Esel
```

Programm 2.8 – `stringzugr2.cpp`:
Abfangen eines unerlaubten Zugriffs bei at()

```
#include <iostream>
#include <stdexcept>
#include <string>
using namespace std;

int main(void) {
  string str("Ein Text");
   try {
      str[100] = 'c'; // keine Exception
   } catch (out_of_range) { cerr << "Unerlaubter Zugriff bei str[100]" << endl; }
   try {
      str.at(100) = 'c'; // löst Exception aus
   } catch (out_of_range) { cerr << "Unerlaubter Zugriff bei at(100)"  << endl; }
   cout << ".... Programmende" << endl;
}
```

Programm 2.8 liefert die folgende Ausgabe, an der erkennbar ist, dass bei einem unerlaubten Zugriff nur bei `at()` eine Exception geschickt wird, aber nicht beim Indexoperator:

```
Unerlaubter Zugriff bei at(100)
.... Programmende
```

2.8 Anhängen von Strings

Es gibt mehrere Möglichkeiten, einen String an einen anderen String anzuhängen:

- **Operator +=**

  `string& operator+=(const string &s)`
  `string& operator+=(const char *str)`
  `string& operator+=(char z)`
  hängen an den String `s` bzw. `str` oder das Zeichen `z` an.

- **Methode `append()`**

  `string& append(const string &s)`
  `string& append(const char *str)`
  hängen den String aus dem `string`-Objekt `s` bzw. den String `str` an.

  `string& append(const char *str, size_type n)`
  hängt die ersten `n` Zeichen des Strings `str` an.

  `string& append(const string &s, size_type pos, size_type n)`
  hängt `n` Zeichen ab Position `pos` des Strings aus `string`-Objekt `s` an.

  `string& append(size_type n, char z)`
  hängt `n` Mal das Zeichen `z` an.

  `string& append(input_iterator begin, input_iterator end)`
  hängt an String die Zeichen an, die sich in [`begin`,`end`) befinden.

- **Operator +**
  Desweiteren kann auch noch der Operator + verwendet werden, um in einem Ausdruck zwei Strings aneinander zu fügen. Dabei ist lediglich zu beachten, dass mindestens einer der beiden Operanden ein `string`-Objekt ist. Der andere Operand kann ein `string`-Objekt, ein C-String (`const char *`) oder sogar nur ein Zeichen (`char`) sein. Wenn keiner der beiden Operanden ein `string`-Objekt ist, muss man mindestens einen der beiden explizit in ein `string`-Objekt konvertieren, wie z. B.:

```
string("Heute") + " lerne ich C++";
```

Programm 2.9 – `stringapp.cpp`:
Anhängen von Strings

```
#include <iostream>
#include <string>
using namespace std;
int main(void) {
   string str1("Wald"),            str2("Wiese"), str3("Igel"),
          str4("Katze jagt die "), str5("Haus"),  str6("Waldmeister");
   str1 += " und " + str2;
    cout << str1 << endl;
   str1 = "Hase";
   str1.append(" und "); // " und " anhängen
   str1.append(str3);    // str3 anhängen
    cout << str1 << endl;
   str1.append(" rennen um die Wette", 7); // nur 7 Zeichen anhängen
    cout << str1 << endl;
   str1 = "schmausen";
   str4.append(str1, 3, 2)      ; // "ma" anhängen
   str4.append("Russland", 1, 2); // "us" anhängen
   str4.append(10, '-'); // 10 Striche anhängen
    cout << str4 << endl;
   str5.append(str6.begin()+4, str6.end()); // str6 (ab Pos. 4) anhängen
    cout << str5 << endl;
   str1 = str2 + 'l'; // ein Zeichen anhängen
    cout << str1 << endl;
   str2 += 'n'; // ein Zeichen mit += anhängen
    cout << str2 << endl;
}
```

Programm 2.9 liefert die folgende Ausgabe:

```
Wald und Wiese
Hase und Igel
Hase und Igel rennen
Katze jagt die maus----------
Hausmeister
Wiesel
Wiesen
```

2.9 Einfügen in Strings

Um in einem `string`-Objekt einzufügen, steht die Methode `insert()` zur Verfügung:

`string& insert(size_type pos, const string &s)`

`string& insert(size_type pos, const char *str)`
fügen den String aus dem `string`-Objekt `s` bzw. den String `str` an der Position `pos` ein.

`string& insert(size_type pos, const char *str, size_type n)`
fügt die ersten `n` Zeichen des Strings vom `string`-Objekt `s` an der Position `pos` ein.

`string& insert(size_type pos, const string &s, size_type pos2, size_type n)`
fügt einen Teilstring (`n` Zeichen ab Position `pos2`) vom `string`-Objekt `s` an der Position `pos` ein.

`string& insert(size_type pos, size_type n, char z)`
fügt `n` Mal das Zeichen `z` an der Position `pos` ein.

`iterator insert(iterator i, char z)`
fügt das Zeichen `z` vor der Position ein, auf die der Iterator `i` zeigt. Der Iterator `i` wird hier zurückgegeben.

`void insert(iterator i, size_type n, char z)`
fügt `n` Mal das Zeichen `z` vor der Position ein, auf die der Iterator `i` zeigt.

`void insert(iterator i, iterator begin, iterator end)`
fügt den String, der sich im Bereich [`begin`,`end`) befindet, vor der Position ein, auf die der Iterator `i` zeigt.

Programm 2.10 – `stringins.cpp`:
Einfügen von Strings

```
#include <string>
#include <iostream>
using namespace std;

int main(void) {
   string str1("Donaufahrt");
   str1.insert(5, "schiff"); // An Position 5 "schiff" einfügen
   cout << str1 << endl;
   str1.insert(5, "-Tageszeit", 6); // An Position 5 "-Tages" einfügen
   cout << str1 << endl;
   str1 = "Donaufahrt";
   string str2("schiff");
   str1.insert(5, str2); // An Position 5 "schiff" einfügen
   cout << str1 << endl;
   str2 = "-Tages-Zeitschrift";
   str1.insert(5, str2, 6, 5); // An Position 5 "-Zeit" einfügen
```

```
   cout << str1 << endl;
   string str3("Mittelstreifen");
   str3.insert(6, 10, '-'); // 10 Striche an Position 6 einfügen
   cout << str3 << endl;
   str3.insert(str3.end(), '+'); // Am Ende '+' anhängen
   cout << str3 << endl;
   str3.insert(str3.begin()+2, 5, '*'); // An 2. Pos. 5 Mal '*' einfügen
   cout << str3 << endl;
   str1 = "0123456789";
   string::iterator it = str3.begin()+20;
   str3.insert(it, str1.begin()+4, str1.end()); // str6 (ab Pos. 4) anhängen
   cout << str3 << endl;
}
```

Programm 2.10 liefert die folgende Ausgabe:

```
Donauschifffahrt
Donau-Tagesschifffahrt
Donauschifffahrt
Donau-Zeitschifffahrt
Mittel----------streifen
Mittel----------streifen+
Mi*****ttel----------streifen+
Mi*****ttel---------456789-streifen+
```

2.10 Ersetzen in Strings

Um Teile eines Strings in einem `string`-Objekt zu ersetzen, steht die Methode `replace()` zur Verfügung:

`string& replace(size_type pos, size_type n, const string &s)`
`string& replace(size_type pos, size_type n, const char *str)`
ersetzt `n` Zeichen ab Position `pos` durch den String `s` bzw. durch `str`.

`string& replace(size_type pos, size_type n, const char *str, size_type n2)`
ersetzt bis zu `n` Zeichen beginnend bei `pos` im String durch `n2` Zeichen des Strings `str`.

`string& replace(size_type pos, size_type n, const string &s, size_type pos2, size_type n2)`
ersetzt bis zu `n` Zeichen beginnend bei `pos` im String durch `n2` Zeichen ab Position `pos2` des Strings im `string`-Objekt `s`.

`string& replace(size_type pos, size_type n, size_type n2, char z)`
ersetzt bis zu `n` Zeichen beginnend bei `pos` im String mit einem String, der `n2` Mal das Zeichen `z` enthält.

`string& replace(iterator begin, iterator end, const string &str)`

`string& replace(iterator begin, iterator end, const char *str)`
ersetzt den Bereich [`begin`,`end`) im String durch den String `str`.

`string& replace(iterator begin, iterator end, const char *str, size_type n)`
ersetzt den Bereich [`begin`,`end`) im String durch die ersten `n` Zeichen von String `str`.

`string& replace(iterator begin,iterator end, size_type n,char z)`
ersetzt den Bereich [`begin`,`end`) im String durch einen String, der `n` Mal das Zeichen `z` enthält.

`string& replace(iterator begin,iterator end, input_iterator q1, input_iterator q2)`
ersetzt den Bereich [`begin`,`end`) im String durch den Bereich [`q1`,`q2`).

Programm 2.11 – `stringrepl.cpp`:
Ersetzen in string-Objekten

```
#include <string>
#include <iostream>
using namespace std;

int main(void) {
   string str1("Donauschifffahrt");
   str1.replace(5, 6, "fluss"); // Donauschifffahrt --> Donauflussfahrt
   cout << str1 << endl;
   str1.replace(0, 5, "Amazonas-Gebiet", 8); // Donauflussfahrt
   cout << str1 << endl;                     //   --> Amazonasflussfahrt
   str1 = "Donauschifffahrt";
   string str2("fluss");
   str1.replace(5, 6, str2); // Donauschifffahrt --> Donauflussfahrt
   cout << str1 << endl;
   str2 = "Regenwald im Amazonas-Gebiet";
   str1.replace(0, 5, str2, 13, 8); // Donauflussfahrt-->Amazonasflussfahrt
   cout << str1 << endl;
   string str3 = "Waldblumen", str4 = "Riesenbaum";
   str3.replace(str3.begin()+1, str3.begin()+4, str4.begin()+1, str4.end()-4);
   cout << str3 << endl;
}
```

Programm 2.11 liefert die folgende Ausgabe:

```
Donauflussfahrt
Amazonasflussfahrt
Donauflussfahrt
Amazonasflussfahrt
Wiesenblumen
```

2.11 Vertauschen von Strings

Um die Inhalte zweier `string`-Objekte zu vertauschen, steht die folgende Methode zur Verfügung:

`void& string::swap(const string& s)` $O(1)$

Programm 2.12 – `stringswap.cpp`:
Vertauschen des Inhalts von zwei string-Objekten

```
#include <string>
#include <iostream>
using namespace std;

int main(void)
{
   string string1("Eins");
   string string2("Zwei");

   cout << " Davor: " << string1 << ", " << string2 << endl;
   string1.swap(string2);
   cout << "Danach: " << string1 << ", " << string2 << endl;
}
```

Programm 2.12 liefert die folgende Ausgabe:

```
 Davor: Eins, Zwei
Danach: Zwei, Eins
```

2.12 Vergleichen von Strings

Es gibt mehrere Möglichkeiten, einen String mit einem anderen zu vergleichen:

- **Operatoren ==, !=, <, <=, >, >=**
  Mit diesen Operatoren können zwei `string`-Objekte oder ein `string`-Objekt mit einem C-String verglichen werden.
- **Methode `compare()`**
  `compare()` informiert über den Rückgabewert, wie das `string`-Objekt lexikographisch zum anderen `string`-Objekt einzuordnen ist:
  - negativer Wert: vor anderem
  - positiver Wert: nach anderem
  - Wert 0: Beide `string`-Objekte sind gleich

  Zur Methode `compare()` existieren mehrere Varianten:

  `int compare(const string &s) const`
  `int compare(const char *str) const`
  vergleicht den String mit dem String `s` bzw. `str`.

  `int compare(size_type pos,size_type n,const string &s) const`
  `int compare(size_type pos,size_type n,const char *str) const`
  vergleicht im String `n` Zeichen ab Position `pos` mit dem String `s` bzw. `str`.

`int compare(size_type pos, size_type n, const string &s, size_type pos2, size_type n2) const`

vergleicht im String `n` Zeichen ab Position `pos` mit `n2` Zeichen ab Position `pos2` des Strings `s`.

`int compare(size_type pos, size_type n, const char *str, size_type n2) const`

vergleicht im String `n` Zeichen ab Position `pos` mit den ersten `n2` Zeichen des Strings `str`.

Programm 2.13 – `stringcomp1.cpp`: Vergleichen von string-Objekten

```
#include <iostream>
#include <string>
using namespace std;

int main(void)
{
   string string1("Anton");
   string string2("Tirol");
   string string3("Tirol");

   if (string1 != string2) cout << "string1 ungleich string2" << endl;
   if (string2 == string3) cout << "string2 gleich string3"   << endl;

   if (string1.compare(string2) > 0)
       cout << "string1 nach string2 im Alphabet" << endl;
   else if (string1.compare(string2) < 0)
       cout << "string1 vor string2 im Alphabet" << endl;
   else
       cout << "Beide Strings sind gleich" << endl;

   //.............................................. Auch möglich
   if (string2 > string1)
       cout << "string2 nach string1 im Alphabet" << endl;
   else if (string2 < string1)
       cout << "string2 vor string1 im Alphabet" << endl;
   else
       cout << "Beide Strings sind gleich" << endl;
}
```

Programm 2.13 liefert die folgende Ausgabe:

```
string1 ungleich string2
string2 gleich string3
string1 vor string2 im Alphabet
string2 nach string1 im Alphabet
```

Programm 2.14 – `stringcomp2.cpp`:
Sortieren mittels der string-Vergleichsoperatoren

```
#include <iostream>
#include <string>
using namespace std;
int main(void) {
   int    i, j;
   string name[] = { "Zorro", "Martin", "Alfons", "Hansi", "Hans" };
   for (i=0; i < 4 ; i++)
      for (j=i+1; j < 5 ; j++)
         if (name[i] > name[j])
            name[i].swap(name[j]);
   for (i=0; i < 5 ; i++)
      cout << name[i] << ", ";
}
```

Programm 2.14 liefert die folgende Ausgabe:

```
Alfons, Hans, Hansi, Martin, Zorro,
```

Programm 2.15 – `stringcomp3.cpp`:
Vergleichen von Teilstrings

```
#include <iostream>
#include <string>
using namespace std;
int main(void) {
   string string1("Hallo Hansilein");
   if (!string1.compare(6, 4, "Hans")) // ab 6. Pos. 4 Zeichen vergl.
      cout << "Hans gefunden" << endl;
   if (!string1.compare(6, 100, "Hansilein")) // Zahl grösser als Zeichen
      cout << "Hansilein gefunden" << endl;
     // ab 6. Pos. 4 Zeichen vergleichen (bei längeren Vergleichsstring)
   if (!string1.compare(6, 4, "Hans ging allein"))
      cout << "Hans gefunden" << endl;
   else
      cout << "Hans nicht gefunden" << endl;
     // ab 6. Pos. 4 Zeichen vergleichen,
     // aber im Vergleichsstring ab 8. Pos. 4 Zeichen
   if (!string1.compare(6, 4, "Kleiner Hans ging allein", 8, 4))
      cout << "Hans gefunden" << endl;
   else
      cout << "Hans nicht gefunden" << endl;
}
```

Programm 2.15 liefert die folgende Ausgabe:

```
Hans gefunden
Hansilein gefunden
Hans nicht gefunden
Hans gefunden
```

2.13 Löschen in Strings

Löschen von Teilen eines Strings mit erase()

`string& erase(size_type pos=0, size_type n=npos)` $O(n)$
löscht im String `n` Zeichen ab Position `pos` und liefert `*this` zurück.

`iterator erase(iterator begin, iterator end)` $O(n)$
löscht die Zeichen im Bereich `[begin,end)` und liefert einen Iterator, der hinter das letzte gelöschte Zeichen zeigt.

`iterator erase(iterator i)` $O(n)$
löscht das Zeichen, auf das der Iterator `i` zeigt und liefert einen Iterator, der hinter das letzte gelöschte Zeichen zeigt.

Löschen aller Zeichen eines Strings mit clear()

`void clear()` $O(n)$
entspricht dem Aufruf `erase(s.begin(), s.end())`.

Programm 2.16 – `stringerase.cpp`:
Löschen in Strings

```
#include  <iostream>
#include  <string>
using namespace std;
int main(void)
{
   string str1("Das ist das Haus vom Nikolaus");
   string str2(str1);

   str1.erase(8, 4);                            cout << str1 << endl;
   str1.erase(str1.begin()+4, str1.begin()+8);  cout << str1 << endl;
   str1.erase(str1.begin()+5);                  cout << str1 << endl;
   str1.erase(str1.begin(), str1.end());  cout << "" << str1 << "" << endl;

   cout << str2 << endl;
   str2.clear();                          cout << "" << str2 << "" << endl;
}
```

Programm 2.16 liefert die folgende Ausgabe:

```
Das ist Haus vom Nikolaus
Das Haus vom Nikolaus
Das Hus vom Nikolaus
""
Das ist das Haus vom Nikolaus
""
```

2.14 Extrahieren von Teilstrings

`string substr(size_type pos=0, size_type n=npos) const`
liefert zum String einen Teilstring, der an `pos` beginnt und `n` Zeichen umfasst. Wird kein `n` angegeben, enthält der zurückgelieferte Teilstring alle restlichen Zeichen ab Position `pos`. Wir auch kein `pos` angegeben, wird eine Kopie des vollständigen Strings geliefert.

Programm 2.17 – `stringsubstr.cpp`: Extrahieren von Teilstrings

```
#include  <iostream>
#include  <string>
using namespace std;
int main(void) {
   string str1("Das ist das Haus vom Nikolaus"), str2;
   str2 = str1.substr(0, 4) + str1.substr(12);
   cout << str2 << endl;
   cout << str2.substr(13) << endl;
   cout << str1.substr() << endl;
}
```

Programm 2.17 liefert die folgende Ausgabe:

```
Das Haus vom Nikolaus
Nikolaus
Das ist das Haus vom Nikolaus
```

2.15 Suchen in Strings

2.15.1 Vorwärtssuche nach einem String

Um in Strings vorwärts nach einem String zu suchen, steht die Methode `find()` zur Verfügung. Findet sie den gesuchten String, liefert sie die Position des ersten Zeichens, wo der zu suchende String sich befindet. Wird der zu suchende String nicht gefunden, gibt sie `string::npos` zurück.
Folgende Varianten stehen zur Methode `find()` zur Verfügung:

`size_type find(const string& s, size_type pos=0) const`
`size_type find(const char *str, size_type pos=0) const`
suchen im String ab Position `pos` nach dem String des `string`-Objekts `s` bzw. nach dem String `str`.

`size_type`
`find(const char*str, size_type pos=0,size_type n=npos) const`
sucht im String ab Position `pos` nach dem String `str`, wobei die Suche jedoch nur auf `n` Zeichen begrenzt ist.

`size_type find(char z, size_type pos=0) const`
sucht im String ab Position `pos` nach dem Zeichen `z`.

Programm 2.18 – `stringfind.cpp`:
Vorwärtssuchen in Strings

```
#include  <iostream>
#include  <string>
using namespace std;
int main(void) {
   string::size_type  pos;
   string str1("eins zwei drei vier"), str2("drei"), str3;
   if ( (pos = str1.find("zwei")) != string::npos)
      cout << "\"zwei\" an Position " << pos << " gefunden" << endl;
   if ( str1.find("two") == string::npos)
      cout << "\"two\" nicht gefunden" << endl;
   str3 = str1.substr(0, str1.find(str2)) + string("---") +
          str1.substr(str1.find(str2) + str2.length());
   cout << str3 << endl;
}
```

Programm 2.18 liefert die folgende Ausgabe:

```
"zwei" an Position 5 gefunden
"two" nicht gefunden
eins zwei --- vier
```

Programm 2.19 ist ein weiteres Demonstrationsbeispiel zur Methode `find()`. Es erwartet auf der Kommandozeile als erstes Argument einen String, der durch den String zu ersetzen ist, der als zweites Argument angegeben ist. Es liest dann Text von Standardeingabe und gibt diesen Text mit den entsprechenden Ersetzung wieder auf der Standardausgabe aus.

Programm 2.19 – `stringfind2.cpp`:
Ersetzen eines Strings in einem Text

```
#include <string>
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {
   if (argc != 3) {
      cerr << "usage: " << argv[0] << "suchstring ersetzstring" << endl; exit(1);
   }
   string zeile;
   string such(argv[1]);
   string ersetz(argv[2]);
   while (getline(cin, zeile)) {
      while (true) {
         string::size_type pos = zeile.find(such);
         if (pos == string::npos)
            break;
         zeile.replace(pos, such.size(), ersetz);
      }
      cout << zeile << endl;
   }
}
```

Hat man z. B. die folgende Datei `brief.txt`:

```
Lieber Herr Meier,

Sollten Sie, Herr Meier, den vereinbarten Termin nicht einhalten
koennen, so lassen Sie uns das rechtzeitig wissen, damit wir uns
auf diese Verzoegerung einstellen koennen.

Mit freundlichen Gruessen und einen schoenen Tag noch, Herr Meier
-- Die Projektleitung
```

und man ruft diese Programm wie folgt auf:
./stringfind2 Meier Leutheusser < brief.txt
so liefert es die folgende Ausgabe:

```
Lieber Herr Leutheusser,

Sollten Sie, Herr Leutheusser, den vereinbarten Termin nicht einhalten
koennen, so lassen Sie uns das rechtzeitig wissen, damit wir uns
auf diese Verzoegerung einstellen koennen.

Mit freundlichen Gruessen und einen schoenen Tag noch, Herr Leutheusser
-- Die Projektleitung
```

2.15.2 Rückwärtssuche nach einem String

Um in Strings rückwärts nach einem String zu suchen, steht die Methode `rfind()` zur Verfügung. Findet sie den gesuchten String, liefert sie die Position des ersten Zeichens, wo der zu suchende String sich befindet. Wird der zu suchende String nicht gefunden, gibt sie `string::npos` zurück.
Folgende Varianten stehen zur Methode `rfind()` zur Verfügung:

`size_type rfind(const string& s, size_type pos=npos) const`
`size_type rfind(const char *str, size_type pos=npos) const`
: suchen rückwärts im String ab Position `pos` nach dem String des `string`-Objekts `s` bzw. nach dem String `str`.

`size_type`
`rfind(const char *str, size_type pos, size_type n) const`
: sucht rückwärts im String ab Position `pos` nach dem String `str`, wobei die Suche jedoch nur auf `n` Zeichen begrenzt ist.

`size_type rfind(char z, size_type pos=npos) const`
: sucht rückwärts im String ab Position `pos` nach dem Zeichen `z`.

Programm 2.20 – `stringrfind.cpp`:
Rückwärtssuchen in Strings

```
#include  <iostream>
#include  <string>
using namespace std;
```

```
int main(void) {
   string::size_type  pos;
   string str1("eins zwei drei vier zwei"), str2("drei"), str3;

   if ( (pos = str1.rfind("zwei")) != string::npos)
      cout << "z̈wei¨ an Position " << pos << " gefunden" << endl;

   if ( str1.rfind("two") == string::npos)
      cout << "ẗwo¨ nicht gefunden" << endl;

      str3 = str1.substr(0, str1.rfind(str2)) + string("---") +
          str1.substr(str1.rfind(str2) + str2.length());
   cout << str3 << endl;
}
```

Programm 2.20 liefert die folgende Ausgabe:

```
"zwei" an Position 20 gefunden
"two" nicht gefunden
eins zwei --- vier zwei
```

2.15.3 Vorwärtssuche nach Zeichen aus einer Zeichenmenge

Um in Strings vorwärts nach einem Zeichen aus einer vorgegebenen Zeichenmenge zu suchen, stehen die beiden Methoden `find_first_of()` und `find_first_not_of()` zur Verfügung.

find_first_of() – Suchen nach Zeichen aus einer Zeichenmenge

Diese Methode sucht vorwärts nach einem Zeichen aus einer vorgegebenen Zeichenmenge. Findet sie ein solches Zeichen, gibt sie dessen Position zurück. Wird kein solches Zeichen gefunden, gibt sie `string::npos` zurück.
Folgende Varianten stehen zur Methode `find_first_of()` zur Verfügung:

`size_type find_first_of(const string& s, size_type pos=0) const`
`size_type find_first_of(const char *str, size_type pos=0) const`
: suchen vorwärts im String ab Position `pos` nach einem Zeichen aus der Zeichenmenge, die über den String des `string`-Objekts `s` bzw. über den String `str` festgelegt ist.

`size_type find_first_of(const char *str, size_type pos, size_type n) const`
: sucht vorwärts im String ab Position `pos` nach einem Zeichen aus dem String `str`, wobei zur Suche jedoch nur `n` Zeichen aus `str` herangezogen werden.

`size_type find_first_of(char z, size_type pos=0) const`
: sucht vorwärts im String ab Position `pos` nach dem Zeichen `z`.

find_first_not_of() – Suchen nach Zeichen, das nicht in Zeichenmenge

Diese Methode sucht vorwärts nach einem Zeichen, das sich nicht in einer vorgegebenen Zeichenmenge befindet. Findet sie ein solches Zeichen, gibt sie dessen Position zurück. Wird kein solches Zeichen gefunden, gibt sie `string::npos` zurück.
Folgende Varianten stehen zur Methode `find_first_not_of()` zur Verfügung:

```
size_type find_first_not_of(const string&s, size_type p=0) const
size_type find_first_not_of(const char*str, size_type p=0) const
```

suchen vorwärts im String ab Position `p` nach einem Zeichen, das sich nicht in der Zeichenmenge befindet, die über den String des `string`-Objekts `s` bzw. über den String `str` festgelegt ist.

```
size_type find_first_not_of(const char *str, size_type pos,
                            size_type n) const
```

sucht vorwärts im String ab Position `pos` nach einem Zeichen, das sich nicht im String `str` befindet, wobei zur Suche jedoch nur `n` Zeichen aus `str` herangezogen werden.

```
size_type find_first_not_of(char z, size_type pos=0) const
```

sucht vorwärts im String ab Position `pos` nach einem Zeichen, das verschieden von `z` ist.

2.15.4 Rückwärtssuche nach Zeichen aus einer Zeichenmenge

Um in Strings rückwärts nach einem Zeichen aus einer vorgegebenen Zeichenmenge zu suchen, stehen die beiden Methoden `find_last_of()` und `find_last_not_of()` zur Verfügung.

find_last_of() – Rückwärtiges Suchen nach Zeichen aus Zeichenmenge

Diese Methode sucht rückwärts nach einem Zeichen aus einer vorgegebenen Zeichenmenge. Findet sie ein solches Zeichen, gibt sie die Position dieses Zeichens zurück. Wird kein solches Zeichen gefunden, gibt sie `string::npos` zurück.
Folgende Varianten stehen zur Methode `find_last_of()` zur Verfügung:

```
size_type find_last_of(const string& s, size_type p=npos) const
size_type find_last_of(const char *str, size_type p=npos) const
```

suchen rückwärts im String ab Position `p` nach einem Zeichen aus der Zeichenmenge, die über den String des `string`-Objekts `s` bzw. über den String `str` festgelegt ist.

```
size_type find_last_of(const char *str, size_type pos,
                       size_type n) const
```

sucht rückwärts im String ab Position `pos` nach einem Zeichen aus dem String `str`, wobei zur Suche jedoch nur `n` Zeichen aus `str` herangezogen werden.

```
size_type find_last_of(char z, size_type pos=npos) const
```

sucht rückwärts im String ab Position `pos` nach dem Zeichen `z`.

find_last_not_of() – Rückwärts Suchen nach Zeichen, das nicht in Zeichenmenge

Diese Methode sucht rückwärts nach einem Zeichen, das sich nicht in einer vorgegebenen Zeichenmenge befindet. Findet sie ein solches Zeichen, gibt sie dessen Position zurück. Wird kein solches Zeichen gefunden, gibt sie `string::npos` zurück.
Folgende Varianten stehen zur Methode `find_last_not_of()` zur Verfügung:

`size_type find_last_not_of(const string&s,size_type p=npos)const`
`size_type find_last_not_of(const char*str,size_type p=npos)const`
suchen rückwärts im String ab Position `p` nach einem Zeichen, das sich nicht in der Zeichenmenge befindet, die über den String des `string`-Objekts `s` bzw. über den String `str` festgelegt ist.

`size_type find_last_not_of(const char *str, size_type pos, size_type n) const`
sucht rückwärts im String ab Position `pos` nach einem Zeichen, das sich nicht im String `str` befindet, wobei zur Suche jedoch nur `n` Zeichen aus `str` herangezogen werden.

`size_type find_last_not_of(char z, size_type pos=npos) const`
sucht rückwärts im String ab Position `pos` nach einem Zeichen, das verschieden von `z` ist.

Programm 2.21 – `stringfirstlast.cpp`: Suchen nach Zeichen aus einer Zeichenmenge

```
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

int main(void) {
   string zeile("Das ist das Haus vom Nikolaus"), vokale("aeiouAEIOU");
   string::size_type pos;
   if ( (pos = zeile.find_first_of("aeiouAEIOU")) != string::npos) {
      cout << zeile << endl;
      cout << setw(pos+1) << "*" << endl;
   }
   if ( (pos = zeile.find_last_of(vokale)) != string::npos) {
      cout << zeile << endl;
      cout << setw(pos+1) << "*" << endl;
   }
   if ( (pos = zeile.find_first_not_of("Ddasit ")) != string::npos) {
      cout << zeile << endl;
      cout << setw(pos+1) << "*" << endl;
   }
   if ( (pos = zeile.find_last_not_of(vokale+"kls")) != string::npos) {
      cout << zeile << endl;
      cout << setw(pos+1) << "*" << endl;
   }
}
```

Programm 2.21 liefert die folgende Ausgabe:

```
Das ist das Haus vom Nikolaus
 *
Das ist das Haus vom Nikolaus
                           *
Das ist das Haus vom Nikolaus
            *
Das ist das Haus vom Nikolaus
                     *
```

Programm 2.22 – `wortstatist.cpp`:
Suchen nach Zeichen aus einer Zeichenmenge

```
#include <string>
#include <iostream>
#include <iomanip>
using namespace std;
#define MAX 1000
//..................... liefert zu einem String kleingeschriebene Kopie
string toLower(string& s) {
  char *str = new char[s.length()];
  s.copy(str, s.length());
  for (unsigned i = 0; i < s.length(); i++)
     str[i] = tolower(str[i]);
  string s2(str, s.length());
  delete str;
  return s2;
}
int main(void) {
   string zeile,
          wort[MAX],
          wortZeichen("abcdefghijklmnopqrstuvwxyzäöü"
                      "ABCDEFGHIJKLMNOPQRSTUVWXYZÄÖÜ");
   string::size_type start, ende;
   int   i, j, n=-1;
   //................................. Einlesen und Wörter extrahieren
   while (getline(cin, zeile)) {
      start = ende = 0;
      while (true) {
         start = zeile.find_first_of(wortZeichen, ende);
         if (start == string::npos)
            break;
         ende = zeile.find_first_not_of(wortZeichen, start);
         if (++n >= MAX) {
            cerr << "Maximale Wortzahl erreicht" << endl;
            break;
         }
         wort[n] = zeile.substr(start, ende-start);
      }
   }
```

```
   //........................................... Sortieren der Wörter
   for (i=0; i<n; i++)
      for (j=i+1; j<=n; j++)
         if (toLower(wort[i]) > toLower(wort[j]))
            wort[i].swap(wort[j]);

   //............... Ausgeben der Wörter (gleiche hintereinander zählen)
   i=0;
   while (i <= n) {
      int z = 1;
      while (++i <= n && wort[i] == wort[i-1])
         z++;
      cout << setw(7) << z << " " << wort[i-1] << endl;
   }
}
```

Wendet man dieses Programm auf die eigene Quelldatei `wortstatist.cpp` an:
./wortstatist < wortstatist.cpp
liefert es die folgende Ausgabe:

```
      1 ABCDEFGHIJKLMNOPQRSTUVWXYZÄÖÜ
      1 abcdefghijklmnopqrstuvwxyzäöü
      1 Ausgeben
      2 break
      1 cerr
      2 char
      1 cin
      1 copy
      ............
      ............
      9 wort
      1 Wortzahl
      3 wortZeichen
      3 Wörter
      3 z
      5 zeile
      1 zu
      1 zählen
```

2.16 Speicherplatz-Größe von Strings

Wird z. B. ein String verkleinert, so bleibt der einmal reservierte Speicherplatz erhalten, so dass man zwei Längen-Eigenschaften für einen String hat:

- Anzahl der wirklich belegten Zeichen, die sich mit `length()` bzw. `size()` erfragen lässt.
- Größe des intern reservierten Speicherplatzes

Bezüglich des intern reservierten Speicherplatzes für einen String in einem `string`-Objekt stehen die folgenden Methoden zur Verfügung:

`size_type capacity() const`
`size_type max_size() const`
: liefern beide die maximale Anzahl der Zeichen, die der reservierte Speicherplatz für den String aufnehmen kann, bevor dieser Speicherplatz (automatisch) vergrößert werden muss. Diese Zahl ist mindestens so groß wie die Zahl, die `length()` bzw. `size()` liefern.

`void resize(size_type n)`
`void resize(size_type n, char z)`
: ändert explizit die aktuelle Größe eines Strings auf `n` Zeichen, wobei es bei der zweiten Variante den neu hinzugekommenen Speicherplatz im Falle einer Vergrößerung mit dem Zeichen `z` füllt.

`void reserve(size_type n)`
: setzt explizit die maximale Anzahl (Kapazität) des String-Speicherplatzes auf `n` Zeichen. Dies ist jedoch nur ein Vorschlag, da intern auch mehr Speicherplatz aus Performanz-Gründen bereitgestellt werden kann.

Programm 2.23 – `stringcapa.cpp`: Speicherplatz-Größen von Strings

```
#include <string>
#include <iostream>
using namespace std;

int main(void)
{
  string str1(4, 'x');
  cout << str1 << " (" << str1.size() << "," << str1.capacity() << ")" << ", ";
  str1.insert(2, "YYYY");
  cout << str1 << " (" << str1.size() << "," << str1.capacity() << ")" << ", ";

  str1.reserve(100); // intern reservierten Speicherplatz auf 100 vergrö"sern
  str1.append("ZZZZZZ");
  cout << str1 << " (" << str1.size() << "," << str1.capacity() << ")" << endl;

  str1.resize(10); // Anzahl der Zeichen auf 10 reduzieren
  cout << str1 << " (" << str1.size() << "," << str1.capacity() << ")" << endl;

  str1.resize(20, '-'); // String auf 20 mit Strich als Füllzeichen vergrö"sern
  cout << str1 << " (" << str1.size() << "," << str1.capacity() << ")" << endl;
}
```

Programm 2.23 liefert die folgende Ausgabe:

```
xxxx (4,4), xxYYYYxx (8,8), xxYYYYxxZZZZZZ (14,100)
xxYYYYxxZZ (10,100)
xxYYYYxxZZ---------- (20,100)
```

2.17 Iterator-Methoden für Strings

Vorwärts-Iteratoren

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

gibt einen Iterator auf das erste Zeichen des jeweiligen Strings zurück. Im Falle von konstanten `string`-Objekten ist die zweite Variante zu verwenden.

`iterator end() const`
`const_iterator end() const`

gibt einen Iterator zurück, der hinter das letzte Zeichen des jeweiligen Strings zeigt. Im Falle von konstanten `string`-Objekten ist die zweite Variante zu verwenden.

Programm 2.24 – `stringiter1.cpp`:
String-Vergleich ohne Unterscheidung von Groß-/Kleinschreibung

```
#include <string>
#include <iostream>
using namespace std;
//.............................................................. stringIComp
int stringIComp(const string& s1, const string& s2) {
   string::const_iterator it1 = s1.begin(), // Iteratoren auf Anfang der Strings
                          it2 = s2.begin();
   while(it1 != s1.end() && it2 != s2.end()) { // Strings mit Iteratoren durchl.
      if (tolower(*it1) != tolower(*it2))
         return (tolower(*it1) < tolower(*it2)) ? -1 : 1;
      it1++;
      it2++;
   }
   return(s1.size() - s2.size());
}
int main(void) {
  char  *zeich[] = { " < ", " == ", " > " };
   string s1("Hans"), s2("HAMMER");
     cout << s1 << zeich[stringIComp(s1, s2)+1] << s2 << endl;
   s1 = "Hammer"; s2 = "HANS";
     cout << s1 << zeich[stringIComp(s1, s2)+1] << s2 << endl;
   s1 = "HANS";  s2 = "Hansi";
     cout << s1 << zeich[stringIComp(s1, s2)+1] << s2 << endl;
   s1 = "HANS";   s2 = "hans";
     cout << s1 << zeich[stringIComp(s1, s2)+1] << s2 << endl;
}
```

Programm 2.24 liefert die folgende Ausgabe:

```
Hans > HAMMER
Hammer < HANS
HANS < Hansi
HANS == hans
```

Rückwärts-Iteratoren

`reverse_iterator rbegin() const`
`const_reverse_iterator rbegin() const`
: gibt einen Rückwärts-Iterator auf das letzte Zeichen des jeweiligen Strings zurück. Im Falle von konstanten `string`-Objekten ist die zweite Variante zu verwenden.

`reverse_iterator rend() const`
`const_reverse_iterator rend() const`
: gibt einen Rückwärts-Iterator zurück der vor das erste Zeichen des jeweiligen Strings zeigt. Im Falle von konstanten `string`-Objekten ist die zweite Variante zu verwenden.

Programm 2.25 – `stringiter2.cpp`:
Demonstrationsprogramm zu Rückärts-Iteratoren

```
#include <string>
#include <iostream>
using namespace std;
int main(void) {
   string s("abcdefghi");
   string::reverse_iterator rit; // Rückwärts-Iterator
   // Inkrementieren eines Rückwärts-Iterators --> auf vorheriges Zeichen
   for (rit = s.rbegin(); rit != s.rend(); rit++)
      cout << *rit << ", ";
   cout << endl;
   string sRueck(s.rbegin(), s.rend());
   for (rit = sRueck.rbegin(); rit != sRueck.rend(); rit++)
      cout << *rit << ", ";
}
```

Programm 2.25 liefert die folgende Ausgabe:

```
i, h, g, f, e, d, c, b, a,
a, b, c, d, e, f, g, h, i,
```

2.18 Anhängen von Zeichen mit push_back()

Mit der Funktion `push_back()` können einzelne Zeichen an einen String angehängt werden:

`void push_back(char z)`

Programm 2.26 – `stringpushback.cpp`:
Demonstrationsprogramm zu push_back() bei Strings

```
#include <iostream>
#include <string>
using namespace std;
int main(void)
{
   string str;
```

```
   for (int i=0; i < 16; i++)
      str.push_back( (i < 10) ? i+'0' : i-10+'a' );
   cout << "Die Hexadezimalziffern sind: " << endl << "  ";
   cout << str << endl;
}
```

Programm 2.26 liefert die folgende Ausgabe:

```
Die Hexadezimalziffern sind:
  0123456789abcdef
```

2.19 Einlesen von Zeilen in Strings mit getline()

Die Klasse `string` definiert die globale Funktion `getline()`, um Strings einzulesen:

`istream& getline(istream& is, string& s, char delimiter = '\n')`
: Die Funktion `getline()`, die nicht Teil der Klasse `string` ist, liest eine Zeile von `is` und speichert diese in `s`. Ist ein `delimiter` angegeben, beendet `getline()` das Lesen, wenn es ein solches Zeichen liest.

Programm 2.27 – `stringgetline.cpp`:
Demonstrationsprogramm zu getline() bei Strings

```
#include <iostream>
#include <string>
using namespace std;
int main(void) {
  string plzOrt, name;
   cout << "PLZ Wohnort: ";
   getline( cin, plzOrt );
   cout << "Dein Name: ";
   cin >> name;
   cout << "Wohnort: "             << plzOrt << endl
        << "Name (nur ein Wort): " << name   << endl;
}
```

Programm 2.27 liefert die folgende Ausgabe:

```
PLZ Wohnort: 47011 Musterstadt ⏎
Dein Name: Emil Kastner ⏎
Wohnort: 47011 Musterstadt
Name (nur ein Wort): Emil
```

2.20 Übungen

2.20.1 Das Focaultsche Pendel

Im Roman „Das Focaultsche Pendel“ von *Umberto Eco* taucht der folgende verschlüsselte Text auf:

```
Kuabris Defrabax Rexulon Wkkazaal Wkzaah Wrpaefel Uaculbain Habrak Hacoruin
Maquafel Uebrain Hmcatuin Rokasor Himesor Argaabil Kaquaan Docrabax Reisaz
Reisabrax Decaiquan Oiquaquil Zaitabor Quaxaop Dugraq Yaelobran Disaeda
Magisuan Raitak Huidal Wscolda Arabaom Zipreus Mecrim Cosmae Duquifas Rocarbis
```

Der Entschlüsselungs-Algorithmus für diesen Text ist, dass nur die Anfangsbuchstaben der ansonsten sinnlosen Wörter relevant sind. Für die extrahierten Anfangsbuchstaben ist dann ein Verschiebechiffre um eine Position anzuwenden. Erstellen Sie ein Programm *focault.cpp*, das solche Texte entschlüsselt. Befindet sich z. B. dieser Text in der Datei `focault.ein`, so sollte ein Aufruf wie z. B. **./focault <focault.ein** Folgendes liefern:

```
....lesxxxviinvisiblesseparezensixbandes
```

Dies ist französisch:
„`les xxxvi invisibles separez en six bandes`“ und bedeutet:
„`Die 36 Unsichtbaren geteilt in sechs Gruppen`“.

2.20.2 Ver-/Entschlüsseln eines Gedichts von Ringelnatz

Von Joachim Ringelnatz stammt das folgende verschlüsselte Gedicht:

```
Ibich habibebi dibich,
Lobittebi, sobi liebib.
Habist aubich dubi mibich
Liebib? Neibin, vebirgibib.
Nabih obidebir Febirn
Gobitt seibi dibir gubit.
Meibin Hebirz habit gebirn
Abin dibir gebirubiht.
```

In diesem Gedicht wurde in einem „normalen“ Text hinter jedem Vokal bzw. jeder Vokalgruppe ein „bi“ eingefügt. Der unverschlüsselte Text ist:

```
Ich habe dich,
Lotte, so lieb.
Hast auch du mich
Lieb? Nein, vergib.
Nah oder Fern
Gott sei dir gut.
Mein Herz hat gern
An dir geruht.
```

Erstellen Sie nun zunächst ein Programm `machebi.cpp`, das den Ausgangstext entsprechend verschlüsselt. Legt man diesem Programm den Ausgangstext, der sich in der Datei `original.ein` befindet, vor, wie z. B. **./machebi < original.ein**, so sollte es das entsprechend verschlüsselte Gedicht liefern.
Erstellen Sie auch ein Programm `entferbi.cpp`, das ein mit „bi“ verschlüsseltes Gedicht wieder entschlüsselt. Legt man ihm das verschlüsselte Gedicht vor, das sich in Datei `verschluesselt.ein` befindet: **./entferbi < verschluesselt.ein**, so sollte es den entschlüsselten Text liefern.

2.20.3 Finden von Zahlwörtern in Strings

Endreim, Kurzweil, Nachtfalter, Wohnviertel, Neunauge, Weinstein, Erdreich, Achtung, Segelflieger, Pfalzwein, Radreifen, Gehhelfer, Leinsamen. Alles zusammen macht 66.
Erstellen Sie ein Programm `wortadd.cpp`, das diese Aufgabe löst, wie z. B.:

```
             Endreim...       drei...    3
            Kurzweil...       zwei...    2
         Nachtfalter...       acht...    8
         Wohnviertel...       vier...    4
            Neunauge...       neun...    9
           Weinstein...       eins...    1
            Erdreich...       drei...    3
             Achtung...       acht...    8
        Segelflieger...        elf...   11
           Pfalzwein...       zwei...    2
           Radreifen...       drei...    3
           Gehhelfer...        elf...   11
           Leinsamen...       eins...    1
--------------------------------------------------
                                      66
```

2.20.4 Palindrome durch Addition von Kehrzahlen

Durch wiederholte Addition von Zahl und Kehrzahl lassen sich Palindrome erzeugen. Erstellen Sie ein Programm `kehrzahl.cpp`das zu einem Zahlenbereich, der einzugeben ist, solche Palindrome erzeugt, wie z. B.

```
Palindrome durch Addition von Kehrzahlen
========================================
Ab welcher Zahl: 158 ⏎
Bis zu welcher Zahl: 167 ⏎
158+851--> 1009+9001--> 10010+01001 = 11011
159+951--> 1110+0111 = 1221
160+061--> 221+122 = 343
161+161--> 322+223 = 545
162+261--> 423+324 = 747
163+361--> 524+425 = 949
164+461--> 625+526--> 1151+1511 = 2662
165+561--> 726+627--> 1353+3531 = 4884
166+661--> 827+728--> 1555+5551--> 7106+6017--> 13123+32131 = 45254
167+761--> 928+829--> 1757+7571--> 9328+8239--> 17567+76571--> 94138+83149-->
           177287+782771--> 960058+850069--> 1810127+7210181-->
           9020308+8030209--> 17050517+71505071 = 88555588
```

Kapitel 3

Die I/O-Stream-Bibliothek

3.1 Die I/O-Stream-Klassenhierarchie

Die neue I/O-Stream-Bibliothek von C++ bietet in etwa die gleiche Funktionalität wie die auf dem Datentyp `FILE` basierende Ein- und Ausgabe in C: Öffnen von Dateien, Lesen und Scheiben in Dateien usw. Diese Ein- und Ausgabe mittels eines `FILE`-Zeigers ist auch weiterhin in C++ möglich.
Mit der neu hinzugekommenen I/O-Stream-Bibliothek ist nun aber eine auf Klassen basierende Ein- und Ausgabe möglich. Diese neue I/O-Stream-Bibliothek bringt einige Vorteile mit sich, wie z. B. automatische Erkennung der ein- bzw. auszugebenden Datentypen oder Erweiterungsmöglichkeiten für den Benutzer.

3.1.1 Grundsätzliches zu den I/O-Streamklassen

Die meisten I/O-Streamklassen von C++ beginnen mit dem Präfix `basic_`. Zu den meisten Klassen mit diesem Präfix sind aber zusätzlich noch entsprechende Namen ohne dieses Präfix definiert, wie z. B.:

```
typedef basic_ios<char>     ios;     // für char I/O
typedef basic_istream<char> istream; //  ....
typedef basic_fstream<char> fstream; //  ....
....
```

Somit muss der Programmierer das Präfix `basic_` nicht mehr angeben. Da C++ aber neben dem Standard-8-Byte `char`-Typ auch den internationalen *wide character*-Typ `wchar_t` unterstützt, wurde die I/O-Bibliothek unter Verwendung des Template-Mechanismus so entworfen, dass sie auch Ein- und Ausgaben von *wide character*-Daten ermöglicht:

```
typedef basic_ios<wchar_t>     wios;     // für wchar_t I/O
typedef basic_istream<wchar_t> wistream; //  ....
typedef basic_fstream<wchar_t> wfstream; //  ....
....
```

Hier werden nur `char`-Varianten vorgestellt, da für die `wchar_t`-Klassen weitgehend das Gleiche gilt.

3.1.2 Standard-Objekte

In der I/O-Stream-Bibliothek befinden sich bereits einige Objekte für die Ein- und Ausgabe auf die Standardstreams:

- `cin`, `cout`, `cerr` und `clog` (für `char`)
- `wcin`, `wcout`, `wcerr` und `wclog` (für `wchar_t`)

Diese Objekte sind in der Headerdatei `iostream` deklariert.

3.1.3 Die I/O-Stream-Klassenhierarchie

Abbildung 3.1 zeigt die zentrale I/O-Stream-Klassenhierarchie.

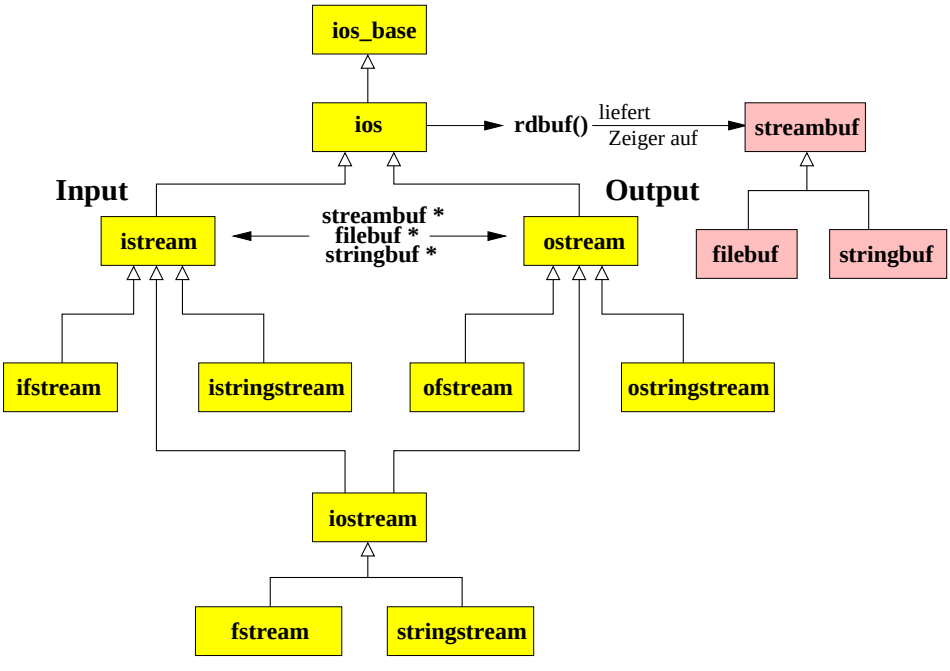


Abbildung 3.1: Zentrale I/O-Stream-Klassenhierarchie

Die einzelnen Streamklassen bieten dabei folgende Funktionalität an:

- **`ios_base` – Grundfunktionalität für alle Streamklassen**
  Diese Grundfunktionalität, die an alle anderen Streamklassen vererbt wird, umfasst z. B. Formatierungsmöglichkeiten oder Methoden, um die Zustände von I/O-Streams zu prüfen.
- **`ios` – Erweiterung der `ios_base`-Grundfunktionalität**
  Der hauptsächliche Zweck der Klasse `ios` ist, die Funktionalität der Basisklasse `ios_base` zu übernehmen, entsprechend zu erweitern und dann an die abgeleiteten Klassen weiterzuvererben.
- **`ostream` – Grundfunktionalität für Ausgaben**
  Die Klasse `ostream` stellt die grundlegende Funktionalität für Ausgaben zur Verfügung. Die Objekte `cout`, `clog` und `cerr` sind z. B. alle `ostream`-Objekte. Werden

diese Objekte in einem Programm verwendet, muss man die Headerdatei `<iostream>` inkludieren. Die beiden folgenden Klassen sind von `ostream` abgeleitet, und verfügen somit über die gleichen Fähigkeiten wie diese. Jedoch erweitern sie diese Funktionalität, indem sie für spezielle Ausgaben ausgelegt sind:

- **`ofstream` – für das Schreiben in Dateien**
- **`ostringstream` – für das Schreiben in Strings**

- **`istream` – Grundfunktionalität für Eingaben**
  Die Klasse `istream` stellt die grundlegende Funktionalität für Eingaben zur Verfügung. Das Objekt `cin` ist z. B. ein `istream`-Objekt. Wird dieses Objekt in einem Programm verwendet, muss man die Headerdatei `<iostream>` inkludieren. Die beiden folgenden Klassen sind von `istream` abgeleitet, und verfügen somit über die gleichen Fähigkeiten wie diese. Jedoch erweitern sie diese Funktionalität, indem sie für spezielle Eingaben ausgelegt sind:
  - **`ifstream` – für das Lesen aus Dateien**
  - **`istringstream` – für das Lesen aus Strings**

- **`streambuf` – Hilfsklasse für Ein/Ausgabe auf entsprechendes Gerät**
  - Die I/O-Streamklassen führen selbst keinerlei Ein- oder Ausgabe durch, sondern delegieren dies immer an die Hilfsklasse `streambuf` oder an von ihr abgeleiteten Klassen, wie z. B. `filebuf` bzw. `stringbuf`.
    Die I/O-Streamklassen sind somit lediglich die Schnittstelle zwischen den ein- bzw. auszugebenden Objekten und der Klasse `streambuf`, die für die eigentliche Ein- bzw. Ausgabe auf das entsprechende Gerät, für welches ein `streambuf`-Objekt angelegt wurde, verantwortlich ist.
  - Da die Klasse `streambuf` viele `protected` virtuelle Methoden besitzt, ist es möglich, von ihr eine eigene Klasse abzuleiten, in der diese virtuellen Methoden überschrieben werden. Objekte dieser eigenen Klasse können dann automatisch mit der von den I/O-Streamklassen angebotenen Funktionalität entsprechend aufbereitet ein- bzw. ausgegeben werden. Darauf wird in Kapitel 3.8 auf Seite 94 näher eingegangen.

3.2 Die Headerdateien der I/O-Stream-Bibliothek

Abbildung 3.2 zeigt die wesentlichen Headerdateien, welche die betreffenden I/O-Stream-Klassen beinhalten.
Die I/O-Stream-Bibliothek und ihre Klassenhierarchie ist – wie in Abbildung 3.2 erkennbar – auf verschiedene Headerdateien aufgeteilt:

- Die Headerdateien `<ios>`, `<istream>`, `<ostream>` und `<streambuf>` werden selten in C++-Programmen inkludiert. Sie stellen lediglich die Basisklassen der Hierarchie zur Verfügung und werden automatisch durch die anderen Headerdateien der Bibliothek, die die abgeleiteten Klassen enthalten, inkludiert.
- Die Headerdatei `<iostream>` muss inkludiert werden, wenn die globalen Standard-Streamobjekte `cin`, `cout`, `cerr` oder `clog` benutzt werden.

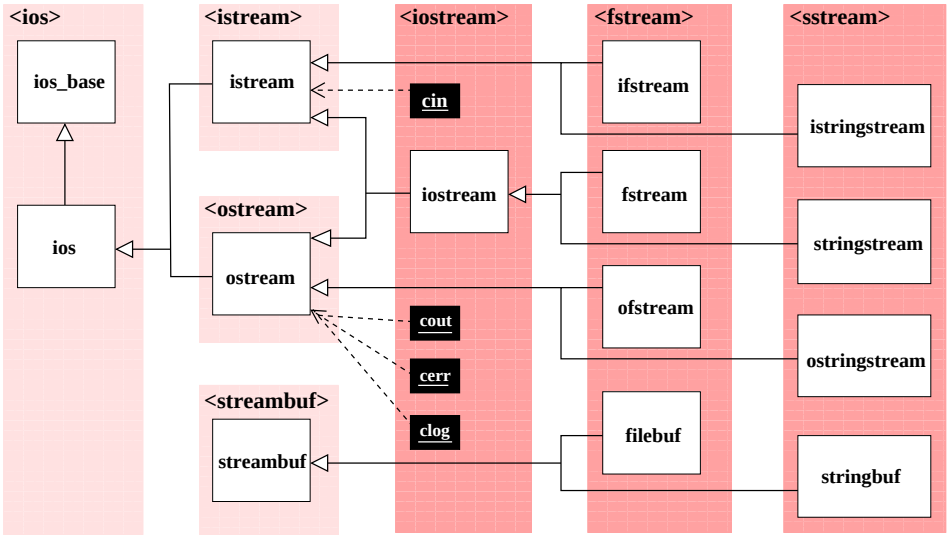


Abbildung 3.2: Headerdateien zu den I/O-Stream-Klassen

- Die Headerdatei `<fstream>` definiert die Datei-Streamklassen (wie z. B. das Template `basic_ifstream` oder die Klasse `ofstream`). Diese Headerdatei muss inkludiert werden, wenn Dateioperationen mit der I/O-Stream-Bibliothek durchgeführt werden.
- Die Headerdatei `<sstream>` definiert String-Streamklassen und muss inkludiert werden, wenn man mit STL-Strings so arbeitet, als ob es Streams wären.

Neben den in Abbildung 3.2 enthaltenen Headerdateien, existieren noch die beiden folgenden Headerdateien, die ebenfalls für die I/O-Stream-Bibliothek zur Verfügung gestellt werden:

- Die Headerdatei `<iomanip>` muss inkludiert werden, wenn man Manipulatoren mit Parametern benutzt.
- Die Headerdatei `<iosfwd>` muss inkludiert werden, wenn eine Forward-Deklaration für eine I/O-Streamklasse erforderlich ist. Statt dem folgenden Code:

```
class ostream;  // fehlerhafte Forward-Deklaration --> Compilerfehler

void funktion(ostream &str);
```

muss der folgende Code angegeben werden:

```
#include <iosfwd> // Richtige Deklaration der Klasse ostream

void funktion(ostream &str);
```

3.3 Die Basisklassen ios_base und ios

- Die Klasse `ios_base` stellt unter anderem Konstrukte zur Verfügung, die es ermöglichen, die Zustände von I/O-Streams zu prüfen oder Ausgaben zu formatieren. Da der Standardkonstruktor der Klasse `ios_base` dem Schutztyp `protected` zugeordnet ist, können keine `ios_base`-Objekte außerhalb der Klassenhierarchie angelegt werden. Ebenso können `ios_base`-Objekte nicht kopiert werden, da sowohl der Kopierkonstruktor als auch der Zuweisungsoperator `private` deklariert sind.
- Jede Streamklasse der I/O-Bibliothek ist über die Klasse `ios` von `ios_base` abgeleitet. Es können zwar Objekte von der Klasse `ios` angelegt werden, aber das wird in der Praxis wohl niemals der Fall sein. Der Zweck der Klasse `ios` ist es vielmehr, die Funktionalität von `ios_base` zu übernehmen, entsprechend zu erweitern und dann an abgeleitete Klassen weiterzuvererben.

Da alle Streamklassen sich von der Klasse `ios` und folglich auch von der Klasse `ios_base` ableiten und damit die Funktionalität dieser beiden Klassen erben, wird hier nicht detailliert auf die Klasse `ios_base` eingegegangen, da dies in Praxis normalerweise nicht von Wichtigkeit ist. Statt dessen wird die Funktionalität beider Klassen hier bei der Vorstellung der Klasse `ios` zusammengefasst. Interessierte Leser seien hier auf die Headerdateien `ios_base.h` und `basic_ios.h` verwiesen.

3.3.1 Die Methode rdbuf() der Klasse ios

Eine wichtige Methode der Klasse `ios` ist die Methode `rdbuf()`, die alle von `ios` abgeleiteten Klassen erben:

`streambuf *ios::rdbuf() const`

liefert einen Zeiger auf das `streambuf`-Objekt, das die Schnittstelle zwischen dem `ios`-Objekt und dem Gerät ist, mit dem das `ios`-Objekt kommuniziert.

Programm 3.1 kann auf zwei Arten aufgerufen werden: Gibt man den Namen einer Datei beim Aufruf an, wird deren Inhalt auf die Standardausgabe ausgeben. Gibt man zwei Dateinamen beim Aufruf an, wird der Inhalt der ersten Datei in die zweite Datei kopiert.

Programm 3.1 – `rdbuf1.cpp`: Demonstrationsprogramm zu rdbuf()

```
#include <fstream>
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {
   ifstream ein(argv[1]);
   if (argc > 2) {
      ofstream aus(argv[2]); // Datei argv[1] nach argv[2] kopieren
      aus << ein.rdbuf();
   } else
      cout << ein.rdbuf(); // Datei argv[1] auf Standardausgabe ausgeben
}
```

`streambuf *ios::rdbuf(streambuf *neu)`
kann benutzt werden, um einem `ios`-Objekt ein anderes `streambuf`-Objekt zuzuordnen. Als Rückgabewert liefert diese Funktion einen Zeiger auf das ursprüngliche `streambuf`-Objekt.

Programm 3.2, das die Ausgabe von `cout` in die Datei `test.txt` umlenkt, ist ein Demonstrationsbeispiel zu dieser Variante der Methode `rdbuf()`. Es schreibt einen String über `cout` in eine Datei.

Programm 3.2 – `rdbuf2.cpp`: Demonstrationsprogramm zu rdbuf()

```
#include <iostream>
#include <fstream>
using namespace std;

int main (void) {
   streambuf *psbuf;    // Streampuffer
   ofstream   ausDatei; // Output-Stream

   ausDatei.open("test.txt"); // Output-Stream auf Datei test.txt setzen
   psbuf = ausDatei.rdbuf();  // psbuf = Zeiger auf das entspr. streambuf-Obj.
   cout.rdbuf(psbuf); // cout von Bildschirm auf ofstream 'ausDatei' umleiten

   cout << "Dieser Text wird in die Datei test.txt geschrieben" << endl;

   ausDatei.close(); // Schliessen des Ouput-Streams ausDatei
}
```

3.3.2 Zustandsflags und -methoden der Klasse ios

Operationen auf Streams können erfolgreich sein oder aber auch fehlschlagen. Mit den in der Klasse `ios` vordefinierten Zustandsflags und entsprechenden -methoden kann man zum einen erfragen, ob ein Zugriff erfolgreich war oder nicht. Außerdem ermöglichen diese auch das Zurücksetzen der Zustandsflags.
Tabelle 3.1 zeigt die Zustandsflags, die die Klasse `ios` zur Verfügung stellt.

Tabelle 3.1: Zustandsflags in der Klasse `ios`

Flag	wird gesetzt
`ios::eofbit`	beim Lesen von *end-of-file*.
`ios::failbit`	bei nicht erfolgreicher, aber korrigierbarer Operation
`ios::badbit`	bei fataler unerlaubter Operation
`ios::goodbit`	vor dem Stattfinden von Operationen und wird erst gelöscht, wenn eines der drei vorherigen Flags gesetzt wird.

Bei einem aufgetretenen Fehler werden nachfolgende Eingaben solange nicht mehr ausgeführt, d. h. bleiben wirkungslos, bis wieder mit der Methode `clear()` die Fehlerflags gelöscht und `ios::goodbit` gesetzt wird.
Die drei Fehlerbits entsprechen unterschiedlichen Fehlerniveaus:

- `ios::eofbit` wird gesetzt, wenn das Dateieende erreicht ist. Das ist normalerweise noch kein Fehler, bedeutet aber, dass der nächste Leseversuch zu einem solchen führt.
- `ios::failbit` wird bei korrigierbaren Fehlern gesetzt, z. B. wenn die Eingabe nicht zum Datentyp passt oder wenn man eine bereits geöffnete Datei erneut zu öffnen versucht. Liest man über das Dateiende hinaus, wird zusätzlich zu `ios::eofbit` auch noch `ios::failbit` gesetzt.
- `ios::badbit` wird gesetzt, wenn der Ein- bzw. Ausgabepuffer unbrauchbar ist. Ein solcher Fall liegt z. B. vor, wenn man versucht, in einem nur zum Lesen geöffneten Stream zu schreiben, oder beim Schreiben in einer Datei, die sich auf einer vollen Festplatte befindet.

Die Tabellen 3.2 und 3.3 zeigen Methoden der Klasse `ios`, um einzelne Zustandsflags zu erfragen bzw. um mehrere Zustandsflags auf einmal zu erfragen bzw. zu setzen.

Tabelle 3.2: Methoden in Klasse `ios` zum Erfragen von Zustandsflags

Methode	**liefert `true`, wenn**
`bool eof() const`	`ios::eofbit` gesetzt ist.
`bool fail() const`	`ios::badbit` oder `ios::failbit` gesetzt ist.
`bool bad() const`	`ios::badbit` gesetzt ist.
`bool good() const`	wenn keines der drei Fehlerflags gesetzt ist

Tabelle 3.3: Methoden der Klasse `ios` zum Lesen bzw. Setzen von Zustandsflags

Methode	**Wirkung**
`void clear(iostate state=goodbit)`	löscht zunächst alle Zustandsflags und setzt dann die mit `state` festgelegten Zustandsflags.
`iostate rdstate() const`	liefert alle Zustandsflags zurück.
`void setstate(iostate state)`	setzt die mit `state` festgelegten Zustandsflags. entspricht: `clear(rdstate() | state)`

Programm 3.3 ist ein Demonstrationsprogramm zum Umgang mit den in der Klasse `ios` vordefinierten Zustandsflags und -methoden. Es zeigt unter anderem auch, dass man `ios`-Objekte (wie z. B. `cin`) an Stellen angeben kann, an denen ein boolescher Ausdruck erwartet wird; in diesem Fall ist dies dann die Kurzform für `cin.good()`.

Programm 3.3 – `ioszustflag.cpp`:
Demonstrationsprogramm zu den Zustandsflags und -methoden in Klasse `ios`

```
#include <iostream>
using namespace std;

void zustandsflags1(int flags) {
   if (flags & ios::badbit)  cout << "badbit, ";
   if (flags & ios::eofbit)  cout << "eofbit, ";
   if (flags & ios::failbit) cout << "failbit, ";
   if (flags & ios::goodbit) cout << "goodbit ";
   cout << endl;
}
void zustandsflags2(ios *iosObj) {
   if (iosObj->rdstate() & ios::badbit)  cout << "badbit, ";
   if (iosObj->eof())  cout << "eofbit, ";
   if (iosObj->rdstate() & ios::failbit)  cout << "failbit, ";
   if (iosObj->good()) cout << "goodbit ";
   cout << endl;
}
int main(void) {
   int x;

   cout << "1. Eingabe: ";
   cin >> x;
   zustandsflags1(cin.rdstate());
   cin.clear();

   cout << "2. Eingabe: ";
   cin >> x;
   zustandsflags2(&cin);
   cin.clear();

   cout << "3. Eingabe: ";
   cin >> x;
   if (!cin) // entspricht: if (!cin.good())
      cout << "Falsche Eingabe" << endl;

   cout << "Explizites Setzen von ios-Flags: ";
   cin.clear(ios::badbit | ios::eofbit);
   zustandsflags1(cin.rdstate());
}
```

Nachfolgend ist ein mögliches Ablaufbeispiel zu Programm 3.3 gezeigt:

```
1. Eingabe: Strg-D eofbit, failbit,
2. Eingabe: 34 ⏎
goodbit
3. Eingabe: hans ⏎
Falsche Eingabe
Explizites Setzen von ios-Flags: badbit, eofbit,
```

Programm 3.4 ist ein weiteres Demonstrationsprogramm zum Umgang mit den in der Klasse `ios` vordefinierten Zustandsflags und -funktionen.

Programm 3.4 – `ioszustflag2.cpp`:
Weiteres Demonstrationsprogramm zu den Zustandsflags und -funktionen in Klasse `ios`

```
#include <iostream>
#include <fstream>
using namespace std;
int main (void) {
   ifstream datei1; // Datei zum Lesen (ifstream)
   datei1.open("test2.txt"); // Versuch, nicht exist. Datei zum Lesen zu öffnen
   if ((datei1.rdstate() & ifstream::failbit) != 0 )
      cerr << "Kann Datei 'test2.txt' nicht öffnen" << endl;
   if (!datei1)
      cerr << "Kann Datei 'test2.txt' nicht öffnen" << endl;
   char zeile[80];
   fstream datei2;
   datei2.open("test2.txt", fstream::in); // Öffnen nicht exist. Datei zum Lesen
   if (datei2.fail()) {
      cerr << "Kann 'test2.txt' nicht zum Lesen öffnen" << endl;
      datei2.clear();
      datei2.open("test.txt", fstream::in); // exist. Datei zum Lesen öffnen
   }
   datei2.getline(zeile, 80); // Lesen einer Zeile aus Datei
   cout << zeile << ": erfolgreich aus Datei gelesen." << endl;
}
```

Existiert die Datei `test.txt`, aber nicht die Datei `test2.txt`, dann liefert Programm 3.4 die folgende Ausgabe:

```
Kann Datei 'test2.txt' nicht öffnen
Kann Datei 'test2.txt' nicht öffnen
Kann 'test2.txt' nicht zum Lesen öffnen
Dies wird in Datei test.txt geschrieben: erfolgreich aus Datei gelesen.
```

3.3.3 Formatflags und -methoden der Klasse ios

Vordefinierte Formatflags

Die Klasse `ios` enthält eine Membervariable (vom Typ `fmtflags`) für die einzelnen Formatbits. Für die Wertigkeiten der einzelnen Formatbits bietet sie vordefinierte Konstanten an, die in Tabelle 3.4 zusammengefasst sind. Die in Tabelle 3.4 gezeigten Flags können unabhängig voneinander gesetzt bzw. gelöscht werden. Es gibt drei Untergruppen, wobei in einer solchen Untergruppe jeweils nur das Setzen eines Flags sinnvoll ist. Zum Ausfiltern dieser Gruppen sind in der Klasse `ios` eigene Bitmasken-Konstanten definiert:

```
ios::basefield   = ios::dec | ios::oct | ios::hex
ios::adjustfield = ios::left | ios::right | ios:internal
ios::floatfield  = ios::scientific | ios::fixed
```

Bei Gleitpunktzahlen lässt sich das grundlegende Ausgabeformat über die Flags `scientific` oder `fixed` festlegen. Die Voreinstellung ist, dass keines der beiden Flags gesetzt ist. Dieses Format wird als *automatic* bezeichnet, was in C dem Formatzeichen `%g` bei `printf()` entspricht. In Tabelle 3.4 sind die Flags, die für `cout`, `cin`, `cerr` und `clog` voreingestellt sind, fett gedruckt. `unitbuf` ist nur bei `cerr` gesetzt. Die Voreinstellung ist, dass kein `adjustfield`-Flag gesetzt ist. In diesem Fall wird `right` angenommen.

Tabelle 3.4: Vordefinierte Formatflags in der Klasse `ios`

Flag	Bedeutung	Bitgruppenkonstante
`ios::dec`	Dezimalkonvertierung (Voreinstellung)	
`ios::oct`	Oktalkonvertierung	`ios::basefield`
`ios::hex`	Hexadezimalkonvertierung	
`ios::right`	Rechtsbündige Ausgabe (Voreinstellung)	
`ios::left`	Linksbündige Ausgabe	`ios::adjustfield`
`ios::internal`	Füllzeichen (Voreinstellung: Leerzeichen) zwischen Präfix und Zahlenwert (wie z. B. bei negativen Zahlen zwischen Minuszeichen und Wert)	
`ios::scientific`	Gleitpunktausgabe im Format 1.23456e12	
	(keine Voreinstellung)	`ios::floatfield`
`ios::fixed`	Gleitpunktausgabe im Format 123.456	
	(keine Voreinstellung)	
`ios::boolalpha`	Bei logischen Ausdrücken „`true`“ bzw. „`false`“ ausgeben bzw. als solche einlesen	
`ios::showbase`	Präfix `0` bei Oktal- und `0x` bei Hexadezimalzahlen	
`ios::showpoint`	Bei Gleitpunktzahlen Punkt immer ausgeben	
`ios::showpos`	Pluszeichen (+) bei positiven Ganzzahlen	
`ios::skipws`	„White spaces“ (Leer-/Tabulatorzeichen, Neuezeile usw.) bei Eingabe überlesen (Voreinstellung)	
`ios::unitbuf`	Nach jeder Ausgabeoperation Ausgabepuffer leeren (*flushen*); Voreinstellung ist, dass Ausgabepuffer nur bei `flush`, `endl`, `unitbuf` oder wenn er geschlossen wird, geleert („*geflusht*“) wird	
`ios::uppercase`	Großbuchstaben (bei Ausgabe von Hexadezimal- bzw. `scientific`-Zahlen)	

Methoden der Klasse ios zum Erfragen bzw. Setzen der Formatflags

Die Klasse `ios` stellt die folgenden Methoden zur Verfügung, um die in Tabelle 3.4 gezeigten Formatflags zu erfragen bzw. zu setzen:

`fmtflags flags() const`

liefert das aktuell eingestellte Bitmuster der Formatflags und kann z. B. wie folgt verwendet werden:

```
if (cout.flags() & ios::hex) {
   //... Ausgabe von ganzen Zahlen im Hexadezimalsystem
}
```

`fmtflags flags(fmtflags neuflags)`

setzt das Bitmuster der Formatflags entsprechend dem übergebenen Argument auf `neuflags` und liefert das zuvor gültige Bitmuster der Formatflags zurück.

`fmtflags setf(fmtflags flags)`

setzt zusätzlich zu den bereits gesetzten Formatflags noch die im übergebenen Argument `flags` gesetzten Flags. Andere Flags bleiben hierbei unverändert.

So legt z. B. `cout.setf(ios::showbase)` fest, dass die Basis des entsprechenden Zahlensystems bei der Ausgabe von ganzen Zahlen anzuzeigen ist: mit Präfix `0x` bei hexadezimalen und Präfix `0` bei oktalen Zahlen.

Diese Funktion liefert das zuvor gültige Bitmuster der Formatflags zurück.

Diese Methode wird üblicherweise benutzt, um die folgenden Flags zu setzen: `boolalpha`, `showbase`, `showpoint`, `showpos`, `skipws`, `unitbuf` und `uppercase`.

`fmtflags setf(fmtflags flags, fmtflags maske)`

löscht zunächst alle in `maske` gesetzten Flags und setzt dann zusätzlich zu den noch gesetzten Formatflags die in `flags` angegebenen Flags. Diese Funktion liefert das zuvor gültige Bitmuster der Formatflags zurück.

Typische Argumente für den Parameter `maske` sind:

`ios::adjustfield`, `ios::basefield` und `ios::floatfield`, wie z. B.:

- `cout.setf(ios::left, ios::adjustfield)`
  löscht zunächst alle `adjustfield`-Bits, bevor linksbündige Ausgabe festgelegt wird. Durch diesen Aufruf ist sichergestellt, dass nicht mehrere `adjustfield`-Bits gleichzeitig gesetzt sind. Dies ist zwar nicht verboten, aber die Wirkung ist undefiniert und vom jeweiligen Compiler abhängig. Ist kein Bit einer Bitgruppe gesetzt, wird die Voreinstellung für diese Bitgruppe herangezogen.
- `cout.setf(ios::hex, ios::basefield)`
  löscht zunächst alle `basefield`-Bits und legt dann fest, dass ganze Zahlen im Hexadezimalsystem auszugeben sind.
- `cout.setf(ios::fixed, ios::floatfield)`
  löscht zunächst alle `floatfield`-Bits und legt dann für die Ausgabe von Gleitpunktzahlen das Format `123.456` fest.

Mit einer Angabe wie

`cout.setf(0, ios::boolalpha)`

könnte man z. B. die eingeschaltete Ausgabe von „`true`“ bzw. „`false`“ bei logischen Ausdrücken wieder ausschalten.

`void unsetf(fmtflags flags)`

löscht die im übergebenen Argument `flags` gesetzten Flags. Andere Flags bleiben hierbei unverändert.

Das Löschen von gesetzten voreingestellten Formatflags wie z. B.

`cout.unsetf(ios::dec)`

ist nicht möglich.

`streamsize width() const`

liefert die aktuell eingestellte Weite (Mindestzahl der auszugebenden Zeichen) zurück.

`streamsize width(int zeichzahl)`
: setzt die Weite (Mindestzahl der auszugebenden Zeichen) auf `zeichzahl` und liefert die zuvor eingestellte Weite zurück. Es sind hierbei noch folgende Punkte zu beachten:

  - Ein Aufruf wie `width(0)` legt fest, dass genauso viel Zeichen auszugeben, wie der entsprechende Wert benötigt.
  - Die Weite wird nach jeder Ausgabe eines Werts wieder auf `0` gesetzt.

`char fill() const`
: liefert das aktuell eingestellte Füllzeichen für Ausgaben zurück. Das voreingestellte Füllzeichen ist das Leerzeichen.

`char fill(char neuFuellzeich)`
: setzt das Füllzeichen auf `neuFuellzeich` und liefert das zuvor eingestellte Füllzeichen zurück.

`streamsize precision() const`
: liefert die eingestellte Anzahl von Nachkommastellen, die bei Gleitpunktzahlen auszugeben sind. Voreinstellung ist 6.

`streamsize precision(streamsize neuGenau)`
: legt fest, dass Gleitpunktzahlen mit `neuGenau` Nachkommastellen auszugeben sind und liefert die zuvor eingestellte Anzahl von Nachkommastellen zurück. Es ist zu beachten, dass dieser Wert `neuGenau` bei *automatic* die Gesamtzahl der auszugebenden Stellen festlegt, während er ansonsten (bei `fixed` und `scientific`) die Anzahl der auszugebenden Nachkommastellen einstellt. Voreingestellter Wert für die Ausgabe von Gleitpunktzahlen ist `6`.

`ios& copyfmt(ios& obj)`
: kopiert alle Formatflags von `obj` in das entsprechende `ios`-Objekt.

Die folgende Tabelle zeigt entsprechende Manipulatoren (siehe auch Kapitel 3.3.4 auf Seite 60), die das Gleiche wie einige der hier vorgestellten Methoden leisten,

Methode	Maniplatoren
`setf()`, `unsetf()`	`setiosflags()` und `resetiosflags()`
`width()`	`setw()`
`fill()`	`setfill()`
`precision()`	`setprecision()`

Programm 3.5 ist ein Demonstrationsprogramm zur Formatierung der Stream-Ausgabe mit Methoden der Klasse `ios`.

Programm 3.5 – `iosfmtflag1.cpp`:
Formatierung der Stream-Ausgabe mit Methoden der Klasse `ios`

```
#include <iostream>
using namespace std;


int main(void) {
   int    n   = 1023;
   double dbl = 12.34;
   ios::fmtflags defFlags = cout.flags(); // Default-Einstellung der Formatflags
```

```
cout << "a) Ausgabe im Dezimal-, Hexa- oder Oktalformat" << endl;
cout << "==============================================" << endl;
cout.flags( (defFlags & ~ios::dec) | ios::hex);
cout << " 1. " << n << " Ausgabe hex" << endl;
cout << " 2. " << n << " ...immer noch hex" << endl; // Basis bleibt, bis...
cout.flags( (cout.flags() & ~ios::hex) | ios::oct); //... sie neu gesetzt wird
cout << " 3. " << n << " Ausgabe oct" << endl << endl;
 // Mit setf() und unsetf() geht es etwas einfacher, da man den Zu-
 // stand der nicht betroffenen Bits nicht berücksichtigen muss.
cout.unsetf(ios::oct);
cout.setf(ios::hex);
cout << " 4. " << n << " Ausgabe hex" << endl;
cout.setf(ios::oct, ios::basefield);
cout << " 5. " << n << " Ausgabe oct" << endl;
cout.unsetf(ios::basefield); // keines der Bits dec, oct, hex gesetzt
cout << " 6. " << n << " Voreingest. Ausgabe: dec" << endl << endl;
 // Ausgabe erfolgt mit führender "0" für Oktal- und "0x" für Hexazahlen
cout.setf(ios::showbase);
cout << " 7. " << n << " bei dec kein Präfix" << endl;
cout.setf(ios::oct, ios::basefield);
cout << " 8. " << n << " führende 0 bei oct" << endl;
cout.setf(ios::hex, ios::basefield);
cout << " 9. " << n << " führendes 0x bei hex" << endl;
cout.setf(ios::uppercase);
cout << "10. " << n << " jetzt Grossbuchstaben bei hex" << endl << endl;

cout << "b) Ausgabe eines positiven Vorzeichens (nur Dezimalsystem)" << endl;
cout << "==========================================================" << endl;
cout.unsetf(ios::basefield);   // zurück zu dec
cout.setf(ios::showpos);
cout << "11. " << n << "   " << dbl << " positives Vorzeichen" <<endl <<endl;

cout << "c) Verwendung von Feldbreiten und Füllzeichen" << endl;
cout << "============================================" << endl;
cout << "12. |";
cout.width(12); // width-Einstellungen gelten nur für naechste Ausgabeelement
cout << n << "| Feldbreite ist 12" << endl;
cout << "13. |";
cout << n << "|        Feldbreite 12 gilt nicht mehr" << endl << endl;
cout << "14. |";
cout.width(12);
cout << n << "| default ist rechtsbündig" << endl;
cout << "15. |";
cout.width(12);
cout.setf(ios::left);
cout << n << "| jetzt linksbündig" << endl;
cout << "16. |";
cout.width(12);
cout.setf(ios::internal, ios::adjustfield);
cout << n << "| Vorzeichen linksbündig, Zahl rechtsbündig" << endl;
```

```
   cout << "17. |";
   cout.width(12);
   cout.setf(ios::hex);
   cout << n << "| Basis linksbündig, Zahl rechtsbündig" << endl << endl;
   cout << "18. |";
   cout.width(12);
   cout.fill('.');
   cout << n << "| mit Punkt aufgefüllt" << endl;
   cout << "19. |";
   cout.width(12);
   cout.unsetf(ios::basefield);   // zurück zu dec
   cout << n << "| Punkt-Füllung gilt immer noch" << endl;
}
```

Programm 3.5 liefert die folgende Ausgabe:

```
a) Ausgabe im Dezimal-, Hexa- oder Oktalformat
==============================================
 1. 3ff Ausgabe hex
 2. 3ff ...immer noch hex
 3. 1777 Ausgabe oct

 4. 3ff Ausgabe hex
 5. 1777 Ausgabe oct
 6. 1023 Voreingest. Ausgabe: dec

 7. 1023 bei dec kein Präfix
 8. 01777 führende 0 bei oct
 9. 0x3ff führendes 0x bei hex
10. 0X3FF jetzt Grossbuchstaben bei hex

b) Ausgabe eines positiven Vorzeichens (nur Dezimalsystem)
==========================================================
11. +1023   +12.34 positives Vorzeichen

c) Verwendung von Feldbreiten und Füllzeichen
=============================================
12. |       +1023| Feldbreite ist 12
13. |+1023|        Feldbreite 12 gilt nicht mehr

14. |       +1023| default ist rechtsbündig
15. |+1023       | jetzt linksbündig
16. |+       1023| Vorzeichen linksbündig, Zahl rechtsbündig
17. |0X       3FF| Basis linksbündig, Zahl rechtsbündig

18. |0X.......3FF| mit Punkt aufgefüllt
19. |+.......1023| Punkt-Füllung gilt immer noch
```

Programm 3.6 ist ein Demonstrationsprogramm zur Formatierung der Stream-Ausgabe bei Gleitpunktzahlen.

Programm 3.6 – `iosfmtflag2.cpp`:
Formatierung der Stream-Ausgabe von Gleitpunktzahlen

```
#include <iostream>
using namespace std;
#define ausg(text) cout << text << dbl << endl;
int main(void) {
   double   dbl = 12.34;
   ios::fmtflags defFlags = cout.flags(); // Default-Einstellung der Formatflags
   cout << "dbl = 12.34 gesetzt:" << endl;      ausg("  1. automatic : ")
   cout.setf(ios::fixed);                       ausg("  2. fixed     : ")
   cout.setf(ios::scientific, ios::floatfield); ausg("  3. scientific: ")
   dbl = 12345678.12345678;
   cout << "dbl = 12345678.12345678 gesetzt:" << endl;
   cout.flags(defFlags); /* Default-Einstellung */ ausg("  4. automatic : ")
   cout.setf(ios::fixed);                          ausg("  5. fixed     : ")
   cout.setf(ios::scientific, ios::floatfield);    ausg("  6. scientific: ")
   cout << "precision vom Defaultwert 6 auf 3 heruntergesetzt:" << endl;
   cout.precision(3);
   cout.flags(defFlags); /* Default-Einstellung */ ausg("  7. automatic : ")
   cout.setf(ios::fixed);                          ausg("  8. fixed     : ")
   cout.setf(ios::scientific, ios::floatfield);    ausg("  9. scientific: ")
   dbl = 123.0;
   cout << "dbl = 123.0 und precision(6) gesetzt:" << endl;
   cout.flags(defFlags); /* Default-Einstellung */
   cout.precision(6);                           ausg(" 10. automatic : ")
   cout.setf(ios::fixed);                       ausg(" 11. fixed     : ")
   cout.setf(ios::scientific, ios::floatfield); ausg(" 12. scientific: ")
   cout << "showpoint-Flag gesetzt:" << endl;
   cout.flags(defFlags);      /* Default-Einstellung */
   cout.setf(ios::showpoint); /* aber mit Dezimalpkt */ ausg(" 13. automatic : ")
   cout.setf(ios::fixed);                               ausg(" 14. fixed     : ")
   cout.setf(ios::scientific, ios::floatfield);         ausg(" 15. scientific: ")
}
```

Programm 3.6 liefert die folgende Ausgabe:

```
dbl = 12.34 gesetzt:
  1. automatic : 12.34
  2. fixed     : 12.340000
  3. scientific: 1.234000e+01
dbl = 12345678.12345678 gesetzt:
  4. automatic : 1.23457e+07
  5. fixed     : 12345678.123457
  6. scientific: 1.234568e+07
precision vom Defaultwert 6 auf 3 heruntergesetzt:
  7. automatic : 1.23e+07
  8. fixed     : 12345678.123
  9. scientific: 1.235e+07
dbl = 123.0 und precision(6) gesetzt:
 10. automatic : 123
```

```
 11. fixed     : 123.000000
 12. scientific: 1.230000e+02
showpoint-Flag gesetzt:
 13. automatic : 123.000
 14. fixed     : 123.000000
 15. scientific: 1.230000e+02
```

3.3.4 Manipulatoren zum Setzen der Formatflags

Außer mit Methoden können die Formatflags auch mit so genannten *Manipulatoren* gesetzt werden. Ein Manipulator ist ein Objekt, das wie andere Größen auch in eine „Shift-Kette“ bei Ein- bzw. Ausgaben eingefügt werden kann. Manipulatoren werden intern durch einen Aufruf der entsprechenden Methode realisiert, haben aber den Vorteil, dass man sie direkt in einer „Shift-Kette“ einbetten kann und diese nicht immer unterbrechen muss, um durch Aufrufe von Methoden die gewünschten Formatflags zu setzen.
Verwendet man Manipulatoren, muss man `<iostream>` inkludieren:

```
#include <iostream>
```

Die Streambibliothek definiert eine Reihe von Manipulatoren. Es gibt parameterlose und solche mit Parametern. Verwendet man in seinem Programm Manipulatoren mit Parametern, so muss man zusätzlich noch `<iomanip>` inkludieren:

```
#include <iomanip> // nötig bei Verwendung von Manipulatoren mit Parametern
```

Eine mit einem Manipulator eingestellte Vorgabe gilt je nach Art des Manipulators entweder nur für das nächste Ausgabeelement oder aber bis diese Vorgabe explizit wieder durch einen anderen Manipulator ausgeschaltet wird.
Tabelle 3.5 zeigt die Manipulatoren, wobei die Bedeutung der einzelnen Flags bereits in Tabelle 3.4 auf Seite 54 näher erläutert wurde.

Programm 3.7 – `iosfmtflag3.cpp`:
Demonstrationsprogramm zu Manipulatoren

```
#include <iostream>
#include <iomanip> // fuer parametrisierte Manipulatoren benoetigt
using namespace std;
int main(void) {
  int    n   = 1023;
  double dbl = 12.34;
   cout << '|' << setw(12) << setfill('=') << n << '|'
        << hex << n << '|' << endl;
   cout << setiosflags(ios::scientific) << setprecision(3) << dbl << endl;
}
```

Programm 3.7 liefert die folgende Ausgabe:

```
|========1023|3ff|
1.234e+01
```

Tabelle 3.5: Manipulatoren

Manipulator	Wirkung	Gültigkeit
`dec`	`setf(ios::dec, ios::basefield)`	m
`hex`	`setf(ios::hex, ios::basefield)`	m
`oct`	`setf(ios::oct, ios::basefield)`	m
`right`	`setf(ios::right, ios::adjustfield)`	m
`left`	`setf(ios::left, ios::adjustfield)`	m
`internal`	`setf(ios::internal, ios::adjustfield)`	m
`scientific`	`setf(ios::scientific, ios::floatfield)`	m
`fixed`	`setf(ios::fixed, ios::floatfield)`	m
`boolalpha`	`setf(ios::boolalpha)`	m
`noboolalpha`	`unsetf(ios::boolalpha)`	m
`showbase`	`setf(ios::showbase)`	m
`noshowbase`	`unsetf(ios::showbase)`	m
`showpoint`	`setf(ios::showpoint)`	m
`noshowpoint`	`unsetf(ios::showpoint)`	m
`showpos`	`setf(ios::showpos)`	m
`noshowpos`	`unsetf(ios::showpos)`	m
`skipws`	`setf(ios::skipws)`	m
`noskipws`	`unsetf(ios::skipws)`	m
`unitbuf`	`setf(ios::unitbuf)`	m
`nounitbuf`	`unsetf(ios::unitbuf)`	m
`uppercase`	`setf(ios::uppercase)`	m
`nouppercase`	`unsetf(ios::uppercase)`	m
`endl`	schreibt Neuezeilezeichen in Ausgabepuffer und leert dann den Ausgabepuffer	
`ends`	schreibt String-Endezeichen (`\0`) in Ausgabepuffer	
`flush`	leert („*flushed*“) den Ausgabepuffer	
`ws`	überliest alle Zwischenraumzeichen (*whitespaces*: Leerzeichen, Tabzeichen usw.) im Eingabepuffer	
`setbase(int b)`	Für `b` kann `8`, `10` oder `16` angegeben werden; Alternative zu `oct`, `dez` und `hex`, wenn Basis z. B. in einer Variable vorliegt	m
`setiosflags(flags)`	ruft `setf(flags)` auf, um die entsprechenden `flags` zu setzen	m
`resetiosflags(flags)`	ruft `unsetf(flags)` auf, um die entsprechenden `flags` zu löschen	m
`setfill(char z)`	legt Zeichen fest, mit dem ein Ausgabefeld aufzufüllen ist (`fill(z)`); Voreinstellung ist Leerzeichen	m
`setprecision(int n)`	legt die Zahl der Nachkommastellen für die Ausgabe von Gleitpunktzahlen fest (`precision(n)`)	m
`setw(int n)`	legt Mindestzahl der auszugebenden Zeichen fest (`width(n)`)	**n**

Die dritte Spalte in Tabelle 3.5 zeigt an, wie lange die durch diesen Manipulator eingestellte Formatierung gültig ist:

- `n`: nur für nächstes Element gültig.
- `m`: so lange gültig, bis Formatierung mit gleichen Manipulator geändert wird.

Gegenüberstellung von Manipulatoren und Methoden

Aus Tabelle 3.5 wird ersichtlich, dass z. B. statt dem Ausdruck

```
cout.setf(ios::left, ios::adjustfield);
```

auch Folgendes angegeben werden kann:

```
cout  << resetiosflags(ios::adjustfield)  << setiosflags(ios::left);
```

Die folgende Tabelle zeigt Aufrufe von Methoden, die man durch Manipulatoren ersetzen kann, wobei als Streamobjekt der Name `s` verwendet wird.

Klasse	Manipulator	Methode
ios	`s << dec;`	`s.setf(ios::dec, ios::basefield);`
ios	`s << hex;`	`s.setf(ios::hex, ios::basefield);`
ios	`s << oct;`	`s.setf(ios::oct, ios::basefield);`
ios	`s << setiosflags(f);`	`s.setf(f);`
ios	`s << resetiosflags(f);`	`s.unsetf(f);`
ios	`s << setfill(z);`	`s.fill(z);`
ios	`s << setprecision(n);`	`s.precision(n);`
ios	`s << setw(n);`	`s.width(n);`
ostream	`s << endl;`	`s << '\n'; s.flush();`
ostream	`s << ends;`	`s << '\0';`
ostream	`s << flush;`	`s.flush();`

Einlesen ganzer char-Arrays unter Verwendung von setw()

Mittels des Manipulators `setw()` lassen sich auch `char`-Arrays einlesen, wie z. B.:

```
cin >> setw(sizeof(array)) >> array;
```

In diesem Fall wird sichergestellt, dass maximal `sizeof(array)-1` Zeichen in `array` eingelesen werden, wobei dieses Array automatisch mit `0` terminiert wird.

Programm 3.8 – `iosfmtflag4.cpp`:
Einlesen ganzer char-Arrays unter Verwendung von setw()

```
#include <iostream>
#include <iomanip> // fuer parametrisierte Manipulator setw() ben\"otigt
using namespace std;

int main(void) {
  char  plz[6];  // wegen 0-Terminierung
  char  wohnort[20];
   cin >> setw(sizeof(plz)) >> plz >> wohnort;
   cout << "PLZ: " << plz << endl;
   cout << "Ort: " << wohnort << endl;
}
```

Nachfolgend ist ein möglicher Ablauf für das Programm 3.8 gezeigt:

```
90489Nürnberg ⏎
PLZ: 90489
Ort: Nürnberg
```

3.3.5 Erstellen eigener Manipulatoren

Erstellen eigener parameterloser Manipulatoren

Parameterlose Manipulatoren sind nichts anderes als Namen von Funktionen, d. h. Zeiger auf Funktionen. Sie sind in etwa wie folgt definiert:

```
ostream& (*manipulator) (ostream& stream)
```

Parameterlose Manipulatoren sind somit Funktionen, die einen `ostream`-Referenzparameter besitzen und eine `ostream`-Referenz als Rückgabe liefern. Die Deklaration von `endl` sieht z. B. wie folgt aus:

```
ostream& endl(ostream&);
```

Ruft man nun Folgendes auf:

```
cout << "das ist ein Text" << endl;
```

wird für `endl` die Adresse dieser Funktion eingesetzt. Die Operatorfunktion $<<$ der Klasse `ostream` ist für Argumente dieser Art z. B. speziell wie folgt überladen:

```
ostream& ostream::operator<<(ostream& (*fktZgr)(ostream&)) {
  return fktZgr(*this);
}
```

Dies bedeutet, dass die korrespondierende Methode mit dem aktuellen `ostream`-Objekt als Argument aufgerufen wird. Ein Aufruf wie

```
cout << hex;
```

hat somit die gleiche Wirkung wie

```
hex(cout);
```

Programm 3.9 demonstriert, wie man sich eigene parameterlose Manipulatoren zur Verfügung stellen kann, indem man eine globale Funktion (hier `striche()`) definiert und diese dann als Manipulator einsetzt, indem man deren Namen (ohne Klammerpaar) in die Ausgabe einsetzt.

Programm 3.9 – `manip1.cpp`:
Erstellen von eigenen parameterlosen Manipulatoren

```
#include <iostream>
using namespace std;

ostream& striche(ostream &os) { return os << "-----------------\n"; }

int main(void) {
   cout << striche << "    Fehler!" << endl << striche;
   cout << "xxxxxxxxxxxxxxx" << endl;
   striche(cout);
}
```

Programm 3.9 liefert die folgende Ausgabe:

```
------------------
    Fehler!
------------------
xxxxxxxxxxxxxxx
------------------
```

Erstellen eigener Manipulatoren mit Parametern

Zum Erstellen von Manipulatoren mit Parametern kann man folgende Vorgehensweise wählen:

1. Erstellen einer eigenen Klasse mit dem Namen des Manipulators *manip*.
2. Erstellen eines Konstruktors zu dieser Klasse *manip* mit der Anzahl der gewünschten Parametern. In diesem Konstruktor kann man die gewünschten Aktionen durchführen.
3. Einbetten einer `friend`-Funktion in diese Manipulatorklasse:

```
friend ostream& operator<<(ostream& os, const manipName& m) {
   return os << m.membervar; // Alternativer Code hier auch denkbar
}
```

Ruft man nun einen solchen selbst erstellten Manipulator auf, wie z. B.:

```
cout << manip(...) << endl;
```

wird durch den Konstruktoraufruf von *manip(...)* ein temporäres Objekt erzeugt, das an `operator<<` weitergeleitet wird. Da im Konstruktor bzw. auch in der `friend`-Funktion `operator<<` die gewünschten Aktionen durchgeführt werden, kann man entsprechend eigene Manipulatoren entwickeln, wie es in Programm 3.10 gezeigt ist, das zwei eigene Manipulatoren definiert:

- `DoubleFormat` – ermöglicht C-ähnliche Formatierung für Gleitpunktzahlen
- `Bin` – ermöglicht die binäre Ausgabe einer ganzen Zahl

Programm 3.10 – `manip2.cpp`:
Erstellen von eigenen Manipulatoren mit Parametern

```
#include <string>
#include <iostream>
#include <iomanip>
using namespace std;
class DoubleFormat { //.... Eigene C-ähnliche Formatierung für Gleitpunktzahlen
  public:
     DoubleFormat(const float& z, int w, int n) {
        char   zahl[100];
        sprintf(zahl, "%*.*lf", w, n, z);
        str = zahl;
     }
     friend ostream& operator<<(ostream& os, const DoubleFormat& fmt) {
        return os << fmt.str;
     }
  private:
     string str;
```

```
};
class Bin { //................................ Binäre Ausgabe einer ganzen Zahl
  public:
     Bin(unsigned long zahl, bool trenn = false) {
        for (int i=sizeof(unsigned long)*8-1; i>=0; i--) {
           str.append(1, ((zahl >> i) & 1) + '0');
           if (trenn && i%4==0)
              str.append(" ");
        }
     }
     friend ostream& operator<<(ostream& os, const Bin& b) {
        return os << b.str;
     }
  private:
     string str;
};
int main(void) {
   cout << "|" << DoubleFormat(3.14159, 10, 2) << "|" << endl;
   cout << "|" << DoubleFormat(3.14159, 8, 5) << "|" << endl;
   cout << "|" << DoubleFormat(1234.444, 12, 0) << "|" << endl;
   cout << "|" << DoubleFormat(1234.567, 8, 0) << "|" << endl;
   cout << setw(12) << "0xAFFE1234: " << Bin(0xAFFE1234, 1) << endl;
   cout << setw(12) << "1234567: "    << Bin(1234567)       << endl;
}
```

Programm 3.10 liefert die folgende Ausgabe:

```
|      3.14|
| 3.14159|
|        1234|
|    1235|
0xAFFE1234: 1010 1111 1111 1110 0001 0010 0011 0100
   1234567: 00000000000100101101011010000111
```

3.4 Streamklassen für die Ausgabe

Für die Ausgabe bietet C++ mehrere Klassen an:

Klasse `ostream` (`#include <iostream>`)
: stellt die grundlegende Funktionalität für die Ausgabe zur Verfügung.

Klasse `ofstream` (`#include <fstream>`)
: ermöglicht das Öffnen von Dateien zum Schreiben;
vergleichbar mit `fopen(`*`dateiname`*`, "w")` in C.

Klasse `ostringstream` (`#include <sstream>`)
: ermöglicht das Schreiben von Daten in den Speicher;
vergleichbar mit `sprintf()` in C.

3.4.1 Die Klasse ostream – Basisfunktionalität für Ausgabe

Die Klasse `ostream` stellt die grundlegende Funktionalität für die Ausgabe zur Verfügung. Die Objekte `cout`, `clog` und `cerr` sind z. B. alle `ostream`-Objekte. Werden diese Objekte in einem Programm verwendet, muss man die Headerdatei `<iostream>` inkludieren. Alle im vorherigen Kapitel vorgestellten Konstrukte der Klasse `ios` sind soweit dies die Ausgabe betrifft auch in der Klasse `ostream` verfügbar, da diese Klasse alle diese Flags und die Methoden der Klasse `ios` erbt.
Folgender Konstruktor legt ein `ostream`-Objekt zu `puffer` an:

```
ostream(streambuf *puffer)
```

Formatiertes Schreiben mit dem Operator <<

Ein Schreiben in `ostream`-Objekte ist unter anderem mit dem Operator << möglich. Der Operator << hat eine Vielzahl von überladenen Varianten, da man ja beliebige Datentypen (wie z. B. `int`, `double`, Zeiger usw.) in ein `ostream`-Objekt schreiben kann, wie z. B:

```
class ostream {
  public:
     ostream& operator<< (char& wert);
     ostream& operator<< (int& wert);
     ostream& operator<< (long& wert);
     ostream& operator<< (double& wert);
     .....
};
```

Der Operator << liefert immer eine Referenz auf das betreffende `ostream`-Objekt zurück, so dass die folgende Anweisung:

```
   cout << "eins" << "zwei";
```

wie folgt abgearbeitet wird:

1. `cout << "eins"`: schreibt String "`eins`" in das `ostream`-Objekt `cout` und liefert eine Referenz auf das `ostream`-Objekt zurück, was hier `cout` ist, so dass dann anschließend noch Folgendes ausgeführt wird:
2. `cout << "zwei"`: schreibt String "`zwei`" in das `ostream`-Objekt `cout`.

Der Operator << wird immer von links nach rechts abgearbeitet. Die obige Anweisung entspricht somit:

```
   (cout << "eins") << "zwei";
```

Dieser Aufruf wird intern wie folgt realisiert:

```
   cout.operator<< ("eins").operator<< ("zwei");
```

Hat man z. B. die beiden `int`-Variablen `a` und `b`, dann kann die Summe wie folgt ausgegeben werden, da der Additionsoperator + eine höhere Priorität als << hat:

```
cout << a + b;
```

Soll dagegen das Ergebnis der bitweisen AND-Verknüpfung ausgegeben werden, dann sind zusätzliche Klammern erforderlich, da der Operator & eine geringere Priorität als der Operator << hat:

```
cout << (a & b);
```

Binäres Schreiben mit den Methoden put() und write()

Die beiden folgenden Methoden der Klasse `ostream` werden nicht von eventuell festgelegten Formateinstellungen (insbesondere die Feldbreite) beeinflusst:

`ostream& put(char c)`
: schreibt ein Zeichen in das `ostream`-Objekt. Da ein Zeichen ein Byte ist, kann diese Funktion auch verwendet werden, um ein Zeichen in eine Textdatei zu schreiben.

`ostream& write(const char *puffer, streamsize n)`
: schreibt `n` Bytes von Adresse `puffer` in das `ostream`-Objekt. Diese Funktion kann zur Ausgabe binärer Daten in eine Datei verwendet werden, wie z. B.

```
float x = 123.65;

strm.write((const char*)&x, sizeof(x)); // strm sollte ein Datei-Stream sein
```

Positionieren in ostream-Objekten

Bei `ostream`-Objekten, die Positionierung erlauben, kann der „Schreibzeiger“ neu positioniert werden. Um den „Schreibzeiger“ neu zu positionieren bzw. dessen aktuelle Position zu erfragen, stehen die folgenden beiden Methoden in der Klasse `ostream` zur Verfügung:

`pos_type tellp()` (*p steht hier für „put“*)
: liefert die Position, an der sich gerade der „Schreibzeiger“ befindet.

  Bei `fail()==true` gibt diese Methode -1 zurück.

`ostream& seekp(off_type offset, ios::seekdir wie=ios::beg)`
: `offset` legt dabei die Bytes fest, um die der „Schreibzeiger“ abhängig vom `wie`-Argument zu verschieben ist. Der Datentyp `off_type` kann als `long` angesehen werden. Das `wie`-Argument legt fest, von ab wo der „Schreibzeiger“ im `ostream`-Objekt um `offset` Bytes zu versetzen ist:

  - `ios::beg`: von Anfang an
  - `ios::cur`: ab aktueller Position
  - `ios::end`: von Ende an

Bei Eingabe-Streams haben `seekp()` und `seekg()` (siehe Seite 82) die gleiche Wirkung, da beide jeweils gleichzeitig die Ein- und Ausgabeposition setzen. Bei String-Streams wird jedoch bei `seekp()` nur die Schreib- und bei `seekg()` nur die Leseposition neu gesetzt. Es ist erlaubt, den „Schreibzeiger“ hinter das Dateiende zu positionieren. Ein Schreiben hinter dem `EOF` bewirkt, dass die dazwischenliegenden Bytes mit dem Wert `0` initialisiert werden. Ein Positionieren vor dem Dateianfang ist nicht erlaubt. Wird es versucht, so wird das Flag `ios::fail` gesetzt.

Leeren (Flushen) des Ausgabepuffers von ostream-Objekten

Wenn das Flag `ios::unitbuf` nicht gesetzt ist, werden die Daten, die in ein `ostream`-Objekt geschrieben werden, nicht sofort auf das entsprechende Ausgabemedium geschrieben, sondern in einem Ausgabepuffer zwischengespeichert. Erst wenn dieser interne Ausgabepuffer voll ist, werden diese zwischengespeicherten Daten auch wirklich auf das Ausgabemedium geschrieben.

Um zu erzwingen, dass der Inhalt des Ausgabepuffers eines `ostream`-Objekts auf das Ausgabemedium geschrieben wird, bietet die Klasse `ostream` die Funktion `flush()` an:

`ostream& flush()`

leert (*flushed*) den Ausgabepuffer, indem sie dessen Inhalt auf das Ausgabemedium schreiben lässt. Die Funktion `flush()` wird automatisch in folgenden Fällen aufgerufen:

- Das `ostream`-Objekt hört auf zu existieren.
- Die Manipulatoren `endl` oder `flush` werden auf das `ostream`-Objekt angewendet.
- Ein vom `ostream`-Objekt abgeleiteter Stream (wie z. B. `ofstream`) wird geschlossen.

Einfaches Beispielprogramm zu ostream

Programm 3.11 schreibt einen Satz in Datei `test.txt`, wobei deren alter Inhalt überschrieben wird.

Programm 3.11 – `ostream.cpp`:
Schreiben einer Zeile in die Datei test.tt

```
#include <iostream>
#include <fstream>
using namespace std;

int main(void) {
   filebuf datei;
   datei.open("test.txt", ios::out);
   ostream ausgabe(&datei);
   ausgabe << "Dieser Text wird in die Datei 'test.txt' geschrieben" << endl;
   datei.close();
}
```

3.4.2 Die Klasse ofstream – Ausgabe in Dateien

Die Klasse `ofstream` ist von der Klasse `ostream` abgeleitet und verfügt somit über die gleichen Fähigkeiten wie die zuvor vorgestellte Klasse `ostream`, nur dass die Klasse `ofstream` für das Schreiben in Dateien vorgesehen ist.

Um die Klasse `ofstream` benutzen zu können, muss man `<fstream>` inkludieren:

```
#include <fstream>
```

Es ist zu beachten, dass mit dem Inkludieren von `<fstream>` die Objekte `cin`, `cout` usw. nicht automatisch definiert sind. Benötigt man diese Objekte, muss man zusätzlich noch `<iostream>` inkludieren:

```
#include <iostream> // für cin, cout, cerr und clog
```

Anlegen von ofstream-Objekten

Die Klasse `ofstream` bietet die folgenden Konstruktoren an:

`ofstream()`
: Dieser Standardkonstruktor erzeugt ein `ofstream`-Objekt, das später mit der Methode `open()` auf eine Datei eingestellt werden kann.

`ofstream(const char *name, int mode=ios::out|ios::trunc)`
: legt zu der Datei `name` ein `ofstream`-Objekt an. Die Datei wird mit dem Modus `mode` geöffnet (siehe auch Tabelle 3.6).

Öffnen und Schliessen von Dateien

Um ein `ofstream`-Objekt nachträglich mit einer Datei zu assoziieren (Datei öffnen) bzw. eine einem `ofstream`-Objekt zugeteilte offene Datei wieder zu schliessen, bietet die Klasse `ofstream` die folgenden Methoden an:

```
void open(const char *name, int mode=ios::out|ios::trunc)
```

```
void close()
```

Um zu prüfen, ob ein `ofstream`-Objekt mit einer Datei verbunden ist, steht die folgende Methode zur Verfügung:

```
bool is_open() const
```

Öffnungsmodi für Dateien

Für den Parameter `mode` beim Konstruktor bzw. der Methode `open` können die in Tabelle 3.6 gezeigten Flags angegeben werden:

Tabelle 3.6: Flags für das Öffnen einer Datei

Flag	Bedeutung
`ios::out`	Öffnen einer Datei zum Schreiben (Voreinstellung für `ofstream`). Wenn Datei nicht existiert, wird sie angelegt. Existiert Datei, wird ihr alter Inhalt überschrieben.
`ios::app`	Schreiben nur am Dateiende; Mindestangabe: `ios::out | ios::app`
`ios::ate`	positioniert beim Öffnen auf das Dateiende und Ausgaben werden hinten angehängt. Benutzer kann aber – anders als bei `ios::app` – vor das ursprüngliche Dateiende positionieren und den alten Inhalt überschreiben
`ios::in`	Öffnen einer Datei zum Lesen (Voreinstellung für `ifstream`). Datei muss existieren
`ios::trunc`	Alten Inhalt zuerst löschen. Ist bei Ausgaben implizit eingestellt, wenn man weder `ios::app` noch `ios::ate` angibt.
`ios::binary`	öffnet eine Binärdatei; nur für Systeme (wie MS-DOS oder MS-Windows) von Wichtigkeit, die zwischen Text- und Binärdateien unterscheiden.

Die folgenden Kombinationen von Datei-Flags haben eine spezielle Bedeutung:

```
out | app // Existiert Datei nicht, wird sie angelegt.
          // Schreiben findet immer am Ende der Datei statt.
out | trunc // Alter Inhalt der Datei wird zuerst gelöscht,
            // bevor dann Datei neu beschrieben wird.
in | out // In Datei kann sowohl geschrieben als auch aus ihr gelesen werden;
         // allerdings muss Datei existieren
in | out | trunc // Alter Inhalt der Datei wird zuerst gelöscht.
                 // Danach ist Schreiben und Lesen in der Datei möglich.
```

Beispiel: Schreiben von Quadratzahlen in eine Datei

Programm 3.12 – `ofstream1.cpp`:
Schreiben von Quadratzahlen in eine Datei

```
#include <fstream>
#include <iomanip>
using namespace std;

int main(void) {
  ofstream of("datei1.txt");
   for (int i=1; i<=10; i++)
      of << setw(2) << i << setw(5) << i*i << endl;

   of.close();
}
```

In Programm 3.12 kann man erkennen, dass man die Headerdatei `<iostream>` nicht inkludieren muss, wenn man keines der Objekte `cout`, `cerr`, `cin` oder `clog` verwendet. Nachdem man Programm 3.12 ablaufen hat lassen, befindet sich im Working-Directory eine Datei `datei1.txt` mit folgendem Inhalt:

```
 1    1
 2    4
 3    9
 4   16
 5   25
 6   36
 7   49
 8   64
 9   81
10  100
```

Beispiel: Prüfen, ob ein ofstream-Objekt mit Datei verbunden ist

Programm 3.13 – `ofstream2.cpp`:
Prüfen, ob ein ostream-Objekt mit einer Datei verbunden ist

```
#include <fstream>
#include <iostream>
using namespace std;
```

```
int main(void) {
  ofstream of;
   cout << "Datei zugeordnet: " << boolalpha << of.is_open() << endl;
   of.open("testdatei");
   cout << "Datei zugeordnet: " << of.is_open() << endl;
   of.close();
   cout << "Datei zugeordnet: " << of.is_open() << endl;
}
```

Programm 3.13 liefert die folgende Ausgabe:

```
Datei zugeordnet: false
Datei zugeordnet: true
Datei zugeordnet: false
```

Beispiel: Unterschiedliche Öffnungsarten einer Datei

Programm 3.14 – `ofstream3.cpp`:
Öffnen und Schliessen einer Datei, sowie Prüfen, ob Öffnen erfolgreich war

```
#include <fstream>
#include <iostream>
#include <iomanip>
using namespace std;

int main(void) {
   ofstream *aus = new ofstream("datei2.txt", ios::out);

   for (int i=1; i<=4; i++)
      *aus << setw(2) << i << setw(5) << i*i << endl;
   aus->close();

   aus->open("datei2.txt", ios::app); // Schreiben am Dateiende

   if (!aus) { // Prüfen, ob Datei geöffnet werden konnte
      cerr << "kann Datei 'datei2.txt' nicht öffnen" << endl;
      exit(1);
   }
   *aus << "**********************" << endl;
   for (int i=5; i<=7; i++)
      *aus << setw(2) << i << setw(5) << i*i << endl;

   delete aus; // Löschen des Objekts --> Schliessen der Datei

   aus = new ofstream("datei2.txt", ios::app);
   *aus << "======================" << endl;
   for (int i=8; i<=10; i++)
      *aus << setw(2) << i << setw(5) << i*i << endl;
   aus->close();
}
```

Nachdem man Programm 3.14 ablaufen hat lassen, befindet sich im Working-Directory eine Datei `datei2.txt` mit folgendem Inhalt:

```
 1    1
 2    4
 3    9
 4   16
**********************
 5   25
 6   36
 7   49
======================
 8   64
 9   81
10  100
```

Beispiel: Schreiben von Binärdaten in eine Datei

Programm 3.15 – `ofstream4.cpp`:
Schreiben von Binärdaten in eine Datei

```
#include <iostream>
#include <fstream>
using namespace std;

int main(void)
{
   double dbl;

     // Binäre Datei zum Schreiben öffnen
   ofstream ofs("test.bin", ios::out | ios::binary);

   if (ofs) { // Öffnen erfolgreich?
      dbl = 3.14159;
      ofs.write((char *)&dbl, sizeof(double));
      dbl = 12345.6789;
      ofs.write((char *)&dbl, sizeof(double));
      dbl = 0.00045;
      ofs.write((char *)&dbl, sizeof(double));
      if (!ofs)
         cerr << "Fehler beim Schreiben in Datei 'test.bin'" << endl;
   } else
      cerr << "Fehler beim Oeffnen der Datei 'test.bin'" << endl;
   //.... Destruktor von ofstream schliesst die Datei!
}
```

Nachdem man Programm 3.15 ablaufen hat lassen, befindet sich im Working-Directory eine Datei `test.bin`, deren Inhalt nicht menschenlesbar ist. Ihren Inhalt kann man nur wieder binär auslesen, wie es z. B. mit Programm 3.28 auf Seite 87 gezeigt wird.

3.4.3 Die Klasse ostringstream – Ausgabe in Strings

Die Klasse `ostringstream` ist von der Klasse `ostream` abgeleitet und verfügt somit über die gleichen Fähigkeiten wie diese zuvor vorgestellte Klasse `ostream`, nur dass die Klasse `ostringstream` für das Schreiben in Strings vorgesehen ist.
Um die Klasse `ostringstream` benutzen zu können, muss man `<sstream>` inkludieren:

```
#include <sstream>
```

Es ist auch hier wieder zu beachten, dass mit dem Inkludieren von `<sstream>` die Objekte `cin`, `cout` usw. nicht automatisch definiert sind. Benötigt man diese Objekte, muss man zusätzlich noch `<iostream>` inkludieren:

```
#include <iostream> // für cin, cout, cerr und clog
```

Anlegen von ostringstream-Objekten

Die Klasse `ostringstream` bietet die folgenden Konstruktoren an:

`ostringstream(openmode mode = ios::out)`
: Dieser Standardkonstruktor erzeugt ein `ostringstream`-Objekt, das mit dem Modus `mode` geöffnet wird. Bei Modus `ios::out`, was die Voreinstellung ist, überschreiben alle Ausgaben in dieses Objekt die vorherigen Ausgaben.

`ostringstream(const string& s, openmode mode = ios::out)`
: Hier wird das `ostringstream`-Objekt mit dem String `s` initialisiert. Über den Parameter `mode` wird wie bei Dateien (siehe auch Tabelle 3.6) festgelegt, wie bei weiteren Schreibaktivitäten in diesem Objekt zu verfahren ist.

Erfragen bzw. Setzen des Strings in ostringstream-Objekten

`string str() const`
: liefert den String, der aktuell im `ostringstream`-Objekt gespeichert ist.

`void str(string& s)`
: speichert den String `s` im `ostringstream`-Objekt, wobei es einen eventuell dort schon vorhandenen String überschreibt.

Veraltete Klasse ostrstream

Bevor die Klasse `ostringstream` eingeführt wurde, wurde die Klasse `ostrstream` verwendet. Diese alte Klasse sollte heute nicht mehr verwendet werden, da sie bei nicht sorgfältiger Verwendung „Speicherleichen“ hinterlässt. Dieses Manko ist bei der Klasse `ostrstream` nun beseitigt.

Beispielprogramm zu ostringstream

Programm 3.16 – `ostringstream.cpp`:
Demonstrationsprogramm zu ostringstream

```
#include <iostream>
#include <string>
#include <sstream>
#include <fstream>
```

```
using namespace std;

int main(void)
{
   ostringstream ueberschreib("Eins ");
   ueberschreib << "wurde überschrieben";
   cout << ueberschreib.str() << endl;

   ostringstream anhaeng("One ", ios::ate);
   anhaeng << "Two: ";
   cout << anhaeng.str() << endl;

   anhaeng.setf(ios::hex, ios::basefield);
   anhaeng << showbase << 45054;
   cout << anhaeng.str() << endl;

   anhaeng.unsetf(ios::hex);
   anhaeng << " = " << 45054 << "(10)";
   cout << anhaeng.str() << endl;
}
```

Programm 3.16 liefert die folgende Ausgabe:

```
wurde überschrieben
One Two:
One Two: 0xaffe
One Two: 0xaffe = 45054(10)
```

3.5 Streamklassen für die Eingabe

Für die Eingabe bietet C++ mehrere Klassen an:

Klasse `istream` (`#include <iostream>`)
: stellt die grundlegende Funktionalität für die Eingabe zur Verfügung.

Klasse `ifstream` (`#include <fstream>`)
: ermöglicht das Öffnen von Dateien zum Lesen;
vergleichbar mit `fopen(`*`dateiname`*`, "r")` in C.

Klasse `istringstream` (`#include <sstream>`)
: ermöglicht das Lesen von Daten aus dem Speicher;
vergleichbar mit `sscanf()` in C.

3.5.1 Die Klasse istream – Basisfunktionalität für Eingabe

Die Klasse `istream` stellt die grundlegende Funktionalität für die Eingabe zur Verfügung. Das Objekt `cin` ist z. B. ein `istream`-Objekt. Wird dieses Objekt in einem Programm verwendet, muss man die Headerdatei `<iostream>` inkludieren.
Alle früher vorgestellten Konstrukte der Klasse `ios` sind soweit dies die Eingabe betrifft auch in der Klasse `istream` verfügbar, da diese Klasse alle diese Flags und die Methoden der Klasse `ios` erbt.

Lesen mit dem Operator >>

Ein Lesen aus `istream`-Objekten ist unter anderem mit dem Operator >> möglich. Der Operator >> hat eine Vielzahl von überladenen Varianten, da man ja beliebige Datentypen (wie z. B. `int`, `double`, `char` usw.) aus einem `istream`-Objekt lesen kann, wie z. B:

```
class istream {
  public:
     istream& operator>> (char& wert);
     istream& operator>> (int& wert);
     istream& operator>> (long& wert);
     istream& operator>> (double& wert);
     .....
};
```

Voreinstellung des Operators >>

- Führende Zwischenraum-Zeichen (*whitspaces*) werden überlesen.
- die folgenden Zeichen werden so lange in die aktuelle Variable eingelesen, bis ein Zeichen auftritt, das nicht zum Typ der Variablen passt.
  Ausnahme: Ein String wird so lange eingelesen, bis entweder ein Zwischenraum-Zeichen oder aber die mit `ios::width()` festgelegte Anzahl von Zeichen gelesen wurde.

>> gibt Referenz auf istream-Objekt zurück

Folgende Anweisung:

```
   cin >> a >> b;
```

wird wie folgt abgearbeitet:

1. `cin >> a`:
   liest aus dem `istream`-Objekt `cin` einen entsprechenden Wert in die Variable `a` ein und liefert eine Referenz auf das `istream`-Objekt zurück, was hier `cin` ist, so dass dann anschließend noch Folgendes ausgeführt wird:
2. `cin >> b`:
   liest nun aus dem `istream`-Objekt `cin` den nächsten Wert in die Variable `b` ein.

Einlesen mehrerer Werte mit einer cin-Anweisung

Werden mit einer `cin`-Anweisung mehrere Variablen eingelesen, so muss man, wenn nicht mit `ios::width()` eine maximale Zahl von einzulesenden Zeichen festgelegt wurde, die einzelnen Variablen durch Zwischenraum-Zeichen bei der Eingabe trennen.

Programm 3.17 – `cin1.cpp`:
Demonstrationsprogramm zu cin

```
#include <iostream>
using namespace std;
int main(void)
{
  int    i;
  double dbl;
  string s1, s2;
```

```
   cin >> i >> dbl; cout << "  i = " << i << endl << "dbl = " << dbl << endl;
   cin >> i >> dbl; cout << "  i = " << i << endl << "dbl = " << dbl << endl;
   cin >> s1 >> s2; cout << "s1 = " << s1 << endl << "s2 = " << s2 << endl;
   cin.width(4);
   cin >> s1 >> s2; cout << "s1 = " << s1 << endl << "s2 = " << s2 << endl;
}
```

Möglicher Ablauf des Programms 3.17:

```
123.456 ⏎
  i = 123
dbl = 0.456
1234 7.5252 ⏎
  i = 1234
dbl = 7.5252
Georg-Simon-Ohm Fachhochschule Nuernberg-university of applied sciences ⏎
s1 = Georg-Simon-Ohm
s2 = Fachhochschule
s1 = Nuer
s2 = nberg-university
```

ios-Flags und Manipulatoren bei cin

- **`ios`-Flags bzw. Manipulatoren `dec`, `oct`, `hex`**
  Ist keines der `ios`-Flags bzw. Manipulatoren `dec` (Voreinstellung), `oct` oder `hex` gesetzt, dann kann bei der Eingabe von Ganzzahlen deren Basis wie in C frei gewählt werden:
  - Oktalzahl (bei führender 0)
  - Hexadezimalzahl (Präfix `0x` oder `0X`)

  Durch Setzen eines der ios-Flags bzw. Manipulatoren `dec`, `oct` oder `hex` wird die Eingabe in der gewählten Basis erzwungen, ohne dass ein Präfix verwendet wird. Dabei können die entsprechenden Manipulatoren wie bei der Ausgabe eingesetzt werden, wie z. B. `cin >> hex >> hexvar`.
- **`ios`-Flag und Manipulator `skipws`**
  Das `ios`-Flag `skipws` ist in der Voreinstellung gesetzt. Dies führt dazu, dass führende Zwischenraum-Zeichen beim Lesen mit >> überlesen werden.
  Durch Löschen dieses Flags z. B. mit

  ```
  cin.unsetf(ios::skipws); // oder aber
  cin >> noskipws >> var; // Einsatz des Manipulators noskipws
  ```

  kann ein abweichendes Verhalten erzwungen werden.
- **`ios`-Methode `width()` und Manipulator `setw()`**
  Mit der Methode `ios::width()` bzw. dem Manipulator `setw()` kann man eine maximal einzulesende Zahl von Zeichen bei Stringvariablen festlegen, wobei bei `char`-Arrays ein Zeichen weniger (wegen 0-Terminierung) eingelesen wird.
- **`ios`-Flag und Manipulator `boolalpha`**
  Ist das `ios`-Flag bzw. der Manipulator `boolalpha` für `cin` gesetzt, kann man bei der Eingabe von booleschen Werten `true` und `false` angeben; siehe auch Programm 3.19.

Programm 3.18 – `cin2.cpp`:
Weiteres Demonstrationsprogramm zu cin

```
#include <string>
#include <iostream>
#include <iomanip>
using namespace std;
int main(void) {
  int n1, n2, n3;
  string str1, str2;
  char   puffer1[4], puffer2[5];
   cin.unsetf(ios::basefield);
   cin >> n1 >> n2 >> n3;
   cout << n1 << ", " << n2 << ", " << n3 << endl;
   cin >> dec >> n1 >> oct >> n2 >> hex >> n3;
   cout << n1 << ", " << n2 << ", " << n3 << endl;
   cin >> dec >> n1 >> dec >> n2 >> dec >> n3;
   cout << n1 << ", " << n2 << ", " << n3 << endl;
   cin.ignore(100, '\n');
   cin.width(4);
   cin >> str1; // 4 Zeichen einlesen
   cin >> setw(5) >> str2; // 5 Zeichen einlesen
   cout << "\"" << str1 << "\", " << "\"" << str2 << "\"" << endl;
   cin.ignore(100, '\n'); cin.width(4);
   cin >> puffer1; // nur 3 Zeichen (wegen \0) einlesen
   cin >> setw(sizeof(puffer2)) >> puffer2; // nur 4 Zeichen (wegen \0) einlesen
   cout << "\"" << puffer1 << "\", " << "\"" << puffer2 << "\""<< endl;
}
```

Möglicher Ablauf des Programms 3.18:

```
100 0100 0x100 ⏎
100, 64, 256
100 100 100 ⏎
100, 64, 256
100 0100 0x100 ⏎
100, 100, 0
Donaukanaldampferschiff ⏎
"Dona", "ukana"
Donaukanaldampferschiff ⏎
"Don", "auka"
```

Programm 3.19 – `cin3.cpp`:
Eingeben von true und false bei cin

```
#include <iostream>
#include <iomanip>
using namespace std;
int main(void) {
  bool x, y;
   cin >> boolalpha >> x >> y;
   cout << x << ", " << y << ": " << boolalpha << x << ", " << y << endl;
}
```

Möglicher Ablauf des Programms 3.19:

```
false true ⏎
0, 1: false, true
```

Fehlerstatus bei cin

Im Zusammenhang mit der Ein-/Ausgabe gibt es eine Reihe von Fehlermöglichkeiten (siehe auch Zustandsflags `badbit`, `failbit` usw. in Kapitel 3.3.2 auf Seite 50). Ein Programm sollte in der Lage sein, auf diese Fehler in geeigneter Form zu reagieren. Außerdem kann der Fehlerstatus auch mit den folgenden beiden Operatorfunktionen abgefragt werden:

```
int ios::operator!()   { return fail();            }
ios::operator void *() { return fail() ? 0 : this; }
```

Mit Hilfe dieser Operatorfunktionen kann ein Stream-Objekt wie ein Wahrheitswert abgefragt werden:

```
cin >> n;
if (cin) {  // ruft cin.operator void *() auf
   // Stream ist nach dem Einlesen von n fehlerfrei
   // ...
}
cin >> n;
if (!cin) { // ruft cin::operator!() auf
   // Stream weist nach dem Einlesen von n einen Fehler auf
   // ...
}
```

Ist eines der Fehlerflags gesetzt, dann wird der Stream „suspendiert“, was bedeutet, dass alle Ein-/Ausgaben aus einem suspendierten Stream so lange ignoriert werden, bis der Fehler beseitigt wird (sofern das möglich ist) und das bzw. die Fehlerflags zurückgesetzt werden. Soll z. B. eine dezimale ganze Zahl eingelesen werden:

```
cin >> n;
```

und der Benutzer tippt versehentlich einen oder mehrere Buchstaben ein, dann kann dieser Fehler leicht erkannt werden, da `ios::failbit` gesetzt wird. Damit der Benutzer die fehlerhafte Eingabe wiederholen kann, muss natürlich der Fehlerstatus zurückgesetzt werden:

```
cin.clear();
```

Diese Maßnahme allein reicht jedoch nicht aus. Der oder die fehlerhaften Buchstaben sind nämlich noch nicht aus dem Stream entfernt und führen beim nächsten Leseversuch erneut zu einem Fehler. Da die Eingabe gepuffert arbeitet, enthält sie als letztes Zeichen ein `'\n'`. Deshalb werden mit der folgenden Anweisung alle „alten“ Zeichen aus dem Eingabepuffer entfernt:

```
cin.ignore(INT_MAX, '\n');
```

Programm 3.20 – `cin4.cpp`:
Demoprogramm zum Fehlerstatus bei cin

```
#include <iostream>
#include <fstream>
#include <climits>
using namespace std;
void zustandsflags(void) {
   cout << "..........";
   if (cin.bad())  cout << "badbit, ";
   if (cin.eof())  cout << "eofbit, ";
   if (cin.fail()) cout << "failbit, ";
   if (cin.good()) cout << "goodbit ";
   cout << endl;
}
int main(void) {
  int n;
   cout << "1. ganze Zahl (ohne Fehler!): ";
   cin >> n;
   cout << "  n = " << n;  zustandsflags();
   cout << "2. ganze Zahl (Buchstaben eingeben!): ";
   cin >> n;
   cout << "  n = " << n;  zustandsflags();
    // Alle Eingabeoperationen mit cin sind solange suspendiert,
    // bis Fehlerflags gelöscht werden
   cout << "3. keine Eingabe mögl., da Stream suspendiert!" << endl;
   cin >> n;
   cout << "  n = " << n;  zustandsflags();
   cin.clear(); // Fehlerflags löschen
   cout << "  clear() "; zustandsflags();

    // Fehlerflags gelöscht, aber unzulässiges Zeichen immer noch im Stream
   cout << "4. Eingabe erfolgt aus Puffer, da nicht leer! ";
   cin >> n;
   cout << endl << "  n = " << n; zustandsflags();

    // Fehlerflags löschen und alten Pufferinhalt löschen
   cin.clear(); // Fehlerflags löschen
   cin.ignore(INT_MAX, '\n'); // Pufferinhalt löschen
   cout << "  clear() und ignore()"; zustandsflags();

    // jetzt erst bereit fuer neue Eingabe
   cout << "5. ganze Zahl (ohne Fehler!): ";
   cin >> n;
   cout << "  n = " << n;  zustandsflags();

    // Eingabe von EOF
   cout << "6. ganze Zahl (Strg-D (Linux/Unix); Strg-Z (Windows/Dos)): ";
   cin >> n;        // durch Eingabe von Ctrl-Z wird Dateiende simuliert
   cout << endl << "  n = " << n;  zustandsflags();
}
```

Möglicher Ablauf des Programms 3.20:

```
1. ganze Zahl (ohne Fehler!): 1234 ⏎
  n = 1234..........goodbit
2. ganze Zahl (Buchstaben eingeben!): x ⏎
  n = 1234..........failbit,
3. keine Eingabe mögl., da Stream suspendiert!
  n = 1234..........failbit,
  clear() ..........goodbit
4. Eingabe erfolgt aus Puffer, da nicht leer!
  n = 1234..........failbit,
  clear() und ignore()..........goodbit
5. ganze Zahl (ohne Fehler!): 5678 ⏎
  n = 5678..........goodbit
6. ganze Zahl (Strg-D (Linux/Unix); Strg-Z (Windows/Dos)): Strg-D bzw. Strg-Z
  n = 5678..........eofbit, failbit,
```

Programm 3.21 – `cin5.cpp`:
Weiteres Demoprogramm zum Fehlerstatus bei cin

```
#include <iostream>
#include <climits>
using namespace std;
int main(void) {
  int n, i;
   for (i = 0; i < 2; i++) {
      cout << "ganze Zahl: ";  cin >> n;
      if (cin)
         cout << "   n = " << n << endl;
      else
         cout << "   ....Eingabefehler" << endl;
   }
   cin.clear(); // Fehlerflags löschen
   cin.ignore(INT_MAX, '\n');  // alten Pufferinhalt löschen
   for (i = 0; i < 2; i++) {
      cout << "ganze Zahl: ";  cin >> n;
      if (!cin)
         cout << "   ....Eingabefehler" << endl;
      else
         cout << "   n = " << n << endl;
   }
}
```

Möglicher Ablauf des Programms 3.21:

```
ganze Zahl: 1234 ⏎
   n = 1234
ganze Zahl: x ⏎
   ....Eingabefehler
ganze Zahl: 5678 ⏎
   n = 5678
ganze Zahl: x ⏎
   ....Eingabefehler
```

Anlegen von istream-Objekten

Um `istream`-Objekte anzulegen, steht der folgende Konstruktor zur Verfügung:

`istream(streambuf *puffer)`
: legt ein `istream`-Objekt zu `puffer` an.

Binäres Lesen

Beim Lesen von biären Dateien (wie z. B. ausführbaren Dateien oder Bildern), darf normalerweise keine Formatierung oder Konvertierung (wie z. B. Umformen eines `int`-Werts in ASCII-Zeichen für die Anzeige am Bildschirm) stattfinden, sondern müssen die Bytes unverändert als Binärdaten gelesen werden. Zum binären Lesen aus `istream`-Objekten stehen die folgenden Methoden zur Verfügung:

`int get()`
: liest das nächste Zeichen aus dem `istream`-Objekt und liefert dessen `int`-Wert. Bei Dateiende liefert diese Methode `EOF` zurück.

`istream& get(char& c)`
: liest das nächste Zeichen aus dem `istream`-Objekt und schreibt es in das Argument zu `c`. Das zurückgegebene `istream`-Objekt kann herangezogen werden, um zu prüfen, ob das Lesen erfolgreich war.

`istream& get(char *s, streamsize n, char endezeichen='\n')`
: liest maximal `n-1` Zeichen aus dem `istream`-Objekt und schreibt diese in den Puffer mit der Anfangsadresse `s`.

  Das Lesen wird dabei vorzeitig beendet, wenn `EOF` oder das Zeichen `endezeichen` gelesen wird. Im letzteren Fall wird das `endezeichen` anders als bei der Methode `getline()` nicht aus dem Eingabestream entfernt und gilt als noch nicht gelesen. Die Methoden `eof()` und `fail()` liefern 0 (`false`), wenn beim Lesen nicht das Dateiende erreicht wurde bzw. nicht auf das `endezeichen` getroffen wurde.

  In jedem Fall wird der Puffer `s` mit `0` terminiert.

`istream& get(streambuf& sb, char endezeichen='\n')`
: entspricht weitgehend der vorherigen Methode, nur dass sie die gelesenen Zeichen nicht in einen Puffer schreibt, sondern in das `streambuf`-Objekt `sb`.

`istream& getline(char *s, streamsize n, char endezeichen='\n')`
: entspricht weitgehend der entsprechenden `get()`-Methode (siehe vorher). Sie unterscheidet sich lediglich in den beiden folgenden Punkten:

  Ein gelesenes Endezeichen wird anders als bei `get()` aus dem Eingabestream entfernt, jedoch nicht in den Puffer `s` geschrieben.

  Die Methoden `fail()` und `eof()` liefern 1 (`true`), wenn beim Lesen nicht auf das `endezeichen` getroffen bzw. das Dateiende erreicht wurde.

`istream& read(char *s, streamsize n)`
: liest bis zu `n` Bytes aus dem `istream`-Objekt und schreibt diese ohne 0-Terminierung in den Puffer mit der Anfangsadresse `s`. Wird beim Lesen vorzeitig auf das Dateiende getroffen, werden weniger als `n` Bytes gelesen. In diesem Fall wird `setstate(failbit)` aufgerufen. Mit der Methode `gcount()` kann man nachträglich erfragen, wie viele Bytes wirklich gelesen wurden.

`streamsize readsome(char *s, streamsize n)`
liest bis zu `n` Bytes aus dem `istream`-Objekt und schreibt diese in den Puffer mit der Anfangsadresse `s`. Kann überhaupt kein Byte gelesen werden, wird `setstate(eofbit)` aufgerufen. Die Anzahl der wirklich gelesenen Bytes liefert diese Methode als Rückgabewert.

`istream& ignore(streamsize n=1, int endezeichen=EOF)`
Für diese Methode gibt es mehrere Aufrufmöglichkeiten:

keine Argumente: es werden alle Zeichen aus dem `istream`-Objekt entfernt.

Ein Argument `n`: es werden `n` Zeichen aus dem `istream`-Objekt entfernt.

Zwei Argumente `n` und `ende`: es werden entweder `n` Zeichen oder aber die Zeichen bis einschließlich dem `endezeichen` aus dem `istream`-Objekt entfernt, je nachdem was zuerst zutrifft.

`streamsize gcount() const`
liefert die Bytezahl, die durch das letzte unformatierte Lesen mit einer der Methoden `get()`, `getline()`, `ignore()`, `read()` oder `readsome()` gelesen wurde.

`istream& unget()`
zuletzt gelesenes Zeichen als „ungelesen“ markieren, so dass es beim nächsten Lesen das erste Zeichen ist, das gelesen wird.

`istream& putback(char c)`
zuletzt gelesenes Zeichen `c` als „ungelesen“ markieren, so dass es beim nächsten Lesen das erste Zeichen ist, das gelesen wird. In jedem Fall muss `c` auch das Zeichen sein, das zuletzt gelesen wurde, andernfalls kann diese Methode nicht erfolgreich ausgeführt werden.

`int peek()`
liefert das nächste Zeichen aus dem `istream`-Objekt, enfernt es aber nicht. Bei Dateiende oder wenn `good()` `false` liefert, wird `EOF` zurückgegeben

Positionieren in istream-Objekten

Bei `istream`-Objekten kann der „Lesezeiger“ neu positioniert werden. So ist es z. B. möglich früher gelesene Zeichen erneut zu lesen, indem man den „Lesezeiger“ weiter nach vorne positioniert. Um den „Lesezeiger“ neu zu positionieren bzw. dessen aktuelle Position zu erfragen, stehen die folgenden beiden Methoden in der Klasse `istream` zur Verfügung:

`pos_type tellg()` (*`g` steht hier für „get“*)
liefert die Position, an der sich gerade der „Lesezeiger“ befindet.

Bei `fail()==true` gibt diese Methode -1 zurück.

`istream& seekg(off_type offset, ios::seekdir wie=ios::beg)`
`offset` legt dabei die Bytes fest, um die der „Lesezeiger“ abhängig vom `wie`-Argument zu verschieben ist. Der Datentyp `off_type` kann als `long` angesehen werden. Das `wie`-Argument legt fest, von ab wo der „Lesezeiger“ im `istream`-Objekt um `offset` Bytes zu versetzen ist:

- `ios::beg`: von Anfang an
- `ios::cur`: ab aktueller Position
- `ios::end`: von Ende an

Bei Ausgabe-Streams haben `seekp()` (siehe Seite 67) und `seekg()` die gleiche Wirkung, da beide jeweils gleichzeitig die Ein- und Ausgabeposition setzen. Bei String-Streams wird jedoch bei `seekp()` nur die Schreib- und bei `seekg()` nur die Leseposition neu gesetzt. Es ist erlaubt, den „Lesezeiger“ hinter das Dateiende zu positionieren, wobei aber ein Leseversuch hinter dem Dateiende zu einem Fehler führt. Ein Positionieren vor dem Dateianfang ist nicht erlaubt. Wird es versucht, so wird das Flag `ios::fail` gesetzt.

Beispiel: Ausgeben einer Datei mit Zeilennumerierung

Programm 3.22 gibt die Datei, deren Name auf der Kommandozeile angegeben wird, mit Zeilennummern aus.

Programm 3.22 – `istream1.cpp`:
Ausgeben einer Datei mit Zeilennumerierung

```
#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;

int main(int argc, char *argv[]) {
  char zeile[1000];
  int  n = 0;
   ifstream datei;
   datei.open(argv[1]); // Datei öffnen
   while (!datei.getline(zeile, 1000).eof()) // Zeile für Zeile aus Datei lesen
      cout << setw(5) << ++n << "  " << zeile << endl;
   datei.close(); // Datei wieder schliessen
}
```

Beispiel: Einlesen einer Datei in den Speicher und dann ausgeben

Programm 3.23 liest den Inhalt der ganzen Datei, deren Name auf der Kommandozeile angegeben wird, in einen Puffer, bevor es diesen Puffer dann auf die Standardausgabe schreibt.

Programm 3.23 – `istream2.cpp`:
Einlesen einer Datei in den Speicher und dann ausgeben

```
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char *argv[]) {
 int  groesse;
 char *puffer;
  ifstream is;
  is.open(argv[1], ios::binary); // Datei öffnen
  is.seekg(0, ios::end); // Ermitteln der Dateigrö"se
  groesse = is.tellg();
  is.seekg(0, ios::beg);
  puffer = new char[groesse]; // Puffer allozieren
  is.read(puffer,groesse);    // Daten aus datei lesen
  is.close(); // Datei schliessen
  cout.write(puffer, groesse); // Pufferinhalt auf Standardausgabe ausgeben
}
```

Beispiel: Rückwärtiges Auslesen von Zahlwörtern aus einer Datei

Programm 3.24 schreibt zunächst in eine Datei deutsche Zahlwörter, schliesst dann diese Datei, bevor es sie dann zum Lesen öffnet, um diese Zahlwörter dann in umgekehrter Reihenfolge aus dieser Datei zu lesen und anschließend auf der Standardausgabe auszugeben.

Programm 3.24 – `istream3.cpp`:
Rückwärtiges Auslesen von Zahlwörtern aus einer Datei

```
#include <fstream>
#include <iostream>
using namespace std;

#define ZAHL_LEN  5 // +1 wegen \0
int main(void)
{
   char *zahlen[] = { "Null", "Eins", "Zwei", "Drei" };
   char neuZahlen[4][ZAHL_LEN] =  0 ;

   //................................. Zahlen in Datei "zahl.txt" schreiben
   ofstream aus("zahl.txt", ios::out);
   for(int i = 0; i < 4; i++)
     aus.write(zahlen[i], ZAHL_LEN);
   aus.close();

   //.......................... Zahlen rückwärts aus Datei "zahl.txt" lesen
   ifstream ein("zahl.txt", ios::in);

   ein.seekg(-ZAHL_LEN, ios::end);    // Auf letzte Zahl positionieren
   ein.read(neuZahlen[0], ZAHL_LEN);  // und einlesen

   ein.seekg(2 * ZAHL_LEN);           // Auf vorletzte Zahl positionieren
   ein.read(neuZahlen[1], ZAHL_LEN);  // und einlesen

   ein.seekg(-ZAHL_LEN * 2, ios::cur); // Auf 2.  Zahl positionieren
   ein.read(neuZahlen[2], ZAHL_LEN);   // und einlesen

   ein.seekg(0, ios::beg);             // Auf 1. Zahl positionieren
   ein.read(neuZahlen[3], ZAHL_LEN);   // und einlese

   for (int j = 0; j < 4; j++)
      cout << neuZahlen[j] << ", ";
   cout << endl;
}
```

Programm 3.24 liefert die folgende Ausgabe:

```
Drei, Zwei, Eins, Null,
```

3.5.2 Die Klasse ifstream – Lesen aus Dateien

Die Klasse `ifstream` ist von der Klasse `istream` abgeleitet und verfügt somit über die gleichen Fähigkeiten wie die zuvor vorgestellte Klasse `istream`, nur dass die Klasse `ifstream` für das Lesen aus Dateien vorgesehen ist.
Um die Klasse `ifstream` benutzen zu können, muss man `<fstream>` inkludieren:

```
#include <fstream>
```

Es ist zu beachten, dass mit dem Inkludieren von `<fstream>` die Objekte `cin`, `cout` usw. nicht automatisch definiert sind. Benötigt man diese Objekte, muss man zusätzlich noch `<iostream>` inkludieren:

```
#include <iostream> // für cin, cout, cerr und clog
```

Anlegen von ifstream-Objekten

Die Klasse `ifstream` bietet die folgenden Konstruktoren an:

`ifstream()`
: Dieser Standardkonstruktor erzeugt ein `ifstream`-Objekt, das später mit der Methode `open()` auf eine Datei eingestellt werden kann.

`ifstream(const char *name, int mode=ios::in)`
: legt zu der Datei `name` ein `ifstream`-Objekt an. Die Datei wird mit dem Modus `mode` geöffnet (siehe auch Tabelle 3.6 uaf Seite 69).

Öffnen und Schliessen von Dateien

Um ein `ifstream`-Objekt nachträglich mit einer Datei zu assoziieren (Datei öffnen) bzw. eine einem `ifstream`-Objekt zugeteilte offene Datei wieder zu schliessen, bietet die Klasse `ifstream` die folgenden Methoden an:

```
void open(const char *name, int mode=ios::in)
```

```
void close()
```

Um zu prüfen, ob ein `ifstream`-Objekt mit einer Datei verbunden ist, steht die folgende Methode zur Verfügung:

```
bool is_open() const
```

Beispiel: Zeichenweises Einlesen und Ausgeben einer Datei

Programm 3.25 liest die Datei, deren Name auf der Kommandozeile angegeben wird, zeichenweise ein und gibt sie entsprechend auch zeichenweise aus.

Programm 3.25 – `ifstream1.cpp`:
Zeichenweises Einlesen und Ausgeben einer Datei

```
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char *argv[]) {
   ifstream datei;
   datei.open(argv[1]);
```

```
   if (datei.is_open()) {
      while (datei.good())
         cout << (char) datei.get();
      datei.close();
   } else
      cerr << "Kann Datei '" << argv[1] << "'nicht \"offnen" << endl;
}
```

Die Programme 3.26 und 3.27 leisten das Gleiche wie Programm 3.25. Sie zeigen lediglich alternative Möglichkeiten, um den Inhalt einer Datei zeichenweise einzulesen und dann zeichenweise auszugeben.

Programm 3.26 – `ifstream1a.cpp`:
Alternative zu Programm 3.25

```
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[]) {
   ifstream datei(argv[1]); // Datei argv[1] öffnen

   if (datei) {
      char z;
      while (true) {
         datei.get(z);
         if (datei.fail())
            break;
         cout << z;
      }
   } else
      cerr << "Kann Datei '" << argv[1] << "'nicht öffnen" << endl;
}
```

Programm 3.27 – `ifstream1b.cpp`:
Weitere Alternative zu Programm 3.25

```
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[]) {
   ifstream datei(argv[1]); // Datei argv[1] öffnen

   if (datei) {
      char z;
      while (datei.get(z))
         cout << z;
   } else
      cerr << "Kann Datei '" << argv[1] << "'nicht öffnen" << endl;
}
```

Beispiel: Lesen von Binärdaten aus einer Datei

Programm 3.28 liest binäre Daten aus der Datei `test.bin`, die dorthin mit Programm 3.15 auf Seite 72 geschrieben wurden, und gibt diese in menschenlesbarer Form auf der Standardausgabe aus.

Programm 3.28 – `ifstream2.cpp`: Lesen von Binärdaten aus einer Datei

```
#include <iostream>
#include <fstream>
using namespace std;
int main(void) {
   double dbl;
     // Binäre Datei zum Lesen öffnen
   ifstream ifs("test.bin", ios::in | ios::binary);
   if (ifs) // Öffnen erfolgreich
      while (ifs.read((char *)&dbl, sizeof(double)))
         cout << dbl << endl;
   else
      cerr << "Fehler beim Öffnen der Datei 'test.bin'" << endl;
   //.... Destruktor von ifstream schliesst die Datei!
}
```

Programm 3.28 liefert die folgende Ausgabe:

```
3.14159
12345.7
0.00045
```

3.5.3 Die Klasse istringstream – Einlesen aus Strings

Die Klasse `istringstream` ist von der Klasse `istream` abgeleitet und verfügt somit über die gleichen Fähigkeiten wie die zuvor vorgestellte Klasse `istream`, nur dass `istringstream` für das Lesen aus Strings vorgesehen ist. Um die Klasse `istringstream` benutzen zu können, muss man `<sstream>` inkludieren:

```
#include <sstream>
```

Es ist auch hier wieder zu beachten, dass mit dem Inkludieren von `<sstream>` die Objekte `cin`, `cout` usw. nicht automatisch definiert sind. Benötigt man diese Objekte, muss man zusätzlich noch `<iostream>` inkludieren:

```
#include <iostream> // für cin, cout, cerr und clog
```

Anlegen von istringstream-Objekten

Die Klasse `istringstream` bietet die folgenden Konstruktoren an:

`istringstream(openmode mode = ios::in)`
: Dieser Standardkonstruktor erzeugt ein leeres `istringstream`-Objekt, das mit dem Modus `mode` geöffnet wird. Dieses Objekt kann dann mit Zeichen gefüllt werden, die man später auslesen kann.

`istringstream(const string& s, openmode mode = ios::in)`
: Hier wird das `istringstream`-Objekt mit dem String `s` initialisiert. Über den Parameter `mode` wird wie bei Dateien (siehe auch Tabelle 3.6) festgelegt, wie bei weiteren Leseaktivitäten in diesem Objekt zu verfahren ist.

Erfragen bzw. Setzen des Strings in istringstream-Objekten

`string str() const`
: liefert den String, der aktuell im `istringstream`-Objekt gespeichert ist.

`void str(string& s)`
: speichert den String `s` im `istringstream`-Objekt, wobei es einen eventuell dort schon vorhandenen String überschreibt.

Beispielprogramm zu istringstream

Programm 3.29 zeigt unter anderem, wie man Zahlen, die als Strings vorliegen, leicht in numerische Werte konvertieren kann.

Programm 3.29 – `istringstream.cpp`: Demonstrationsprogramm zu istringstream

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
int main(void) {
   int    n, ganz;
   double gleit;
   //---------------------------------------------------------------------
   istringstream istr1; // leeres istringstream-Objekt
   string zahlen = "1980 200 12 10001";
   istr1.str(zahlen); // Werte in istr1 speichern
   for (n=0; n<4; n++) {
      istr1 >> ganz;
      cout << ganz+1 << ", ";
   }
   cout << endl;
   //---------------------------------------------------------------------
   string gltpkt;
   gltpkt = "12.34  3.14159  4.5  -123.9";
   istringstream istr2(gltpkt, ios::in);

   while (!istr2.eof()) {
      istr2 >> gleit;
      cout << gleit << ", ";
   }
   cout << endl;
   //---------------------------------------------------------------------
   istringstream istr3("123 789");
   istr3.seekg(2);  // überspringe "12"
   istr3 >> ganz;   // liest eine ganze Zahl (bleibt 3 zu lesen)
   cout << ganz << ", ";
```

```
   istr3.seekg(0);  // Positioniere Lesezeiger auf den Anfang
   istr3 >> ganz;   // liest eine ganze Zahl (123 wird gelesen)
   cout << ganz << ", ";
   istr3.str("9993.14159"); // speichere neuen String in istr3
   istr3.seekg(3);  // Positioniere auf 4. Zeichen
   istr3 >> gleit;  // liest eine Gleitpunktzahl (hier 3.14159)
   cout << gleit << endl;
}
```

Programm 3.29 liefert die folgende Ausgabe:

```
1981, 201, 13, 10002,
12.34, 3.14159, 4.5, -123.9,
3, 123, 3.14159
```

Konvertieren von Strings in beliebige Datentypen

Programm 3.30 zeigt eine von `istringstream` abgeleitete Klasse `strtox`, die das Konvertieren eines `istringstream`-Strings in einen beliebigen Datentyp ermöglicht.

Programm 3.30 – `strtox.cpp`:
Konvertieren von Strings in beliebige Datentypen

```
#include <iostream>
#include <sstream>
using namespace std;
//-------------------------------------------------------------------------strtox
class strtox : public istringstream {
  public:
     strtox() {}
     strtox(const char *str) : istringstream(string(str)) {}
     strtox(const string& s) : istringstream(s)           {}

     strtox& operator=(const string& s)     { return operator=(s.c_str());   }
     strtox& operator=(const strtox& other) { return operator=(other.str()); }
     strtox& operator=(const char *cstr);

     //... Membertemplate T, das sich automat. an den Typ der Variable anpasst,
     //... der ein Wert zuzuweisen ist. Konvertiert wird dabei der String des
     //... jeweiligen strtox (istringstream)-Objekts in den entspr. Typ.
     //... Ist keine erfolgreiche Konvertierung möglich, wird entspr. Fehlerflag
     //... gesetzt (good() liefert danach false).
     template <typename T>
     operator T() {
        T typ;
        return (*this >> typ) ? typ : T();
     }
     //... Membertemplate to: ermöglicht die Angabe des gewünschten Typs,
     //... in den zu konvertieren ist
     template <typename T>
     T to(T const&) { return *this; }
};
```

```
//-----------------------------------------------------operator= (nicht inline)
strtox& strtox::operator=(const char *cstr) {
   clear(); // evtl. gesetzte Fehlerflags löschen
   str(cstr);
   return *this;
}
//-------------------------------------------ausgabeInt() und ausgabeDouble()
void ausgabeInt(int i)       { cout << i << endl; }
void ausgabeDouble(double d) { cout << d << endl; }
//-------------------------------------------------------------------------main
int main(void) {
   int i = strtox("4711");  // String in int konvertieren
   cout << i << endl;

   strtox str("3.1415");
   double d = str;   // String in double konvertieren
   cout << d << endl;

   str = "eins2gesuffa";
   d = str;          // d wird 0, da keine Konvertierung (am Anfang) möglich
   cout << d << endl;

   str = "24.5mal2=49";
   d = str;          // d wird 24.5, da Konvertierung am Anfang
   cout << d << endl;

   str = "     viele Leerzeichen am Anfang";
   char *s = str; // führende Leerzeichen werden überlesen
   cout << s << endl;

   ausgabeInt(strtox("753 Rom kroch aus dem Ei"));
   ausgabeDouble(strtox("0.33333 entspricht einem Drittel"));

   d = strtox("59.76").to(int()); // d wird 59 und nicht 59.76 zugewiesen
   cout << d << endl;
}
```

Programm 3.30 liefert die folgende Ausgabe:

```
4711
3.1415
0
24.5
viele
753
0.33333
59
```

3.6 Die Klassen fstream und stringstream

- **Die Klasse `fstream`**
  Die Klasse `fstream` beinhaltet alle Eigenschaften und Methoden der beiden Klassen `ofstream` und `ifstream`, so dass in der zugeörigen Datei gleichzeitig sowohl geschrieben als auch gelesen werden kann.
  Bei `fstream`-Objekten mit abwechselnden Schreiben und Lesen in einer Datei ist oft nicht erkennbar, an welcher Position sich der Schreib-/Lesezeiger gerade befindet. In solchen Fällen kann man `seekp()` und `seekg()` einsetzen, um unerwünschtes Lesen und Schreiben an falschen Stellen zu vermeiden.
  Kapitel 3.9.3 auf Seite 110 zeigt hierzu Beispiele.
- **Die Klasse `stringstream`**
  Die Klasse `stringstream` beinhaltet alle Eigenschaften und Methoden der beiden Klassen `ostringstream` und `istringstream`, so dass der zugehörige String gleichzeitig sowohl geschrieben als auch gelesen werden kann.

3.7 Benutzerdefinierte Ein-/Ausgabe

3.7.1 Stream-Ausgabe für eigene Klassen

Man kann die Ausgabe auf eigene Klassen erweitern, indem man die Operatorfunktion `operator<<()` global überlädt, wie z. B. für die eigene Klasse `CBruch`:

```
ostream& operator<<(ostream& os, const CBruch& b) {
   return os << '(' << b.getZaehler() << "/" << b.getNenner() << ')';
}
```

Wenn nun z. B. `b1` ein Objekt der Klasse `CBruch` ist, dann kann man diesen Bruch wie folgt ausgeben:

```
   cout << b1;
```

Für das Überladen des Operators << gilt Folgendes:

- **Operator << kann nur mittels einer globalen Funktion überladen werden**
  Man kann die Operatorfunktion nicht als Memberfunktion in der entsprechenden Klasse überladen, da dann das Objekt (hier `b1`) der linke Operand sein müsste. Hier soll aber der linke Operand immer das `ostream`-Objekt `cout` und der rechte Operand ein Objekt der eigenen Klasse sein. Folglich ist nur eine globale Operatorfunktion möglich, die außerhalb der Klasse definiert wird und keine Einschränkungen hinsichtlich der Reihenfolge der Operanden aufweist.
- **Rückgabe einer Referenz auf `ostream`**
  Um eine Verkettung mehrerer Ausgaben zu ermöglichen, sollte immer eine Referenz auf das Stream-Objekt (hier: `os`) als Funktionswert zurückgegeben werden. Es ist hier innerhalb der Operatorfunktion kein direkter Zugriff auf die privaten Membervariablen `m_zaehler` und `m_nenner` möglich. Soll ein direkter Zugriff ermöglicht werden, müsste die Funktion `operator<<()` als `friend` in der Klasse `CBruch` deklariert werden.

Programm 3.31 – `ausoverload.cpp`:
Überladen des Operators << für die Klasse CBruch

```
#include <iostream>
using namespace std;
class CBruch {
  public:
    CBruch( int z = 1, int n = 1 ) : m_zaehler(z), m_nenner(n) { }
    int getZaehler(void) const { return m_zaehler; }
    int getNenner(void)  const { return m_nenner;  }
    const CBruch operator+ (const CBruch& b) const {
       CBruch  sum;
       sum.m_zaehler = m_zaehler* b.m_nenner + b.m_zaehler*m_nenner;
       sum.m_nenner  = m_nenner * b.m_nenner;
       sum.kuerzen();
       return sum;
    }
    CBruch& operator+= (const CBruch& b) {
       *this = *this + b; // Aufruf von operator+
       return *this;
    }
  private:
    int m_zaehler, m_nenner;
    int ggT( int n, int m) { return (m==0) ? n : ggT(m, n%m); }
    void kuerzen(void) {
       int ggTeiler = ggT(m_zaehler, m_nenner);
       m_zaehler /= ggTeiler;
       m_nenner  /= ggTeiler;
    }
};
ostream &operator<<(ostream &os, const CBruch &b) {
   return os << '(' << b.getZaehler() << "/" << b.getNenner() << ')';
}
int main(void) {
    CBruch  b1( 1, 4 );  // 1. Bruch (1/4)
    CBruch  b2( 3, 8 );  // 2. Bruch (3/8)
    CBruch  b3;          // 3. Bruch (1/1)

    b1 += b2;
    cout << "b1 = " << b1 << endl;  // Ausgabe von b1
    b3 = b1 + b2;
    cout << "b3 = " << b3 << endl;  // Ausgabe von b3
}
```

Programm 3.31 liefert die folgende Ausgabe:

```
b1 = (5/8)
b3 = (1/1)
```

3.7.2 Stream-Eingabe für eigene Klassen

Auch die Stream-Eingabe kann in C++ auf benutzerdefinierte Klassen erweitert werden. Dazu muss man die Operatorfunktion `operator>>()` global überladen, wie z. B. für die Klasse `CBruch`:

```
istream& operator>>(istream& is, CBruch& b) {
   int  z, n;
   is >> z >> n;
   b.set(z, n);
   return is;
}
```

Wenn nun z. B. `b1` ein Objekt der Klasse `CBruch` ist, dann kann man diesen Bruch wie folgt einlesen:

```
   cin >> b1;
```

Für das Überladen des Operators >> gilt wieder Folgendes:

- **Operator >> kann nur mittels einer globalen Funktion überladen werden**
  Aus den gleichen Gründen, die bereits beim Überladen des Ausgabeoperators >> erläutert wurden,kann auch `operator>>()` nicht als Memberfunktion in der entsprechenden Klasse überladen werden.
- **Rückgabe einer Referenz auf `istream`**
  Um eine Verkettung mehrerer Eingaben zu ermöglichen, sollte auch hier immer eine Referenz auf das Stream-Objekt (hier: `is`) als Funktionswert zurückgegeben werden. Es ist auch hier innerhalb der Operatorfunktion kein direkter Zugriff auf die privaten Membervariablen `m_zaehler` und `m_nenner` möglich. Soll ein direkter Zugriff ermöglicht werden, müsste die Funktion `operator>>()` als `friend` in der Klasse `CBruch` deklariert werden.

Programm 3.32 – `einoverload.cpp`:
Überladen des Operators >> für die Klasse CBruch

```
#include <iostream>
using namespace std;
class CBruch {
 public:
    CBruch( int z = 1, int n = 1 ) : m_zaehler(z), m_nenner(n) { }
    void set(int z, int n) { m_zaehler = z; m_nenner = n; }
    //.... Rest dieser Klasse identisch zu Programm 3.31
};
ostream &operator<<(ostream& os, const CBruch& b) {
   return os << '(' << b.getZaehler() << "/" << b.getNenner() << ')';
}
istream& operator>>(istream& is, CBruch& b) {
   int z, n;
   is >> z >> n;
   b.set(z, n);
   return is;
}
```

```
int main(void) {
    CBruch  b1, b2, b3;

    cout << "1. Bruch: ";   cin >> b1;
    cout << "2. Bruch: ";   cin >> b2;
    b1 += b2;
    cout << "b1 = " << b1 << endl;  // Ausgabe von b1
    b3 = b1 + b2;
    cout << "b3 = " << b3 << endl;  // Ausgabe von b3
}
```

Ein mögliches Ablaufbeispiel zu Programm 3.32 ist:

```
1. Bruch: 1 4 ⏎
2. Bruch: 3 8 ⏎
b1 = (5/8)
b3 = (1/1)
```

3.8 Die Basisklasse streambuf und von ihr abgeleitete Klassen

Der Hauptgrund für die Einführung der Klasse `streambuf` war, die zuvor vorgestellten I/O-Streamklassen unabhängig von dem Gerät zu machen, auf dem die jeweiligen Ein- bzw. Ausgaben erfolgen:

- `streambuf`-Objekte sind eine Zwischenschicht zwischen den I/O-Streamobjekten und den Geräten, auf denen die Ein- bzw. Ausgabe durchgeführt wird. Ein Programm kommuniziert mit den I/O-Streamobjekten, die ihrerseits die entsprechenden Anfoderungen an die zugehörigen `streambuf`-Objekte weiterleiten, welche dann die erforderlichen Aktivitäten auf den zugeordneten Ein-/Ausgabegeräten veranlassen.
- Jedes Streamobjekt beinhaltet intern einen Zeiger auf sein zugehöriges `streambuf`-Objekt.
- Ein `streambuf`-Objekt unterhält intern ein lokales Objekt (für länder- und kulturspezifische Eigenheiten) und bis zu zwei Puffer: für die Ein- und/oder für die Ausgabe. Ein Objekt kann abhängig vom Typ des abgeleiteten `streambuf`-Objekts und vom Öffnungsmodus Zugriff auf einen, auf beide oder aber auch auf keinen dieser beiden Puffer besitzen.
- Um auf ein `streambuf`-Objekt zugreifen zu können, steht jedem I/O-Streamobjekt die von der Klasse `ios` geerbte Methode `rdbuf()` zur Verfügung, die einen Zeiger auf das zugehörige `streambuf`-Objekt liefert. So kann jedes I/O-Streamobjekt über diesen Zeiger auf die Methoden des `streambuf`-Objekts zugreifen, die nicht `private` sind.
- Die Klasse `streambuf` stellt keinen `public`-Konstruktor zur Verfügung, dafür aber eine ganze Reihe von `public`-Methoden.

3.8.1 protected-Methoden von streambuf

Auf die `protected`-Methoden der Klasse `streambuf` kann normalerweise nicht direkt zugegriffen werden. Leitet man aber eine eigene Klasse von `streambuf` ab, so hat man

Zugriff auf diese Methoden. Wie bereits früher erwähnt, hat ein `streambuf`-Objekt bis zu zwei Puffer: einen für die Eingabe- und/oder einen für die Ausgabe. Ein Objekt kann abhängig vom Typ des abgeleiteten `streambuf`-Objekts und vom Öffnungsmodus Zugriff auf einen, auf beide oder aber auch auf keinen dieser beiden Puffer besitzen.
Abbildung 3.3 zeigt wichtige interne Zeiger für diese beiden Puffer, die bei der Vorstellung der `protected`-Methoden der Klasse `streambuf` benötigt werden. *g* beim Eingabepuffer steht dabei für *get* und *p* beim Ausgabepuffer für *put*.

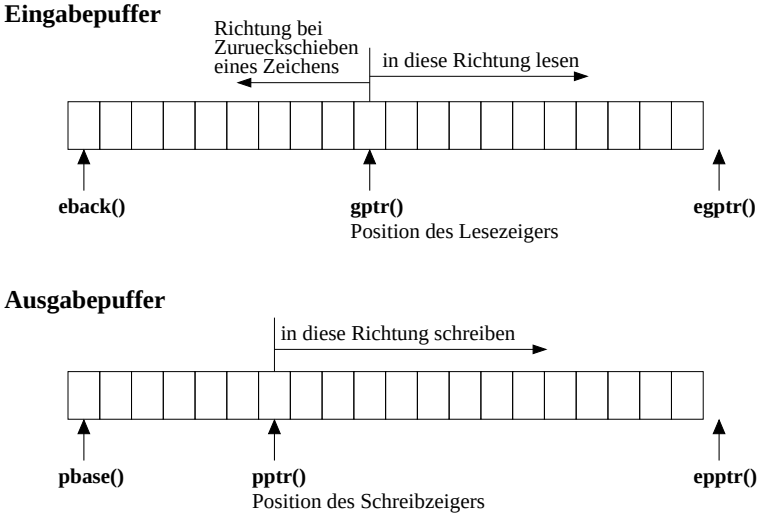


Abbildung 3.3: Interne Zeiger der Ein- und Ausgabepuffer von streambuf-Objekten

protected-Konstruktor

`streambuf()`

legt ein `streambuf`-Objekt an und initialisiert alle seine Zeiger mit `0`.

protected-Methoden für die Eingabe

In der folgenden Liste sind die virtuellen Methoden, die man in einer von `streambuf` abgeleiteten Klasse überschreiben kann, mit ^{V} gekennzeichnet.

- Erfragen der internen Eingabepuffer-Zeiger

```
char *eback() const
char *gptr() const
char *egptr() const
```

- Aktuellen „Lesezeiger“ um n Zeichen weitersetzen

```
void gbump(int n)
```

- Explizites Setzen aller internen Eingabepuffer-Zeiger

```
void setg(char *a, char *n, char *e)
```

```
eback = a; // Anfangsposition
gptr  = n; // n muss mind. a+1 sein (für Zurückschieben eines Zeichens)
egptr = e; // zeigt auf 1. Zeichen direkt hinter dem Eingabepuffer
```

- Lesen von n Zeichen aus dem Eingabepuffer

  `streamsize xsgetn(char *s, streamsize n)` ^{V}

  schreibt die gelesenen Zeichen an die Adresse `s`, wobei kein 0-Zeichen angehängt wird. Beim Lesen von `EOF` wird das Lesen vorzeitig abgebrochen. Die Anzahl der wirklich gelesenen Zeichen wird zurückgegeben.

- Leere Eingabepuffer

  `int underflow()` ^{V}

  wird von der `public`-Methode `sgetc()` aufgerufen, wenn `gptr()==0` (kein Eingabepuffer) oder `gptr()>=egptr()` (leerer Eingabepuffer) gilt.

  Bei `gptr()!=0` und `gptr()<egptr()` wird das Zeichen zurückgegeben, auf das `gptr()` zeigt. Ansonsten fordert diese Methode vom Eingabegerät neue Zeichen an, um den Eingabepuffer zu füllen und dann das erste Zeichen zurückzuliefern. Ist diese Anforderung nicht erfolgreich, wird `EOF` zurückgegeben.

  `int uflow()` ^{V}

  ruft `underflow()` auf und gibt das von dieser Methode gelieferte Zeichen (eventuell `EOF`) zurück. Anders als `underflow()` setzt `uflow()` den „Lesezeiger" `gptr` um eine Position weiter.

- Erfragen der noch verfügbaren Zeichen im Eingabepuffer

  `int showmanyc()` ^{V}

  liefert die Anzahl der Zeichen, die noch aus dem Eingabepuffer gelesen werden können. Gibt diese Methode einen positiven Wert zurück, liefert der nächste Aufruf von `uflow()` oder `underflow()` kein `EOF` zurück.

- Zurückschieben eines Zeichens in den Eingabepuffer

  `int pbackfail(int c=EOF)` ^{V}

  schreibt das Zeichen `c` zurück in den Eingabepuffer, so dass dieses beim nächsten Lesevorgang als nächstes gelesen wird. Ist `c` gleich `EOF`, so wird automatisch das zuletzt aus dem Eingabepuffer gelesene Zeichen zurückgeschoben. Diese Methode wird von den `public`-Methoden `sungetc()` und `sputbackc()` aufgerufen, wenn diese nicht erfolgreich ausgeführt werden konnten, was in folgenden Situationen zutrifft:

  - `gptr()==0`: ungepufferte Eingabe
  - `gptr()==eback()`: Eingabepuffer ist leer
  - `gptr()!=c`: letztes gelesene Zeichen ist verschieden von `c`

  Als Rückgabewert liefert diese Methode bei Erfolg einen beliebigen Wert, und ansonsten `EOF`.

protected-Methoden für die Ausgabe

In der folgenden Liste sind die virtuellen Methoden, die man in einer von `streambuf` abgeleiteten Klasse überschreiben kann, wieder mit ^{V} gekennzeichnet.

- Erfragen der internen Ausgabepuffer-Zeiger

  `char *pbase() const`

  `char *pptr() const`

  `char *epptr() const`

- Aktuellen „Schreibzeiger“ um n Zeichen weitersetzen

  ```
  void pbump(int n)
  ```

- Explizites Setzen aller internen Ausgabepuffer-Zeiger

  ```
  void setp(char *a, char *e)
  ```

  ```
  pbase = pptr = a; // Anfangsposition und aktuelle Schreibzeigerposition
  epptr = e;        // zeigt auf 1. Zeichen direkt hinter dem Ausgabepuffer
  ```

  Möchte man ungepufferte Ausgabe festlegen, so dass jede Ausgabe sofort auf das entsprechende Gerät geschrieben wird, muss man für die beiden Argumente hier 0 angegeben. In diesem Fall wird für jedes auszugebendes Zeichen die Methode `overflow()` aufgerufen.

- Schreiben von n Zeichen in den Ausgabepuffer

  ```
  streamsize xsputn(char *s, streamsize n)
  ```
  ^{V}

  schreibt `n` Zeichen von der Adresse `s` in den Ausgabepuffer. Als Rückgabewert liefert sie die Anzahl der wirklich geschriebenen Zeichen.

- Voller Ausgabepuffer

  ```
  int overflow(int c=EOF)
  ```

  schreibt den Inhalt des Ausgabepuffers plus das Zeichen `c` (wenn es nicht `EOF` ist) auf das entsprechende Zielgerät. Bei Erfolg liefert sie einen beliebigen Wert zurück, und ansonsten `EOF`.

Weitere protected-Methoden

In der folgenden Liste sind die virtuellen Methoden, die man in einer von `streambuf` abgeleiteten Klasse überschreiben kann, wieder mit ^{V} gekennzeichnet.

- Relatives Positionieren im Ein- bzw. Ausgabepuffer

  ```
  streampos seekoff(streamoff offset, seekdir wie,
                    openmode mode=in|out)
  ```
  ^{V}

  Diese Methode muss man in einer abgeleiteten Klasse überschreiben, um ein relatives Positionieren des „Schreibzeigers“ (`mode=out`) und/oder des „Lesezeigers“ (`mode=in`) um `offset` Bytes abhängig vom `wie`-Argument zu ermöglichen:

  - `ios::beg`: von Anfang an
  - `ios::cur`: ab aktueller Position
  - `ios::end`: von Ende an

- Absolutes Positionieren im Ein- bzw. Ausgabepuffer

  ```
  streampos seekpos(streampos offset, openmode mode=in|out)
  ```
  ^{V}

  Diese Methode muss man in einer abgeleiteten Klasse überschreiben, um ein absolutes Positionieren des „Schreibzeigers“ (`mode=out`) und/oder des „Lesezeigers“ (`mode=in`) auf die Position `offset` zu ermöglichen.

➤ Neuen Puffer für streambuf-Objekt festlegen

`streambuf *setbuf(char *puffer, streamsize n)` ^{V}

Diese Methode muss man in einer abgeleiteten Klasse überschreiben, wenn man es ermöglichen will, dass man für ein `streambuf`-Objekt `puffer` einen neuen Puffer der Länge `n` festlegen kann. Angabe von `NULL` für `puffer` oder `0` für `n` bedeutet ungepuffert. Bei erfolgreicher Ausführung dieser Methode, liefert sie `this` und ansonsten einen `NULL`-Zeiger.

➤ Synchronisieren von Ein-/Ausgabepuffer

`int sync()` ^{V}

Diese Methode kann man in einer abgeleiteten Klasse überschreiben, um dem Ausgabepuffer zu leeren („*flushen*“), also auf das entsprechende Gerät zu schreiben, oder um das Eingabegerät auf das zuletzt verarbeitete Zeichen zurückzusetzen. Die Voreinstellung ist, dass keine Pufferung stattfindet und dass 0 zurückgegeben wird, um anzuzeigen, dass diese Methode erfolgreich ausgeführt wurde. Die Rückgabe von -1 deutet auf einen Fehler hin.

3.8.2 public-Methoden von streambuf

Bei den nachfolgend vorgestellten `public`-Methoden wird mittels eines Pfeils angezeigt, welche `protected`-bzw. andere `public`-Methoden diese eventuell bei Bedarf aufrufen.

public-Methoden für die Eingabe

➤ Verfügbare Zeichenzahl im Eingabepuffer

`streamsize in_avail()` → *`showmanyc()`*

liefert die Anzahl der Zeichen, die aktuell aus dem Eingabepuffer gelesen werden können. Sind aktuell keine Zeichen eim Eingabepuffer vorhanden, wird die `protected`-Methode `showmanyc()` aufgerufen.

➤ Lesen des nächsten Zeichens

`int sgetc()` → *`underflow()`*
`int sbumpc()` → *`uflow()`*

liefern beide das nächste Zeichen aus dem Eingabepuffer, auf das der „Lesezeiger“ zeigt, oder `EOF`, wenn dieser am Pufferende steht. In letzterem Fall werden die virtuellen `protected`-Methoden `underflow()` bzw. `uflow()` aufgerufen. Während `sgetc()` das Zeichen im Puffer belässt, wird dieses bei `sbumpc()` aus dem Puffer entfernt.

➤ Lesen des übernächsten Zeichens

`int snextc()` → *`sbumpc()`* → *`sgetc()`*

entfernt zunächst das Zeichen aus dem dem Eingabepuffer, auf das der „Lesezeiger“ aktuell zeigt, und liefert dann das folgende Zeichen, das allerdings nicht aus dem Puffer entfernt wird. Sollte sich der ursprüngliche bzw. der weitergesetzte „Lesezeiger“ am Pufferende befinden, wird `EOF` zurückgegeben. Dies wird dadurch erreicht, dass `snextc()` die Methode `sbumpc()` aufruft, und wenn

`sbumpc()` `EOF` zurückgibt, gibt auch `snextc()` `EOF` zurück, anderfalls wird das Zeichen zurückgegeben, das `sgetc()` liefert.

- Lesen mehrerer Zeichen

  `streamsize sgetn(const char *s, streamsize n)` → *`xsgetn()`*
  liest mittels Aufruf der `protected`-Methode `xsgetn()` `n` Zeichen aus dem Eingabepuffer und schreibt diese in den Puffer, dessen Anfangsadresse `s` hier übergeben wird. Wie viele Bytes wirklich gelesen wurden, liefert diese Methode als Rückgabwert.

- Zuletzt gelesene Zeichen zurückschieben

  `int sungetc()` → *`pbackfail()`*
  setzt den „Lesezeiger“ eine Position zurück. Ist ein solches Zurückschieben erfolgreich, wird das Zeichen an der neuen Position des „Lesezeigers“ (`*gptr()`) zurückgegeben, ansonsten wird der Rückgabewert von `pbackfail()` zurückgegeben.

- Bestimmtes letztes Zeichen zurückschieben

  `int sputbackc(char c)` → *`pbackfail()`*
  versucht, das Zeichen `c` in den Eingabepuffer zurückzuschieben, so dass beim nächsten Lesen dieses Zeichen gelesen wird. Ein erfolgreiches Zurückschieben ist allerdings nur möglich, wenn `gptr() > ebase()` ist und das Zeichen vor dem „Lesezeiger“ (`*(gptr()-1)`) gleich dem übergebenen Zeichen `c` ist. In diesem Fall wird das Zeichen `c` zurückgegeben. Ist ein Zurückschieben nicht möglich, wird wird der Rückgabewert von `pbackfail()` zurückgegeben.

Programm 3.33 – `streambuf1.cpp`:
Demoprogramm zu public-Eingabemethoden der Klasse streambuf

```
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char *argv[])
{
  char z;
  long n;
  //.............................. Auf Kdozeile angegebene Datei ausgeben
  cout << "------------------------------------------------------------" << endl;
  ifstream datei(argv[1]);
  streambuf *pufZgr = datei.rdbuf();
  do {
     z = pufZgr->sgetc();
     cout << z;
  } while (pufZgr->snextc()!=EOF); // ein Zeichen überlesen
  datei.close();
  cout << "------------------------------------------------------------" << endl;
  //....................... Aus eingegebenen Text eine Zahl herausfiltern
  pufZgr = cin.rdbuf();
  cout << "Gib ein paar Buchstaben gefolgt von einer Zahl ein: ";
```

```
  do {
    if (isdigit(z=pufZgr->sbumpc())) {
       pufZgr->sungetc();
       cin >> n;
       cout << "Die Zahl war: " << n << endl;
       break;
    }
  } while (z != EOF);
}
```

Ein mögliches Ablaufbeispiel zu Programm 3.33:

```
-------------------------------------------------------
Ausgabe der Datei, die auf Kommandozeile angegeben ist
-------------------------------------------------------
Gib ein paar Buchstaben gefolgt von einer Zahl ein: Das Parfum:---4711 ⏎
Die Zahl war: 4711
```

public-Methoden für die Ausgabe

- Schreiben eines Zeichens

  `int sputc(char c)` → *`overflow()`*

  schreibt das Zeichen `c` in den Ausgabepuffer und positioniert den „Schreibzeiger“ um eine Position weiter.

  Bei erfolgreichem Schreiben liefert diese Methode das geschriebene Zeichen `c` als Rückgabewert.

  Bei nicht erfolgreichem Schreiben (z. B. weil der Ausgabepuffer voll ist), wird die `protected`-Methode `overflow()` aufgerufen.

- Schreiben mehrerer Zeichen

  `streamsize sputn(const char *s, streamsize n)` → *`xsputn(s,n)`*

  schreibt mittels Aufruf der `protected`-Methode `xsputn()` `n` Zeichen aus dem Puffer `s` in den Ausgabepuffer und positioniert den „Schreibzeiger“ um entsprechend viele Positionen weiter.

  Wie viele Bytes wirklich geschrieben wurden, liefert diese Methode als Rückgabwert.

- Physikalisches Schreiben des Pufferinhalts auf das Gerät

  `int pubsync()` → *`sync()`*

  leert („*flushed*“) Ausgabepuffer durch Aufruf der `protected`-Methode `sync()`, indem sie dessen Inhalt auf das Ausgabemedium schreiben lässt.

  Bei Fehler gibt diese Methode -1, und sonst einen anderen Wert zurück.

Programm 3.34 – `streambuf2.cpp`:
Demoprogramm zu public-Ausgabemethoden der Klasse streambuf

```
#include <fstream>
#include <iostream>
using namespace std;

int main(void)
{
   char z, text[]= "Dies ist ein Text\n";

   ofstream datei("test.txt");
   streambuf *pufZgr = datei.rdbuf();

   //............................. Text in Datei schreiben
   pufZgr->sputn(text, sizeof(text)-1);

   //............ Text bis # einlesen und in Datei schreiben
   do { pufZgr->sputc(z = cin.get()); } while (z != '#');
   datei.close();
}
```

Ein mögliches Ablaufbeispiel zu Programm 3.34:

```
Nun ich muss wohl ⏎
hier noch etwas Text eingeben ⏎
Jetzt reicht es mir aber# ⏎
```

Die Datei `test.txt` hat danach folgenden Inhalt:

```
Dies ist ein Text
Nun ich muss wohl
hier noch etwas Text eingeben
Jetzt reicht es mir aber#
```

Weitere public-Methoden

- Relatives und absolutes Positionieren im Ein- bzw. Ausgabepuffer

  ```
  streampos pubseekoff(streamoff offset, seekdir wie,
                       openmode mode=in|out)
  ```
  → *return seekoff(offset, wie, mode)*

  ```
  streampos pubseekpos(streampos offset,
                       openmode mode=in|out)
  ```
  → *return seekpos(offset, mode)*

- Neuen Puffer für streambuf-Objekt festlegen

  ```
  streambuf *pubsetbuf(char *puffer, streamsize n)
  ```
  → *return setbuf(puffer, n)*

Programm 3.35 – `streambuf3.cpp`:
Demoprogramm zu den weiteren public-Methoden der Klasse streambuf

```
#include <fstream>
#include <iostream>
#include <iomanip>
using namespace std;

int main(void)
{
   long     groesse;
   filebuf *sbufZgr;

   //.....................................................pubseekoff
   fstream datei("test.txt", ios::out);
   datei << "Der erste  Satz" << endl;
   sbufZgr = datei.rdbuf();
   groesse = sbufZgr->pubseekoff(0,ios::end); // auf Dateiende positionieren
   datei.close();
   cout << "Grösse der Datei 'test.txt' ist: " << groesse << endl;

   //.....................................................pubseekpos
   datei.open("test.txt", ios::out|ios::in);
   sbufZgr = datei.rdbuf();
   sbufZgr->pubseekpos(4);       // Schreibzeiger auf 5. Zeichen in Datei
   sbufZgr->sputn("zweite", 6);  // 'erste' durch 'zweite' ändern
   sbufZgr->pubseekoff(0,ios::end); // auf Dateiende positionieren
   datei << "und noch eine Zeile" << endl;
   datei.close();

   //.....................................................pubsetbuf
   char puffer[128], z;
   datei.open("test.txt", ios::in);
   datei.rdbuf()->pubsetbuf(puffer, 128); // neuen Eingabepuffer festlegen
   sbufZgr = datei.rdbuf();
   sbufZgr->pubseekpos(0);    // Schreibzeiger auf 1. Zeichen in Datei
   datei >> z; // ein Zeichen lesen, um neuen Eingabepuffer zu füllen
   puffer[sbufZgr->pubseekoff(0,ios::end)] = 0; // puffer mit 0 terminieren
   cout << "-------------Inhalt der Datei 'test.txt:" << endl;
   cout << puffer; // Pufferinhalt (Dateiinhalt) ausgeben
   datei.close();
}
```

Programm 3.35 liefert die folgende Ausgabe, wobei die letzten beiden Zeilen den Inhalt der Datei `test.txt` nach dem Ablauf dieses Programms zeigen:

```
Grösse der Datei 'test.txt' ist: 16
-------------Inhalt der Datei 'test.txt:
Der zweite Satz
und noch eine Zeile
```

3.8.3 Die Klasse filebuf – Pufferklasse für Dateien

Die Klasse `filebuf` ist von der Klasse `streambuf` abgeleitet und verfügt somit über die gleichen Fähigkeiten wie die zuvor vorgestellte Klasse `streambuf`, nur dass die Klasse `filebuf` mit Ein- und/oder Ausgabepuffer ausgestattet ist, die für das Lesen und Schreiben in Dateien ausgelegt sind.
Um die Klasse `filebuf` benutzen zu können, muss man `<fstream>` inkludieren:

```
#include <fstream>
```

Neben den `public`-Methoden, die `filebuf` von der Klasse `streambuf` erbt, stellt die Klasse `filebuf` noch folgende `public`-Methoden zur Verfügung:

`filebuf()`
: Dieser Konstruktor legt ein `filebuf`-Objekt an, das die geerbten Membervariablen mittels Aufruf des Konstruktors der Basisklasse `streambuf` initialisiert und die internen Dateizeiger auf `0` setzt.

`bool is_open()`
: liefert `true`, wenn das `filebuf`-Objekt mit einer offenen Datei verbunden ist, und ansonsten `false`.

`filebuf *open(const char *name, openmode mode)`
: verbindet das `filebuf`-Objekt mit der Datei `name`, wobei die Datei mit dem Modus `mode` geöffnet wird. Diese Methode liefert bei Erfolg `this` zurück, und ansonsten einen `0`-Zeiger.

`filebuf *close()`
: schliesst eine offene Datei, wobei zuvor der Ausgabepuffer dorthin geschrieben wird. Diese Methode liefert bei Erfolg `this` zurück, und ansonsten einen `0`-Zeiger.

Programm 3.36 – `filebuf.cpp`:
Demoprogramm zu den public-Methoden der Klasse filebuf

```
#include <iostream>
#include <fstream>
using namespace std;
int main(void) {
 ofstream  ausDatei;
 char *text = "Dieser Text wird in Datei 'test.txt' geschrieben";
  filebuf  *fb = ausDatei.rdbuf();
  cout << boolalpha << "mit Datei verbunden: " << fb->is_open() << endl;
  fb->open("test.txt", ios::out);
  cout << "mit Datei verbunden: " << fb->is_open() << endl;
  fb->sputn(text, strlen(text)); // schreibt Text in Datei 'test.txt'
  fb->close();
}
```

Programm 3.36 schreibt eine Zeile in die Datei `test.txt` und gibt Folgendes aus:

```
mit Datei verbunden: false
mit Datei verbunden: true
```

3.8.4 Die Klasse stringbuf – Pufferklasse für Strings

Die Klasse `stringbuf` ist von der Klasse `streambuf` abgeleitet und verfügt somit über die gleichen Fähigkeiten wie diese Klasse, nur dass die Klasse `stringbuf` mit Ein- und/oder Ausgabepuffer ausgestattet ist, die für das Lesen und Schreiben in Strings ausgelegt sind. Um die Klasse `stringbuf` benutzen zu können, muss man `<sstream>` inkludieren:

```
#include <sstream>
```

Neben den `public`-Methoden, die `stringbuf` von der Klasse `streambuf` erbt, stellt die Klasse `stringbuf` noch folgende `public`-Methoden zur Verfügung:

`stringbuf(openmode mode=in|out)`

`stringbuf(const string& str, openmode mode=in|out)`

Diese Konstruktoren legen ein `stringbuf`-Objekt an, das die geerbten Membervariablen mittels Aufruf des Konstruktors der Basisklasse `streambuf` initialisiert und die internen Dateizeiger auf `0` setzt. Über Parameter `mode` wird festgelegt, ob aus diesem `stringbuf`-Objekt gelesen (`in` oder in es geschrieben (`out`) werden kann. Der zweite Konstruktor initialisiert den entsprechenden Puffer mit den String `str`.

`string str() const`

liefert eine Kopie des Strings im entsprechenden Puffer zurück.

`void str(string& s)`

initialisiert den entsprechenden Puffer mit den String `s`.

Programm 3.37 – `stringbuf.cpp`:
Demoprogramm zu den public-Methoden der Klasse stringbuf

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

int main(void)
{
   stringbuf sb;
   string string1 = "Das ist der erste String",
          string2 = "der zweite String";

   sb.sputn(string1.c_str(), string1.size());
   cout << sb.str() << endl;

   sb.str(string2);
   cout << sb.str() << endl;
}
```

Programm 3.37 liefert die folgende Ausgabe:

```
Das ist der erste String
der zweite String
```

3.8.5 streambuf- und stream-Objekte

Nachfolgend werden einige Punkte aufgezählt, die beim gleichzeitigen Arbeiten mit `streambuf`- und `stream`-Objekten zu beachten sind:

- Die den jeweiligen Streams zugeordneten `streambuf`-Objekte sind keine `ifstream`/`istringstream`- oder `ofstream`/`ostringstream`-Objekte, sondern `istream` bzw. `ostream`-Objekte.
- Ein `streambuf`-Objekt muss nicht in einem `ifstream`- oder `ofstream`-Objekt definiert werden, sondern kann auch getrennt für sich allein definiert werden, wie z. B.:

```
filebuf puffer("dateiname", ios::in | ios::out | ios::trunc);
istream eingabe(&puffer);
ostream ausgabe(&puffer);
```

3.9 Fortgeschrittenere Techniken

3.9.1 Zentrale Fehlermeldungsroutine

In größeren Softwareprojekten ist es üblich, dass man eine zentrale Fehlermeldungsroutine zur Verfügung stellt, die die anderen Module zur Ausgabe von Fehler- oder eventuell auch Diagnosemeldungen aufrufen.
Programm 3.38 zeigt eine mögliche Headerdatei und Programm 3.39 eine mögliche Realisierung zu einer solchen zentralen Fehlermeldungsroutine.

Programm 3.38 – `fehler.h`:
Klassendeklaration zu einer zentralen Fehlermeldungsroutine

```
#ifndef FEHLER_H
#define FEHLER_H
#include <iostream>

#define  WARNUNG      0   /*--- Kennungen fuer unterschiedl. Fehlerarten */
#define  WARNUNG_SYS  1
#define  FATAL        2
#define  FATAL_SYS    3

#define  MAX_ZEICHEN   4096    /*--- Maximale Pufferlaenge */

class fehlmeldstream : public std::ostream {
  public:
     fehlmeldstream(std::ostream &os) : std::ostream(os.rdbuf()) { }
     void fehler_meld(int kennung, const char *fmt, ...);
  private:
     char           puffer[MAX_ZEICHEN];

     void fehl_meldung(int sys_meld, const char *fmt, va_list az);
};

#endif
```

Programm 3.39 – `fehler.cpp`:
Implementierung einer zentralen Fehlermeldungsroutine

```
#include  <cerrno>
#include  <cstdarg>
#include  "fehler.h"
using namespace std;
/*------------------------ Lokale Routinen zur Abarbeitung der Argumentliste -*/
void fehlmeldstream::fehl_meldung(int sys_meld, const char *fmt, va_list az) {
  int     n, fehler_nr = errno;
  ostream output(rdbuf());
   memset(puffer, 0, MAX_ZEICHEN);
   n = vsnprintf(puffer, MAX_ZEICHEN, fmt, az);
   if (sys_meld)
      snprintf(puffer+n, MAX_ZEICHEN-n, ": %s ", strerror(fehler_nr));
   output << puffer << endl;
}
/*---------------------------------------- Global aufrufbare Fehlerroutine -*/
void fehlmeldstream::fehler_meld(int kennung, const char *fmt, ...) {
  va_list  az;
   va_start(az, fmt);
   if (kennung == WARNUNG || kennung == FATAL)
      fehl_meldung(0, fmt, az);
   else if (kennung == WARNUNG_SYS || kennung == FATAL_SYS)
      fehl_meldung(1, fmt, az);
   else {
      fehl_meldung(1, "Falscher fehler_meld-Aufruf", az);
      exit(2);
   }
   va_end(az);
   if (kennung==FATAL || kennung==FATAL_SYS)
      exit(1);
}
```

Das erste Argument von `fehler_meld()` legt fest, wie der entsprechende Fehler zu behandeln ist.
Es sind die folgenden in `fehler.h` definierten Konstanten als erstes Argument erlaubt:

`WARNUNG`	`WARNUNG_SYS`
`FATAL`	`FATAL_SYS`

Es wurde hier folgende Regelung bei der Vergabe der Konstanten-Namen getroffen:

- Die Endung `SYS` bedeutet, dass zusätzlich zur eigenen Meldung noch die zum entsprechenden Fehler gehörige System-Fehlermeldung auszugeben ist.
- Nur bei den `WARNUNG`-Konstanten bewirkt die Fehlerroutine `fehler_meld()` nicht die Beendigung des gesamten Programms (kein `exit()`-Aufruf).
- Bei den `FATAL`-Konstanten bewirkt die Fehlerroutine `fehler_meld()` einen Programmabbruch mittels `exit(1)`.

Die weiteren Argumente zu `fehler_meld()` entsprechen denen bei `printf()`. Jedes andere Modul, das die hier angebotene Funktion `fehler_meld()` aufrufen will, muss dann ein `#include “fehler.h“` beinhalten.

Das Programm 3.40 ermittelt die Häufigkeit des Vorkommens jedes einzelnen Buchstabens (Groß- und Kleinbuchstaben werden nicht unterschieden) in den auf der Kommandozeile angegebenen Dateien. Um Fehler- und Diagnosemeldungen auszugeben, ruft es die Fehlerroutine `fehler_meld()` auf.

Programm 3.40 – `buchstat.cpp`:
Testprogramm zur eigenen Fehlerroutine

```
#include  <cctype>
#include  <iomanip>
#include  <iostream>
#include  <fstream>
#include  "fehler.h"
using namespace std;

int  main(int argc, char *argv[]) {
   char           zeich;
   long int       buchst[26]={0L}, i, z=0;
   fehlmeldstream fehl(cerr);
   ifstream       datei;

   if (argc < 2)
      fehl.fehler_meld(FATAL, "....usage:  %s datei(en)", argv[0]);

   for (i=1; i<argc; i++) {
      datei.clear();
      datei.open(argv[i]); // Datei argv[i] öffnen
      if (!datei)
         fehl.fehler_meld(WARNUNG_SYS, "....kann ‘%s’ nicht öffnen", argv[i]);
      else {
         do {
            datei.get(zeich);
            if (datei.bad()) {
               fehl.fehler_meld(WARNUNG_SYS, "....Lesefehler in ‘%s’", argv[i]);
               break;
            } else if (isalpha(zeich))
               buchst[tolower(zeich)-’a’]++;
         } while (!datei.eof());
         z++;
         datei.close();
      }
   }
   if (z > 0)
      for (i=’a’; i<=’z’; i++)
         cout << setw(10) << char(i) << " | " << setw(7) << buchst[i-’a’] << "|"
              << ((i%3==0) ? "\n" : "   ");
   cout << endl << endl;
   fehl.fehler_meld(WARNUNG, "....%d von %d Dateien ausgewertet", z, argc-1);
}
```

Nachdem dieses Programm z. B. mit

g++ -o buchstat buchstat.cpp fehler.cpp
kompiliert und mit dem Modul `fehler.cpp` zusammen gelinkt wurde, könnten sich folgende Abläufe ergeben:

```
./buchstat ⏎
....usage:  buchstat datei(en)
./buchstat  *  aaa.b ⏎
....kann 'aaa.b' nicht öffnen: No such file or directory
         a |    1851|            b |     689|            c |    1568|
         d |    1331|            e |    3599|            f |     871|
         g |    1020|            h |     792|            i |    2839|
         j |      74|            k |     165|            l |    1378|
         m |     686|            n |    2733|            o |    1391|
         p |     644|            q |      19|            r |    2115|
         s |    2693|            t |    2999|            u |    1334|
         v |     267|            w |     200|            x |     212|
         y |      81|            z |     508|

....76 von 77 Dateien ausgewertet
```

3.9.2 Exceptions in Streamklassen

Da die I/O-Stream-Bibliothek von C++ bereits entworfen wurde, bevor Exceptions in C++ eingeführt wurden, treten normalerweise keine Exceptions in den Klassen I/O-Stream-Bibliothek auf. Dieses voreingestellte Verhalten lässt sich jedoch mit `ios::exceptions()` ändern. Diese Methode wird in zwei überladenen Versionen angeboten:

- `void exceptions(iostate state)`
  schaltet für die im Zustand `state` angegebenen Zustandsflags das Schicken von Exceptions ein.
- `iostate exceptions() const`
  liefert die Zustandsflags zurück, für die im entsprechenden Stream Exceptions auftreten können.

Für Exceptions in Streamklassen ist noch Folgendes zu beachten:

- Von Streamklassen geschickte Exceptions sind Objekte der Klasse `ios::failure`, die von der Klasse `ios::exception` abgeleitet ist. Die zu einem geschickten `failure`-Objekt gehörige Meldung kann man mit der Methode `what()` erfragen.
- Exceptions sollten bei Streamklassen nur für Fehlersituationen eingeschaltet werden, und nicht z. B. für `ios::eofbit`, da das Erreichen eines Dateieendes ja eigentlich kein Fehler ist, sondern eine normale Situation.

Programm 3.41 ist ein Demonstrationsbeispiel zu Exceptions bei Streamklassen.

Programm 3.41 – `streamexcept.cpp`:
Demonstrationsbeispiel zu Exceptions bei Streamklassen

```
#include <iostream>
#include <string>


using namespace::std;
```

```
class MeinFehler : public ios::failure {
  public:
     MeinFehler(const string& txt = "") : ios::failure(txt) {}
};

int main(void) {
   double  zahl1, zahl2;
   string  zeile;
   ios_base::iostate altFlags = cin.exceptions();
   cin.exceptions(ios::failbit); // Exceptions für ios::failbit einschalten
   do {
      try {
         cout << "Gib eine 1. Zahl ein: ";
         cin >> zahl1;
         cout << "    Eingegeben wurde: " << zahl1 << endl;
      } catch (const ios::failure& fehler) {
         cerr << "..." << fehler.what() << endl;
         cin.clear(); // Fehlerflag löschen
         getline(cin, zeile); // ganzen Eingabepuffer für neue Eingabe leeren
      }
   } while (zahl1 != 0);
   cin.exceptions(altFlags); // Exceptions für ios::failbit wieder ausschalten
   do {
      try {
         cout << "Gib eine weitere Zahl ein: ";
         cin >> zahl2;
         if (!cin)
            throw MeinFehler("Falsche Eingabe (habe Zahl erwartet)");
         cout << "    Eingegeben wurde: " << zahl2 << endl;
      } catch (const MeinFehler& fehler) {
         cerr << "..." << fehler.what() << endl;
         cin.clear(); // Fehlerflag löschen
         getline(cin, zeile); // ganzen Eingabepuffer für neue Eingabe leeren
      }
   } while (zahl2 != 0);
}
```

Ein möglicher Ablauf des Programms 3.41 wäre z. B.:

```
Gib eine 1. Zahl ein: hallo ⏎
...basic_ios::clear(iostate) caused exception
Gib eine 1. Zahl ein: 2.34 ⏎
    Eingegeben wurde: 2.34
Gib eine 1. Zahl ein: 0 ⏎
    Eingegeben wurde: 0
Gib eine weitere Zahl ein: wie gehts ⏎
...Falsche Eingabe (habe Zahl erwartet)
Gib eine weitere Zahl ein: 3.1415 ⏎
    Eingegeben wurde: 3.1415
Gib eine weitere Zahl ein: 0 ⏎
    Eingegeben wurde: 0
```

3.9.3 Gleichzeitiges Lesen und Schreiben in einer Datei

Öffnen sowohl einer existierenden wie auch nicht existierenden Datei

Um eine Datei gleichzeitig zum Lesen und Schreiben zu öffnen, bietet sich die folgende Anweisung an:

```
fstream datei("dateiname", ios::out | ios::in | ios::binary);
```

Das Problem ist hier, dass die Datei existieren muss, sonst kann sie in diesem Fall nicht erfolgreich geöffnet werden. Existiert eine Datei nämlich nicht, muss sie hier mit der folgenden Flag-Kombination geöffnet werden:

```
ios::out | ios::trunc | ios::binary | ios::in
```

Das Setzen des Flags `ios::binary` ist nur unter Windows/DOS-Systemen erforderlich, wenn man in Dateien den „Schreib-/Lesezeiger“ explizit versetzt. Um nun unabhängig davon, ob eine Datei existiert oder nicht, diese in jedem Fall erfolgreich zu öffnen, kann man die in Programm 3.42 gezeigte Technik anwenden.

Programm 3.42 – `readwrite1.cpp`:
Öffnen einer Datei zum gleichzeitigen Lesen und Schreiben

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
int main(void) {
    //... 1. Versuch, Datei zum Lesen und Schreiben zu öffnen
   fstream datei("test.txt", ios::out | ios::in | ios::binary);
   if (!datei) { // schlägt fehl, wenn Datei nicht existiert
      datei.clear(); // alle Fehlerflags löschen
       // 2. Versuch: mit ios::trunc (erfolgreich, wenn Datei nicht existiert)
      datei.open("test.txt", ios::out | ios::trunc | ios::binary | ios::in);
      cout << "... Nicht existierende Datei 'test.txt' geöffnet" << endl;
   } else
      cout << "... Existierende Datei 'test.txt' geöffnet" << endl;
   if (!datei) {
      cerr << "Fehler (z.B. keine Schreibrechte oder Platte voll)" <<endl; exit(1);
   }
   datei << "Dies ist eine Zeile" << endl; // Schreiben in Datei
   cout << "Datei 'test.txt' hat " << datei.tellp() << " Zeichen" << endl;
   string s;
   datei.seekg(0);    // Lesezeiger auf Anfang positionieren
   getline(datei, s); // und erste Zeile lesen
   cout << "Habe gelesen: " << s << endl;
}
```

Programm 3.42 liefert die folgende Ausgabe, wobei statt „`Existierende`“ in der ersten Zeile eventuell „`Nicht existierende`“ ausgegeben wird:

```
... Existierende Datei 'test.txt' geöffnet
Datei 'test.txt' hat 20 Zeichen
Habe gelesen: Dies ist eine Zeile
```

Schreib/Lesezeiger wird beim Lesen und beim Schreiben weiterbewegt

Nachdem eine Datei zum gleichzeitigen Lesen und Schreiben geöffnet wurde, ist Lesen bzw. Schreiben in ihr mit den entsprechenden Methoden und den Operatoren >> und << möglich. In diesem Fall wird der Schreib/Lesezeiger sowohl beim Schreiben wie beim Lesen entsprechend weiterbewegt. So können z. B. beim Schreiben danach stehende Daten überschrieben werden, wie es in Programm 3.43 gezeigt wird.

Programm 3.43 – `readwrite2.cpp`:
Schreib/Lesezeiger wird beim Lesen und beim Schreiben weiterbewegt

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
int main(void) {
   string  wort1, wort2, zeile1, zeile2;
   fstream datei("test.txt", ios::out | ios::in | ios::binary);
   if (!datei) {
      datei.clear();
      datei.open("test.txt", ios::out | ios::trunc | ios::binary | ios::in);
   }
   datei >> wort1;  // Ein Wort (aus 1. Zeile) einlesen und
   datei << " war"; // nach dem aktuellen Schreib/Lesezeiger Text einfügen
   datei.seekg(0);        // Lesezeiger auf Anfang positionieren
   getline(datei, zeile1); // und erste Zeile lesen
   datei >> wort2;          // Ein Wort (aus 2. Zeile) einlesen und
   datei << " war einmal "; // nach aktuellem Schreib/Lesezeiger Text einfügen
   cout << "wort1: " << wort1 << endl
        << "wort2: " << wort2 << endl;
   datei.seekg(0);  // Lesezeiger auf Anfang positionieren
   getline(datei, zeile1); // erste  Zeile lesen
   getline(datei, zeile2); // zweite Zeile lesen
   cout << "1. Zeile: " << zeile1 << endl
        << "2. Zeile: " << zeile2 << endl;
}
```

Hätte z. B. die Datei `test.txt` vor dem Start des Programms 3.43 folgenden Inhalt:

```
Das ist das Haus vom Nikolaus.
Im Dezember kommt er zu den Kindern.
```

so hätte sie nach Ablauf dieses Programm folgenden Inhalt:

```
Das war das Haus vom Nikolaus.
Im war einmal mmt er zu den Kindern.
```

und Programm 3.43 würde Folgendes ausgeben:

```
wort1: Das
wort2: Im
1. Zeile: Das war das Haus vom Nikolaus.
2. Zeile: Im war einmal mmt er zu den Kindern.
```

Verwendung von << und >> in einer Anweisung ist nicht erlaubt

In einer Anweisung dürfen nicht gleichzeitig die beiden Operatoren << und >> angegeben werden. So würde z. B. bei Angabe der folgenden Anweisung in Programm 3.43 der Compiler einen Fehler melden:

```
datei >> wort2 << " war";
```

In diesem Fall würde für den Ausdruck

```
datei >> wort2;
```

ein `istream`-Objekt geliefert, für das der Operator << nicht definiert ist.

```
istream-Objekt << " war";
```

Anwendungsbeispiel: Dateien mit festen Satzlängen

Eine typische Anwendung von gleichzeitigem Lesen und Scheiben in Dateien sind Datenbank-Applikationen, wo Datensätze eine feste Länge haben, oder aber die Position und Länge von bestimmten Informationen bekannt ist. Programm 3.44 ist ein Demonstrationsbeispiel hierzu. Es unterhält zusätzlich eine Indexdatei, in der die Positionsnummern zu den entsprechenden Datensätzen gehalten werden.

Programm 3.44 – `readwrite3.cpp`:
Lesen und Schreiben in Dateien mit festen Satzlängen

```
#include <iostream>
#include <fstream>
using namespace std;
class DatBank {
 public:
   DatBank(string inDatei, bool create=true, int len=80)
                                                : satzLen(len), maxOffset(-1) {
      if (create) {
         long   nr = 0;
         string zeile;
         //... Originaldatei öffnen und dazu Daten- und Index-Datei anlegen
         orgDatei.open(inDatei.c_str(), ios::in);
         idxDatei.open((inDatei + ".idx").c_str(), ios::trunc|ios::in|ios::out);
         datDatei.open((inDatei + ".dat").c_str(), ios::trunc|ios::in|ios::out);
         while (getline(orgDatei, zeile)) { // Übertragen Daten aus Originaldatei
            write(nr, nr, zeile);
            nr++;
         }
      } else {
         //... Vorhandene Daten- und Index-Datei öffnen
         idxDatei.open((inDatei + ".idx").c_str(), ios::in|ios::out);
         datDatei.open((inDatei + ".dat").c_str(), ios::in|ios::out);
         idxDatei.seekg(0, ios::end);
         maxOffset = idxDatei.tellg()/sizeof(long) - 1;
      }
   }
```

```
   string read(long offset) {
      string zeile;
      long datsatzNr;
      idxDatei.seekg(offset * sizeof(long)); // Lesezeiger in idxDatei
       //... Datensatz-Nummer aus Index-Datei lesen
      if (idxDatei.read(reinterpret_cast<char *>(&datsatzNr), sizeof(long)) &&
          datDatei.seekg(datsatzNr*satzLen)  && // Lesezeiger in Daten-Datei
          getline(datDatei, zeile)) // Zeile aus Daten-Datei lesen
         return zeile;
      return fehlMeld("Lesefehler fuer Offset ", offset);
   }
   string write(long offset, long satznr, string zeile="") {
      idxDatei.seekp(offset*sizeof(long), ios::beg); // auf offset in Index-Datei
       //... Offset der neuen Zeile in Index-Datei schreiben
      if (!idxDatei.write(reinterpret_cast<char *>(&satznr), sizeof(long)))
         return fehlMeld("Schreibfehler in Index-Datei; Offset: ", offset);
       //... Evtl. neuen Datensatz schreiben
      if (zeile.length() > 0) {
         maxOffset++;
         datDatei.seekp(satznr*satzLen, ios::beg); // auf entspr. Datensatz
         if (!(datDatei << zeile << endl)) // Zeile in Daten-Datei schreiben
            return fehlMeld("Kein erfolgreiches Schreiben bei Zeile ", offset);
      }
      return zeile;
   }
   long getMaxOffset(void) { return maxOffset; }
   long getSatzNr(long offset) {
      long datsatzNr;
      idxDatei.seekg(offset * sizeof(long)); // Lesezeiger in idxDatei
      if (idxDatei.read(reinterpret_cast<char *>(&datsatzNr), sizeof(long)))
         return datsatzNr;
      fehlMeld("Lesefehler fuer Offset ", offset);
      return -1;
   }
 private:
   long    satzLen, maxOffset;
   fstream orgDatei, idxDatei, datDatei;
   string fehlMeld(char const *text, long wert=-1) {
      cerr << "..." << text;
      if (wert >= 0)
         cerr << wert;
      cerr << endl;
      return "";
   }
};
int main(int argc, char *argv[]) {
   if (argc != 2) {
      cerr << "usage: " << argv[0] << " dateiname" << endl;
      exit(1);
   }
```

```
   long i, j;
   DatBank d(argv[1]);

   //... Neuen Datensatz hinzufügen
   string neuDatSatz("Galster, Fritz, 47113 Gutriech");
   d.write(d.getMaxOffset()+1, 2000, neuDatSatz);
   //... Lesen der Daten über Index- und Daten-Datei
   for (i=0; i <= d.getMaxOffset(); i++)
      cout << d.read(i) << endl;
   //... Sortieren der Satznummern in der Index-Datei
   for (i=0; i < d.getMaxOffset(); i++)
      for (j=i+1; j <= d.getMaxOffset(); j++)
         if (d.read(i) > d.read(j)) {
            long hilf = d.getSatzNr(i); // nur Indizes in Index-Datei vertauschen
            d.write(i, d.getSatzNr(j));
            d.write(j, hilf);
         }
   cout << "---------------------------- Nun sortiert" << endl;
   for (i=0; i <= d.getMaxOffset(); i++)
      cout << d.read(i) << endl;
}
```

Hat man z. .B. eine Datei `adressen` mit folgenden Inhalt:

```
Meier, Hans, 91828 Musterstadt
Waller, Janette, 12345 Kleindorf
Adam, Fritz, 49092 Entenhausen
Zander, Emilia, 87373 Jansenhausen
Bender, Michael, 43562 Gressthal
```

und man ruft das Programm 3.44 wie folgt auf:
./readwrite3 adressen
so liefert es die folgende Ausgabe:

```
Meier, Hans, 91828 Musterstadt
Waller, Janette, 12345 Kleindorf
Adam, Fritz, 49092 Entenhausen
Zander, Emilia, 87373 Jansenhausen
Bender, Michael, 43562 Gressthal
Galster, Fritz, 47113 Gutriech
---------------------------- Nun sortiert
Adam, Fritz, 49092 Entenhausen
Bender, Michael, 43562 Gressthal
Galster, Fritz, 47113 Gutriech
Meier, Hans, 91828 Musterstadt
Waller, Janette, 12345 Kleindorf
Zander, Emilia, 87373 Jansenhause
```

Unter Verwendung von `rdbuf()` hätte die gleiche Aufgabenstellung auch wie in Programm 3.45, das das Gleiche wie Programm 3.44 leistet, gelöst werden können.

Programm 3.45 – `readwrite4.cpp`:
Lesen und Schreiben in Dateien mit festen Satzlängen mittels rdbuf()

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
//..................................................................... DatBank
class DatBank {
 public:
   DatBank(string inDatei, bool create=true, int len=80)
                                              : satzLen(len), maxOffset(-1) {
      if (create) {
         long   nr = 0;
         string zeile;
         //... Originaldatei öffnen und dazu Daten- und Index-Datei anlegen
         orgDatei.open(inDatei.c_str(), ios::in);
         readIdxDatei = new ifstream((inDatei + ".idx").c_str(),
                                     ios::trunc|ios::in|ios::out);
         readDatDatei = new ifstream((inDatei + ".dat").c_str(),
                                     ios::trunc|ios::in|ios::out);
         writeIdxDatei = new ostream(readIdxDatei->rdbuf());
         writeDatDatei = new ostream(readDatDatei->rdbuf());
         while (getline(orgDatei, zeile)) { // Übertragen Daten aus Originaldatei
            write(nr, nr, zeile);
            nr++;
         }
      } else {
         //... Vorhandene Daten- und Index-Datei öffnen
         orgDatei.open(inDatei.c_str(), ios::in);
         readIdxDatei = new ifstream((inDatei+".idx").c_str(), ios::in|ios::out);
         readDatDatei = new ifstream((inDatei+".dat").c_str(), ios::in|ios::out);
         writeIdxDatei = new ostream(readIdxDatei->rdbuf());
         writeDatDatei = new ostream(readDatDatei->rdbuf());
         readIdxDatei->seekg(0, ios::end);
         maxOffset = readIdxDatei->tellg()/sizeof(long) - 1;
      }
   }
   string read(long offset) {
      string zeile;
      long datsatzNr;
      readIdxDatei->seekg(offset * sizeof(long)); // Lesezeiger in idxDatei
       //... Datensatz-Nummer aus Index-Datei lesen
      if (readIdxDatei->read(reinterpret_cast<char *>(&datsatzNr), sizeof(long)) &&
          readDatDatei->seekg(datsatzNr*satzLen)  && // Lesezeiger in Daten-Datei
          getline(*readDatDatei, zeile)) // Zeile aus Daten-Datei lesen
         return zeile;
      return fehlMeld("Lesefehler fuer Offset ", offset);
   }
   string write(long offset, long satznr, string zeile="") {
      writeIdxDatei->seekp(offset*sizeof(long), ios::beg); // auf offset
```

```
       //... Offset der neuen Zeile in Index-Datei schreiben
      if (!writeIdxDatei->write(reinterpret_cast<char *>(&satznr), sizeof(long)))
         return fehlMeld("Schreibfehler in Index-Datei; Offset: ", offset);
       //... Evtl. neuen Datensatz schreiben
      if (zeile.length() > 0) {
         maxOffset++;
         writeDatDatei->seekp(satznr*satzLen, ios::beg); // auf entspr. Datensatz
         if (!(*writeDatDatei << zeile << endl)) // Zeile in Daten-Datei schreiben
            return fehlMeld("Kein erfolgreiches Schreiben bei Zeile ", offset);
      }
      return zeile;
   }
   long getMaxOffset(void) { return maxOffset; }
   long getSatzNr(long offset) { /* ...siehe Programm 3.44 */ }
 private:
   long     satzLen, maxOffset;
   fstream  orgDatei;
   ifstream *readIdxDatei,  *readDatDatei;
   ostream  *writeIdxDatei, *writeDatDatei;

   string fehlMeld(char const *text, long wert=-1) {
      /* ...siehe Programm 3.44 */
   }
};
//..................................................................... main
int main(int argc, char *argv[])
{
   /* ...siehe Programm 3.44 */
}
```

3.9.4 Eigener setw-Manipulator für den Operator >>

In manchen Situationen ist es hilfreich, nur eine bestimmte Anzahl von Zeichen aus der Eingabe zu lesen. Ein Beispiel dafür wäre die Interpretation einer HTML-Seite, bei der kodierte Sonderzeichen verwendet werden. Für solche Anwendungsfälle kann man sich einen eigenen `setw`-Manipulator für den Operator >> erstellen, wie es in Programm 3.46 gezeigt ist.

Programm 3.46 – `meinsetw.cpp`:
Eigener setw-Manipulator für den Operator >>

```
#include <string>
#include <iostream>
#include <sstream>
#include <fstream>
using namespace std;

class CIstream;
//-------------------------------------------------------------------------isetw
class isetw {
     friend class CIstream;
```

```
  public:
     isetw(unsigned w) : m_weite(w) { }
  private:
     unsigned m_weite;
};
//-------------------------------------------------------------------------CIstream
class CIstream: public istream {
 public:
    //..................................... Konstruktor für istream-Objekte
    CIstream(istream &s): istream(s.rdbuf()), m_streampuf(rdbuf()), m_weite(0) {}

    //.............................................. Konstruktor für Dateien
    CIstream(const char *name, ios::openmode mode)
         : istream(m_puffer = new filebuf()),
           m_streampuf((streambuf*)get()), m_weite(0) {
        m_puffer->open(name, mode);
    }
    //.......................................................... setzeWeite
    istream& setzeWeite(const isetw& weite) {
       m_weite = weite.m_weite; // neue Weite setzen
       if (!m_weite)          // bei Weite 0
          rdbuf(m_streampuf); // zum alten Puffer wechseln
       else {
          char *tmpPuffer = new char[m_weite + 1];
          rdbuf(m_streampuf);
          // m_weite Zeichen aus istream-Puffer mit 0 nach tmpPuffer übertragen
          tmpPuffer[read(tmpPuffer, m_weite).gcount()] = 0;
          m_andererPuffer.str(tmpPuffer); // tmpPuffer --> m_andererPuffer
          delete tmpPuffer;
          rdbuf(m_andererPuffer.rdbuf()); // m_andererPuffer nun istream-Puffer
       }
       return *this;
    }
 private:
    filebuf       *m_puffer;    // nur für Dateien benötigt
    streambuf     *m_streampuf; // Zeiger auf voreingest. istream-Puffer
    unsigned       m_weite;     // aktuell eingestellte Weite
    istringstream  m_andererPuffer;  // alternativer istream-Puffer
};
//-------------------------------------------------------------------------operator>>
namespace std { //........... muss im Namensraum std deklariert werden
   istream& operator>>(istream &stream, const isetw &weite) {
      return reinterpret_cast<CIstream *>(&stream)->setzeWeite(weite);
   }
}

int main(void) {
  CIstream eing(cin);
  string   str, ausStr;
  unsigned zeich;
```

```
   while (true) {
      switch (zeich = eing.get()) {
         case '&' :
            eing >> isetw(3) >> str >> isetw(0);
            if (str == "lt;")
               ausStr = "<";
            else if (str == "gt;")
               ausStr = ">";
            else {
               ausStr = str;
               eing >> isetw(1) >> str >> isetw(0);
               if ( (str = ausStr + str) == "amp;")
                  ausStr = "&";
               else {
                  ausStr = str;
                  eing >> isetw(1) >> str >> isetw(0);
                  str = ausStr + str;
                  if      (str == "auml;") ausStr = "ä";
                  else if (str == "ouml;") ausStr = "ö";
                  else if (str == "uuml;") ausStr = "ü";
                  else if (str == "Auml;") ausStr = "Ä";
                  else if (str == "Ouml;") ausStr = "Ö";
                  else if (str == "Uuml;") ausStr = "Ü";
                  else if (str == "quot;") ausStr = "";
                  else if (str == "nbsp;") ausStr = " ";
                    //.... usw.
               }
            }
            break;
         case EOF : return 0;
          default : ausStr = zeich;
      }
      cout << ausStr;
   }
}
```

Hat nun z. B. die Datei `sonderzeich.htm` den folgenden Inhalt:

```
<HTML>
<HEAD> <TITLE>Sonderzeichen</TITLE> </HEAD>
<BODY>
&Auml; = &amp;Auml; | &Ouml; = &amp;Ouml; | &Uuml; = &amp;Uuml;
&auml; = &amp;auml; | &ouml; = &amp;ouml; | &uuml; = &amp;uuml;
&lt; = &amp;lt;   | &gt; = &amp;gt;   | &quot; = &amp;quot;
  = &amp;nbsp; (Leerzeichen)
</BODY>
</HTML>
```

und man ruft Programm 3.46 wie folgt auf:
./meinsetw < sonderzeich.htm
so gibt es Folgendes aus:

```
<HTML>
<HEAD> <TITLE>Sonderzeichen</TITLE> </HEAD>
<BODY>
Ä = &Auml; | Ö = &Ouml; | Ü = &Uuml;
ä = &auml; | ö = &ouml; | ü = &uuml;
< = &lt;   | > = &gt;   | " = &quot;
  =   (Leerzeichen)
</BODY>
</HTML>
```

3.9.5 Verknüpfen von ostream-Objekten

`ostream`-Objekte können mittels der von der Klasse `ios` angebotenen Methode `tie()` mit `ios`-Objekte verbunden werden:

`ostream *ios::tie(ostream *neu)`
: kann benutzt werden, um ein `ios`-Objekt mit einem anderen `ostream`-Objekt `neu` zu verknüpfen. Jede Ein- und Ausgabeoperation auf das `ios`-Objekt, mit dem das `ostream`-Objekt nun verbunden ist, resultiert dann in einem Leeren des Ausgabepuffers vom `ostream`-Objekt mittels `flush()`. Die Voreinstellung ist, dass `cout` mit `cin` verbunden ist (`cin.tie(&cout)`), so dass bei jeder Operation auf `cin` automatisch der Ausgabepuffer von `cout` *„geflusht“* wird. Um diese Verbindung zu lösen, muss man `ios::tie(0)` aufrufen. Als Rückgabewert liefert diese Funktion einen Zeiger auf das ursprüngliche `ostream`-Objekt, wobei der Rückgabewert 0 anzeigt, dass momentan kein `ostream`-Objekt mit dem `ios`-Objekt verknüpft ist. Um z. B. das `ostream`-Objekt `cerr` mit `cout` zu verbinden, müsste Folgendes angegeben werden:

```
cerr.tie(&cout);
```

`ostream *ios::tie() const`
: liefert einen Zeiger auf das `ostream`-Objekt, mit dem das `ios`-Objekt aktuell verknüpft ist. Der Rückgabewert 0 zeigt an, dass momentan kein `ostream`-Objekt mit dem `ios`-Objekt verknüpft ist.

Programm 3.47 gibt zunächst über das voreingestellte Streamobjekt zu `cin` einen Text auf den Bildschirm aus, bevor es die Standardausgabe in eine Datei umlenkt. Dorthin schreibt es dann zunächst mittels dem nun für `cin` eingestellten Streamobjekt einen Text, bevor es anschließend die Ausgabe wieder auf den Bildschirm umlenkt, um dann dort wieder einen Text auszugeben.

Programm 3.47 – `tie.cpp`:
Demonstrationsprogramm zu tie()

```
#include <iostream>
#include <fstream>
using namespace std;
int main(void) {
  ostream  *stdStream;   // erhält Zeiger auf voreingest. Streamobj.
  ofstream  dateiStream; // Stream, der auf Datei eingestellt wird
   dateiStream.open("test.txt"); // dateiStream auf Datei test.txt setzen
```

```
    // Über tie() voreingest. ios-Obj. holen, um dort Text auszugeben
   *cin.tie() << "Dies wird auf stdout (cout) ausgegeben" << endl;
    // voreingest. Streamobj mit dateiStream verknüpfen
   stdStream = cin.tie(&dateiStream);
    // Über tie() akt. eingest. streambuf-Obj. holen, um dort Text auszugeben
   *cin.tie() << "Dies wird in Datei test.txt geschrieben" << endl;
    // Streamobj. wieder auf voreingestelltes setzen
   cin.tie(stdStream);
   *cin.tie() << "Dies wird wieder auf stdout (cout) ausgegeben" << endl;
   dateiStream.close(); // Schliessen von dateiStream
}
```

3.9.6 Ein-/Ausgabeumlenkung mit Streams

Mittels `ios::rdbuf()` können Stream-Objekte `streambuf`-Objekte gemeinsam benutzen, so dass das Schreiben bzw. Lesen in dem einen Stream im anderen Stream stattfindet. Unter Verwendung `ios::rdbuf()` ist es somit möglich, Ein-/Ausgabeumlenkung zu realisieren wie sie Unix/Linux üblich ist, wie z. B.:

```
progr 2> /tmp/logdatei // Alle Ausgaben auf stderr nach /tmp/logdatei schreiben
progr < datei // Anstelle vom Bildschirm soll dieses Prog. aus datei lesen
```

Programm 3.48 demonstriert, wie man die Standardfehlerausgabe in eine Datei umlenken kann. Die Datei, in die die `cerr`-Ausgaben umzulenken sind, muss dabei als erstes Argument auf der Kommandozeile angegeben werden.

Programm 3.48 – `cerruml.cpp`:
Umlenken der Standardfehlerausgabe in eine Datei

```
#include <iostream>
#include <fstream>
using namespace std;


int main(int argc, char *argv[]) {
  ofstream  logDatei;
  streambuf *cerrPuffer = 0;
   if (argc != 2) {
      cerr << "usage: " << argv[0] << " logDatei" << endl; exit(1);
   }
   logDatei.open(argv[1]);  // Öffnen der Log-Datei
   cerrPuffer = // Sichern des ursprünglichen Stream-Puffers von cerr
       cerr.rdbuf(logDatei.rdbuf()); // Umlenken von cerr in logDatei
   cerr << "Zeile 1 nach cerr (>logDatei) schreiben" << endl;
   cerr << "Zeile 2 nach cerr (>logDatei) schreiben" << endl;
   cerr.rdbuf(cerrPuffer);  // Umlenken von cerr wieder auf Bildschirm
   cerr << "Zeile 3 wird auf Bildschirm ausgegeben" << endl;
}
```

Ein mögliches Ablaufbeispiel des Programms 3.48 ist:

```
./cerruml test.txt ⏎
Zeile 3 wird auf Bildschirm ausgegeben
```

Die Datei `test.txt` hat dann folgenden Inhalt:

```
Zeile 1 nach cerr (>logDatei) schreiben
Zeile 2 nach cerr (>logDatei) schreiben
```

Programm 3.49 demonstriert, wie man die Standardeingabe in eine Datei umlenken kann. Die Datei, aus der die `cin`-Eingaben zu lesen sind, muss dabei als erstes Argument auf der Kommandozeile angegeben werden.

Programm 3.49 – `cinuml.cpp`:
Umlenken der Standardeingabe in eine Datei

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main(int argc, char *argv[]) {
  string    zeile;
  ifstream  einDatei;
  streambuf *cinPuffer = 0;
   if (argc != 2) {
      cerr << "usage: " << argv[0] << " einDatei" << endl; exit(1);
   }
   einDatei.open(argv[1]);  // Öffnen der Eingabe-Datei
   cinPuffer = // Sichern des ursprünglichen Stream-Puffers von cin
       cin.rdbuf(einDatei.rdbuf()); // Umlenken von cin in einDatei
   cout << "------------------Inhalt von Datei " << argv[1] << endl;
   while (getline(cin, zeile))
      cout << zeile << endl;
   cout << "-----------------------------------------------" << endl;
   cin.rdbuf(cinPuffer);  // Umlenken von cin wieder auf Tastatur
   cout << "Gib auch noch eine Zeile ein:" << endl;
   getline(cin, zeile);
   cout << "...." << zeile << "...." << endl;
}
```

Im nachfolgenden Ablaufbeispiel des Programms 3.49 wird der Inhalt der Datei `test.txt` mittels Eingabeumlenkung ausgegeben, bevor die Standardeingabe wieder auf die Tastatur umgelenkt wird, von der dann noch eine Zeile eingelesen wird.

```
./cinuml test.txt ⏎
------------------Inhalt von Datei test.txt
Zeile 1 nach cerr (>logDatei) schreiben
Zeile 2 nach cerr (>logDatei) schreiben
-----------------------------------------------
Gib auch noch eine Zeile ein:
Na gut, gebe ich halt noch eine ein ⏎
....Na gut, gebe ich halt noch eine ein....
```

3.9.7 Ableiten eigener Klassen von streambuf

Leitet man eine eigene Klasse von `streambuf` ab, muss man zumindest die folgenden virtuellen `protected`-Methoden überschreiben:

`int underflow()` - *leerer Eingabepuffer*
wenn man von Geräten lesen möchte, und/oder

`int overflow(int c=EOF)` - *voller Ausgabebereich*
wenn man auf Geräte schreiben möchte.

Weitere Kandidaten sind je nach Bedarf alle anderen virtuellen `protected`-Methoden der Klasse `streambuf`:

- für Klassen mit Eingabeoperationen:
  `streamsize xsgetn(char *s, streamsize n)` - *liest n Zeichen*
  `int uflow()` - *leerer Eingabepuffer*
  `int showmanyc()` - *liefert Anzahl noch verfügbarer Zeichen*
  `int pbackfail(int c=EOF)` - *schiebt ein Zeichen zurück*
- für Klassen mit Ausgabeoperationen:

  `streamsize xsputn(char *s, streamsize n)`
  für das Schreiben von `n` Zeichen

- für Klassen, die Positionieren ermöglichen und eigene Puffer verwenden:

  `streampos`
  `seekoff(streamoff offs, seekdir wie, openmode mod=in|out)`

  `streampos`
  `seekpos(streampos offset, openmode mode=in|out)`
  für relatives und absolutes Positionieren im Ein- bzw. Ausgabepuffer

  `streambuf *setbuf(char *puffer, streamsize n)`
  zum Festlegen eines neuen Puffers für `streambuf`-Objekt

  `int sync()`
  für Synchronisieren von Ein-/Ausgabepuffer

Einfaches Anwendungsbeispiel

Als Beispiel für das Ableiten einer eigenen Klasse wollen wir die Übung von Seite 43 hier mittels einer von `streambuf` abgeleiteten Klasse realisieren. Bei der „Ringelnatz“-Übung war es die Aufgabe, hinter jedem eingegebenen Vokal bzw. Vokalgruppe zusätzlich den String „`bi`“ bei der Ausgabe des entsprechenden Textes einzufügen. Programm 3.50 löst diese Ausgabenstellung in dem es in der von `streambuf` abgeleiteten eigenen Klasse die virtuelle `protected`-Methode `overflow()` überschreibt.

Programm 3.50 – `streambufneu.cpp`:
Ableiten einer eigenen Klasse von streambuf mit eigener Methode overflow()

```
#include <string>
#include <iostream>
#include <streambuf>
using namespace std;
```

```
class Ringelnatz : public streambuf {
   char   letztZeich;
   string vokal;
 public:
   Ringelnatz() : letztZeich('.'), vokal("aeiouAEIOU") { }
 protected:
   int overflow(int c) { // Überschreiben der protected-Methode
      if (c != EOF) {
         if (vokal.find(c) == string::npos &&
             vokal.find(letztZeich) != string::npos)
            cout << "bi";
         cout << char(c);
         letztZeich = c;
      }
      return c;
   }
};
int main(void) {
  string zeile;
  Ringelnatz rPuffer;
   ostream ausgabe(&rPuffer);
   while (getline(cin, zeile)) {
      for (unsigned i=0; i<zeile.length(); i++)
         ausgabe << zeile[i];
      cout << endl;
   }
}
```

Hat die Datei `original.ein` z. B. den folgenden Inhalt:

```
Ich habe dich,
Lotte, so lieb.
Hast auch du mich
Lieb? Nein, vergib.
Nah oder Fern
Gott sei dir gut.
Mein Herz hat gern
An dir geruht.
```

so liefert ein Aufruf wie: **./streambufneu < original.ein** folgende Ausgabe:

```
Ibich habibebi dibich,
Lobittebi, sobi liebib.
Habist aubich dubi mibich
Liebib? Neibin, vebirgibib.
Nabih obidebir Febirn
Gobitt seibi dibir gubit.
Meibin Hebirz habit gebirn
Abin dibir gebirubiht.
```

Nachfolgend wird gezeigt, wie man eigene Klassen von `streambuf` ableiten kann, um mit Filedeskriptoren direkt auf Geräte (wie Dateien, Pipes, Sockets usw.) zuzugreifen.

Eine Klasse für die Ausgabe mittels elementarer Filedeskriptoren

Programm 3.51 zeigt eine Klasse `outFdPuffer`, mit der man mittels Filedeskriptoren direkt auf Geräte schreiben kann. In `main()` werden hier mittels Objekte dieser Klasse alle Eingaben in der Standardeingabe wieder auf der Standardausgabe ausgegeben. Es werden dabei vier Möglichkeiten vorgestellt, wobei man die gewünschte Möglichkeit mit Angabe der entsprechenden Nummer beim Programmaufruf festlegen kann. Während bei der ersten Möglichkeit jede eingegebene Zeile sofort wieder ausgegeben wird, wird bei den anderen Möglichkeiten immer erst dann der Pufferinhalt auf die Standardausgabe geschrieben, wenn der Puffer voll ist.

Programm 3.51 – `fdoutstream.cpp`:
Klasse für die Ausgabe mittels elementarer Filedeskriptoren (unter Linux/Unix)

```
#include <iostream>
#include <streambuf>
using namespace std;
class outFdPuffer : public streambuf {
     unsigned m_groesse;
     int      m_fd;
     char    *m_puffer;
  public:
    outFdPuffer() : m_groesse(0), m_puffer(0) { }
    outFdPuffer(int fd, unsigned groesse = 1) { open(fd, groesse); }
    ~outFdPuffer() {
       if (m_puffer) {
          sync();          // Puffer flushen
          delete m_puffer; // und dann löschen
       }
    }
    void open(int fd, unsigned groesse = 1) {
       m_fd = fd;
       m_puffer = new char[m_groesse = (groesse == 0) ? 1 : groesse];
       setp(m_puffer, m_puffer + m_groesse); // Initial. interner Pufferzeiger
    }
    int sync() {
       if (pptr() > pbase()) {                    // Pufferinhalt
          write(m_fd, m_puffer, pptr() - pbase()); // physikal. schreiben
          setp(m_puffer, m_puffer + m_groesse);  // Rücksetzen interner Zeiger
       }
       return 0;
    }
    int overflow(int z) {
       sync(); // Puffer flushen
       if (z != EOF) {
          *pptr() = static_cast<char>(z); // Zeichen in Puffer schreiben
          pbump(1); // Schreibzeiger eine Position weiterbewegen
       }
       return z;
    }
};
```

```
int main(int argc, char *argv[]) {
  outFdPuffer  oPuffer(STDOUT_FILENO, 1024);
  ostream      os(&oPuffer);
   if (argc < 2 || atoi(argv[1]) <= 0 || atoi(argv[1]) > 4) {
      cerr << "usage: " << argv[0] << " [1|2|3|4]" << endl; exit(1);
   }
   switch (atoi(argv[1])) {
      case 1: //..... Zeilenweise ausgeben
         for (string s; getline(cin, s); )
            os << s << endl;
         break;
      case 2:  cin >> os.rdbuf(); break; // Puffer erst ausgeben, wenn voll
      case 3:  cin >> &oPuffer;   break; // Puffer erst ausgeben, wenn voll
      case 4:  os << cin.rdbuf(); break; // Puffer erst ausgeben, wenn voll
   }
}
```

Klasse zur Eingabe mittels elementarer Filedeskriptoren (Puffer der Größe 1)

Programm 3.52 zeigt eine Klasse `in1FdPuffer`, mit der man mittels Filedeskriptoren direkt von Geräten lesen kann. In `main()` werden hier mittels eines Objekts dieser Klasse alle Eingaben in der Standardeingabe wieder auf der Standardausgabe ausgegeben. Als internen Puffer verwendet die Klasse `in1FdPuffer` einen Puffer der Größe 1.

Programm 3.52 – `fdin1stream.cpp`:
Eine Klasse für die Eingabe mittels elementarer Filedeskriptoren (Puffer der Größe 1; unter Linux/Unix)

```
#include <iostream>
#include <streambuf>
using namespace std;
class in1FdPuffer : public streambuf {
  public:
    in1FdPuffer(int fd) : m_fd(fd) {
       setg(m_puffer, m_puffer+1, m_puffer+1); // Puffer initialisieren
    }
    int underflow(void) {
       if (gptr() < egptr()) // Wenn Puffer nicht leer
          return *gptr(); // --> nächstes Zeichen aus Puffer zurückgeben
       if (read(m_fd, m_puffer, 1) <= 0) // Ein Zeichen in Puffer lesen
          return EOF;
       setg(m_puffer, m_puffer, m_puffer+1); // interne Pufferzeiger initial.
       return *gptr(); // Zurückgeben des Zeichens im Puffer
    }
  protected: // Um abgeleiteten Klassen Zugriff auf Membervariablen zu erlauben
     int   m_fd;
     char  m_puffer[1]; // Puffergrösse == 1
};
int main(void) {
  in1FdPuffer inPuffer(STDIN_FILENO);
  istream     is(&inPuffer);
   cout << is.rdbuf();
}
```

Klasse zur Eingabe mittels elementarer Filedeskriptoren (Puffer der Größe n)

Programm 3.53 zeigt eine Klasse `innFdPuffer`, mit der man mittels Filedeskriptoren direkt von Geräten lesen kann. In `main()` werden hier mittels eines Objekts dieser Klasse alle Eingaben in der Standardeingabe wieder auf der Standardausgabe ausgegeben. Als internen Puffer verwendet diese Klasse `innFdPuffer` nun einen Puffer der Größe `n`, die der Benutzer selbst festlegen kann.

Programm 3.53 – `fdinnstream.cpp`:
Eine Klasse für die Eingabe mittels elementarer Filedeskriptoren (Puffer der Größe n; unter Linux/Unix)

```
#include <iostream>
#include <streambuf>
using namespace std;
class innFdPuffer : public streambuf {
  public:
    innFdPuffer() : m_groesse(0), m_puffer(0) { }
    innFdPuffer(int fd, unsigned groesse = 1) { open(fd, groesse); }
    ~innFdPuffer() {
       if (m_groesse) {
          close(m_fd);     // Filedeskriptor schliessen
          delete m_puffer; // und Puffer löschen
       }
    }
    void open(int fd, unsigned groesse = 1) {
       m_fd = fd;
       m_puffer = new char[m_groesse = (groesse == 0) ? 1 : groesse];
       setg(m_puffer, m_puffer + m_groesse, m_puffer + m_groesse);
    }
    int underflow(void) {
       int n;
       if (gptr() < egptr())
          return *gptr();
       if ( (n = read(m_fd, m_puffer, m_groesse)) <= 0)
          return EOF;
       setg(m_puffer, m_puffer, m_puffer + n);
       return *gptr();
    }
    streamsize xsgetn(char *zielPuffer, streamsize n) {
      int gelesen = 0, nochda;
       while (n) {
          if (!in_avail() && underflow() == EOF)
             break;
          if ( (nochda = in_avail()) > n)
             nochda = n;
          memcpy(zielPuffer + gelesen, gptr(), nochda);
          gbump(nochda);
          gelesen += nochda;
          n -= nochda;
       }
       return gelesen;
    }
```

```
  protected: // Um abgeleiteten Klassen Zugriff auf Membervariablen zu erlauben
    unsigned m_groesse;
    int      m_fd;
    char    *m_puffer;
};
int main(void) {
  innFdPuffer iPuffer(STDIN_FILENO, 20); // intern: Puffer mit 20 Zeichen
  char        zeile[80];                 // Dieser Puffer hat 80 Zeichen
   while (true) {
      unsigned n = iPuffer.sgetn(zeile, 80);
      if (n == 0)
         break;
      cout.write(zeile, n);
   }
}
```

Eine Klasse für das Positionieren mittels elementarer Filedeskriptoren

Programm 3.54 zeigt eine von `in1FdPuffer` abgeleitete Klasse `inFdPos`, mit der man mittels Filedeskriptoren direkt auf Geräten positionieren kann. Dieses Programm erwartet auf der Kommandozeile den Namen einer Datei, bevor es dann dem Benutzer über die Eingabe zweier Bytenummern einen Bereich festlegen lässt, der aus dieser Datei auszugeben ist.

Programm 3.54 – `fdposstream.cpp`:
Eine Klasse für das Positionieren mittels elementarer Filedeskriptoren (unter Linux/Unix)

```
#include <iostream>
#include <streambuf>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
using namespace std;

//-------------------------------------------------------------------- in1FdPuffer
class in1FdPuffer : public streambuf { /* siehe Programm 3.52 */ }

//...... Einführung von Abkürzungen
typedef streambuf::pos_type  pos_type;
typedef streambuf::off_type  off_type;
typedef ios::seekdir         seekdir;
typedef ios::openmode        openmode;

//---------------------------------------------------------------------- inFdPos
class inFdPos : public in1FdPuffer { // von in1FdPuffer abgeleitet
  public:
    inFdPos(int fd) : in1FdPuffer(fd) { }
    pos_type seekoff(off_type offset, seekdir dir, openmode) {
       pos_type pos = lseek(m_fd, offset,
                        (dir ==  std::ios::beg) ? SEEK_SET :
                        (dir ==  std::ios::cur) ? SEEK_CUR :
                                                  SEEK_END);
```

```
        if (pos < 0)
           return -1;
        setg(m_puffer, m_puffer + 1, m_puffer + 1);
        return pos;
     }
     pos_type seekpos(pos_type offset, openmode mode) {
        return seekoff(offset, std::ios::beg, mode);
     }
};

int main(int argc, char *argv[]) {
   int   fd;
   long  von, bis;
   char  z;

   if (argc != 2) {
      cerr << "usage: " << argv[0] << " dateiname" << endl;
      exit(1);
   }
   if ( (fd = open(argv[1], O_RDONLY)) < 0) {
      cerr << "kann Datei '" << argv[1] << "' nicht öffnen" << endl; exit(1);
   }
   inFdPos  posPuffer(fd);
   istream  is(&posPuffer);

   cerr << "Von Byte: "; cin >> von;
   cerr << "Bis Byte: "; cin >> bis;

   if (!is.seekg(bis) || !is.seekg(von)) {
      cerr << "Ungültige Bytenummer" << endl; exit(1);
   }
   while (is.get(z)) {
      cout << z;
      if (is.tellg() > bis)
         break;
   }
   cout << endl;
}
```

3.10 Übungen

3.10.1 Ausgeben einer Datei ab einer bestimmten Zeile

Erstellen Sie ein Programm `abzeile.cpp`, das den Inhalt einer Datei, deren Name als erstes Argument auf der Kommandzeile anzugeben ist, ab einer bestimmten Zeilennummer, die als zweites Argument auf der Kommandozeile anzugeben ist, ausgibt.

3.10.2 Erstellen eigener Manipulatoren

Geben Sie zum folgenden Programm `eigmanip.cpp` die fehlenden eigenen Manipulatoren an:

```
#include <string>
#include <iostream>
using namespace std;

//.... hier die fehlden Manipulatoren einfügen

//.............................................................. main
int main(void) {
   cout << cHeader;
   cout << "------------------------------------------------------" << endl;
   cout << pkte(5) << "Hallo" << pkte(20) << endl;
   cout << "Wie gehts, denn so" << pkte(15) << endl;
   cout << "------------------------------------------------------" << endl;
   cout << " 999  = " << romzahl(999)   << endl
        << " 6345 = " << romzahl(6345)  << endl
        << "15689 = " << romzahl(15689) << endl
        << " 1787 = " << romzahl(1787)  << endl;
}
```

Programm `eigmanip.cpp` sollte dann die folgende Ausgabe liefern:

```
#include <iostream>
using namespace std;

int main(void) {
}
------------------------------------------------------
.....Hallo...................
Wie gehts, denn so..............
------------------------------------------------------
 999  = IM
 6345 = MMMMMMCCCVL
15689 = MMMMMMMMMMMMMMMDCLXXXIX
 1787 = MDCCLXXXVII
```

Kapitel 4

Die Standard Template Library (STL)

4.1 Eine kurze Einführung in die STL

The Standard Template Library (STL) ist eine generische C++-Bibliothek, die grundlegende Datenstrukturen und Algorithmen zur Verfügung stellt. Nahezu jede Komponente der STL ist als Template realisiert.

4.1.1 Komponenten der STL

Die STL setzt sich aus folgenden Komponenten zusammen:

- **Vergleichsoperatoren und Klasse `pair` (zum Speichern von Paaren)**
- **Container**: Es gibt mehrere Arten von Containern:
  - *Sequenzielle*: `vector`, `list`, `deque`
  - *Assoziative*: `set`, `multiset`, `map`, `multimap`
  - *Adaptoren*: `stack`, `queue`, `priority_queue`
    Container-Adaptoren können mittels anderer Container realisiert werden. So ist es z. B. möglich, einen Stack intern mittels den Containerklasse `vector` oder aber auch mit der Containerklasse `list` zu realisieren.
- **Algorithmen**: sind effizient implementierte Standard-Algorithmen (wie `sort`, `find` oder `merge`), die auf die Container angewendet werden können.
- **Iteratoren**: sind abstrakte Zeiger, die es ermöglichen, alle Elemente eines Containers schrittweise zu durchlaufen.
- **Funktionsobjekte und Funktionsadaptoren**
  - Ein *Funktionsobjekt* ist eine Instanz einer Klasse, die den `operator()` überladen hat, so dass man ein Objekt dieser Klasse wie eine Funktion verwenden kann.
  - *Funktionsadaptoren* erlauben es, neue Funktionsobjekte aus bereits existierenden Funktionsobjekten zu erzeugen.
- **Allokatoren**: Jede STL-Containerklasse benutzt eine Allokator-Klasse zur Speicherverwaltung. Auf Allokatoren geht dieses Buch nicht ein.

4.1.2 Das grundlegende Konzept der STL

Die STL ist eine Bibliothek von Templateklassen, die größtenteils nebeneinander stehen und nicht voneinander abgeleitet sind. Durch ihre hohe Abstraktion ist es möglich, mit einer kleinen Anzahl von Klassen einen Großteil von *Standard-Datenstrukturen und -Algorithmen* abzudecken. Die STL unterscheidet dazu:

- **Container** („Behälter“)
  sind Templateklassen, die Objekte eines Datentyps aufnehmen und verwalten. Container sind also vordefinierte Datentypen (Klassen), die zur Lösung von häufig auftretenden Problemarten herangezogen werden können. Entscheidend ist, dass der Typ der Daten, die in solchen Containern (Templateklassen) gespeichert werden können, nicht festgelegt ist. So kann man in einem Container-Objekt z. B ebenso `int`-Zahlen wie auch eigene Objekte ablegen.
- **Algorithmen**
  sind Templatefunktionen, die eine gewisse Funktionalität implementieren, die auf einen Container bzw. auf dessen Iterator angewendet werden kann. Algorithmen sind also fertige Lösungen zu bestimmten Aufgabenstellungen (wie z. B. Sortieren von Daten). Die Algorithmen, die die STL dabei zur Verfügung stellt, sind wie Container auch nicht auf bestimmte Datentypen festgelegt.
- **Iteratoren**
  Jeder Container verfügt über Iteratoren, mit denen auf ein einzelnes Datenelement in ihm zugriffen werden kann. Iteratoren sind abstrakte Zeiger, die es ermöglichen, alle Elemente in einem Container schrittweise zu durchlaufen.

Hinsichtlich der Verwendung der STL-Klassen gilt:

- Die folgende Tabelle zeigt die Headerdateien, die inkludiert werden müssen, wenn man die entsprechende Funktionalität aus der STL nutzen möchte:

Headerdatei	**bei Verwendung von**	
`<utility>`	`pair`	

`<list>`
`<vector>`
`<deque>`
	`list`
	`vector`
	`deque`
		sequenzielle Container

`<stack>` `<queue>`	`stack` `queue`, `priority_queue`	*Container-Adaptoren*
`<set>` `<map>`	`set`, `multiset` `map`, `multimap`	*Assoziative Container*
`<algorithm>` `<numeric>`	Algorithmen speziellen mathematischen Operationen	*STL-Algorithmen*

- Alle Bezeichner der STL sind im Standard-Namensbereich `std` enthalten.
- Die STL ist so aufgebaut, dass die verschiedenen Container möglichst in der gleichen Weise benutzt werden können:
  - Der Zugriff auf die Elemente im Container ist über Iteratoren möglich.
  - Methoden für gleichartige Operationen besitzen den gleichen Namen. So liefert z. B. die Methode `size()` – unabhängig vom jeweiligen Containertyp – grundsätzlich die Anzahl der Elemente im Container zurück.

  Diese Gleichbehandlung ist zum einen für den STL-Nutzer von Vorteil und zum anderen ist sie ein entscheidendes Mittel, um verschiedene Algorithmen mit den verschiedenen Containern zusammenarbeiten zu lassen.

4.1.3 Beispiel zu Containern, Algorithmen und Iteratoren

Anhand eines ersten Beispiels sollen die wichtigen STL-Komponenten *Container*, *Algorithmen* und *Iteratoren* bereits vorab etwas erläutert werden.
Programm 4.1 ist ein einfaches Programm, das Folgendes kurz vorstellt:

- den STL-Container `vector` (vergleichbar mit Arrays in C)
- die STL-Algorithmen `reverse()` (Reihenfolge der Elemente in einem Bereich eines Arrays bzw. `vector`-Objekts umkehren) und `sort()` (zum Sortieren eines Bereichs in einem Array bzw. `vector`-Objekt).
- `begin()`-Iterator zeigt auf das erste Element des Vektors.
  `end()`-Iterator zeigt hinter das letzte Element des Vektors.
  Durch den Aufruf `reverse(v.begin(), v.end())` legt man dabei fest, dass alle Elemente des Vektors (von Anfang bis zum Ende) umzukehren sind.

Programm 4.1 – `stldemo.cpp`:
Einfaches Demoprogramm zu einem STL-Container und -Algorithmus

```
#include <vector>
#include <iostream>
using namespace std;
int main(void) {
   unsigned  i;
   //.................................................... vector (Container)
   vector<int> v(5); // deklariert einen Vector mit 5 Elementen
   for (i=0; i < 5; i++)
       v[i] = i;
   for (i=0; i < 5; i++)
      cout << v[i] << ", ";
   cout << endl;
   //.................................................... reverse (Algorithmus)
   reverse(v.begin(), v.end()); // Inhalt von Vector v umkehren
   for (i=0; i < v.size(); i++) // size() liefert Anzahl der Elemente in v
      cout << v[i] << ", ";
   cout << endl;
   //.................................................... sort (Algorithmus)
   double a[] =  6.6, 5.5, 3.3, 0.1, 2.2, 4.4 ;
   sort(a, a+4); // Ersten 4 Elemente im Array a sortieren
   for (i=0; i < 6; i++)
      cout << a[i] << ", ";
   cout << endl;
}
```

Programm 4.1 liefert die folgende Ausgabe:

```
0, 1, 2, 3, 4,
4, 3, 2, 1, 0,
0.1, 3.3, 5.5, 6.6, 2.2, 4.4,
```

In Programm 4.1 werden zwei Punkte ersichtlich:

- *STL-Container sind Templateklassen*
- *STL-Algorithmen sind globale Funktionen (keine Memberfunktionen)*

4.1.4 Das grundlegende Prinzip der Iteratoren

Wie bereits erwähnt, verfügt jeder Container über einen oder mehrere zugehörige *Iterator(en)*. Das grundlegende Prinzip des Iterators wird im Folgenden am Beispiel der STL-Liste erklärt. Alle anderen Container arbeiten nach dem gleichen Prinzip.
Die STL-Klasse `list` repräsentiert eine *doppelt verkettete Liste*.
Programm 4.2 legt eine Liste von Integern an und füllt diese mit einigen Werten, bevor es diese Liste mittels eines *Vorwärts-Iterators* und dann mittels eines *Rückwärts-Iterators* durchläuft und jeweils die einzelnen Elemente (hier ganze Zahlen) ausgibt.

Programm 4.2 – `stliter.cpp`:
Demoprogramm zum list-Container mit Vorwärts- und Rückwärts-Iterator

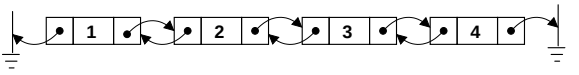
```
 1  #include <list>
 2  #include <iostream>
 3  using namespace std;
 5  int main(void) {
 6     list<int> myList;
 8     //..................................... Liste mit Zahlen füllen
 9     for (int i=1; i <= 4; i++)
10        myList.push_back(i);
12     //........................................... Vorwärts-Iterator
13     list<int>::iterator it;
14     // Durchlaufen der Liste
15     //    von Anfang:   it = myList.begin())
16     //    bis Ende:     it != myList.end();
17     //    schrittweise: ++it
18     for (it=myList.begin(); it != myList.end(); ++it)
19        cout << *it << " "; // *it = Zugriff auf Listen-Element
20     cout << endl;
22     //........................................ Rückwärts-Iterator
23     list<int>::reverse_iterator rit;
24     // Durchlaufen der Liste
25     //    vom Ende:     rit = myList.rbegin())
26     //    bis Anfang:   rit != myList.rend();
27     //    schrittweise: ++rit
28     for (rit=myList.rbegin(); rit != myList.rend(); ++rit)
29        cout << *rit << " "; // *rit = Zugriff auf Listen-Element
30     cout << endl;
31  }
```

Programm 4.2 liefert die folgende Ausgabe:

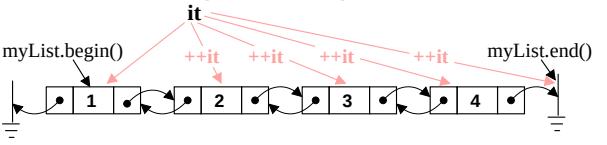
```
1 2 3 4
4 3 2 1
```

Abbildung 4.1 verdeutlicht den Ablauf im Programm 4.2.
Um die Liste `myList` in Programm 4.2 zu durchlaufen, benötigen wir einen Iterator. Dieser ermöglicht es uns, auf die einzelnen Elemente schrittweise zuzugreifen. Iteratoren sind Klassen, die innerhalb des jeweiligen Containers definiert sind, und stehen daher über den Scopeoperator `::` zur Verfügung.
In Kapitel 4.4 auf Seite 227 werden Iteratoren ausführlich vorgestellt.

Nach dem Füllen der Liste myList (Zeilen 8 – 10):



Liste vorwärts durchlaufen (Zeilen 12 – 19):



Liste rückwärts durchlaufen (Zeilen 22 – 29):

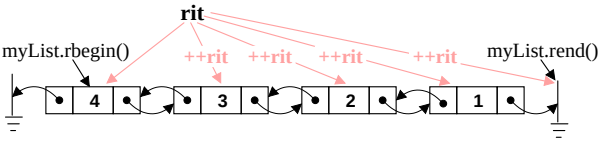


Abbildung 4.1: Iteratoren bei myList im Programm 4.2

Vorwärts-Iterator – Durchlaufen einer Liste von Anfang an

```
list<int>::iterator it; // legt Vorwärts-Iterator an
```

Dieser Iterator `it` entspricht einem Zeiger auf den Containertyp (in unserem Fall: `int`), hat aber auch Zusatzfunktionalität (Operator `++`), um durch die Liste zu iterieren.

```
it = myList.begin(); // Iterator auf den Anfang der Liste positionieren
```

Danach zeigt der Iterator `it` auf das Container-Element mit dem Wert 1. Da der Iterator auch ein Zeiger, kann der Wert dieses Elements wie folgt ausgegeben werden:

```
cout << *it;
```

Das Ende einer Liste wird durch einen Ende-Iterator gekennzeichnet, der mit der Methode `end()` abgefragt werden kann. Zeigt ein Iterator auf dieses Element, bedeutet dies, dass er auf das `NULL`-Element zeigt, das nicht mehr ausgegeben werden kann. Die ganze Liste lässt sich somit wie folgt vorwärts ausgeben:

```
for (it=myList.begin(); it != myList.end(); ++it)
   cout << *it << " "; // *it = Zugriff auf Listen-Element
```

Rückwärts-Iterator – Rückwärtiges Durchlaufen einer Liste vom Ende an

```
list<int>::reverse_iterator rit; // legt Rückwärts-Iterator (Reverse-Iterator) an
```

- Dieser Rückwärts-Iterator `rit` arbeitet nach denselben Prinzipien wie der Vorwärts-Iterator, nur dass Beginn und Ende vertauscht sind.
- Die entsprechenden Methoden lauten `rbegin()` und `rend()`.

Die ganze Liste lässt sich somit wie folgt rückwärts ausgeben:

```
for (rit=myList.rbegin(); rit != myList.rend(); ++rit)
   cout << *rit << " "; // *rit = Zugriff auf Listen-Element
```

4.2 Vergleichsoperatoren und Klasse pair

In diesem Kapitel werden mit den allgemein verwendbaren Vergleichsoperatoren und der Klasse `pair` grundlegende Komponenten der Standardbibliothek vorgestellt, die von anderen Bibliothekskomponenten benutzt werden.

4.2.1 Allgemeine Vergleichsoperatoren

Um redundante Definitionen von Vergleichsoperatoren zu vermeiden, stellt die Headerdatei `<utility>` vier Funktionstemplates zur Verfügung, die nur unter Verwendung der beiden Operatoren `==` und `<` den jeweiligen Vergleich durchführen. Die Definitionen dieser vier Funktionstemplates könnten wie folgt aussehen:

```
//.............................................. != (verwendet ==)
template <typename T>
inline bool operator!= (const T& x, const T& y) { return !(x == y); }
//........................................... >, <=, >= (verwenden <)
template <typename T>
inline bool operator> (const T& x, const T& y)  { return y < x;    }
template <typename T>
inline bool operator<= (const T& x, const T& y) { return !(y < x); }
template <typename T>
inline bool operator>= (const T& x, const T& y) { return !(x < y); }
```

Möchte man nun für eigene Klassen entsprechende Vergleichsoperatoren anbieten, reicht es aus, nur die beiden Operatoren `==` und `<` selbst definieren, um dann über diese vordefinierten alle Vergleichsoperatoren für die eigene Klasse anzubieten. Es ist dabei Folgendes zu beachten:

- Diese vier vordefinierten Funktionstemplates sind im Namensraum `std::rel_ops` definiert. Möchte man sie also automatisch zur Verfügung stellen lassen, muss man die folgende Anweisung angeben:

```
using namespace std::rel_ops;
```

Programm 4.3 definiert eine Klasse `QSum` und die beiden überladenen Operatoren `==` und `<`, wobei diese beiden Operatoren so überladen sind, dass sie nicht die Zahlen selbst, sondern deren Quersumme vergleichen. Die anderen Operatoren sind automatisch über die Headerdatei `<utility>` verfügbar.

Programm 4.3 – `stlvergl.cpp`:
Überladen der Operatoren == und < zum Vergleich von Quersummen

```
#include <utility>
#include <vector>
#include <iomanip>
#include <iostream>
using namespace std;

class QSum {
  friend bool operator==(const QSum& x, const QSum& y) { return x.qsum == y.qsum;}
  friend bool operator< (const QSum& x, const QSum& y) { return x.qsum < y.qsum; }
```

```
public:
  QSum& operator= (int z) {
     qsum = 0;
     zahl = z;
     while (z > 0) {
        qsum += z%10;
        z /= 10;
     }
     return *this;
  }
  int getZahl(void) { return zahl; }
  int getQsum(void) { return qsum; }
private:
  int zahl, qsum;
};
int main(void) {
  using namespace std::rel_ops;
  unsigned i, j;
  vector<QSum> q(5);
   q[0] = 12345;  q[1] = 99;  q[2] = 1881;  q[3] = 54321;  q[4] = 199;
   cout << setw(10) << " ";
   for (i=0; i < q.size(); i++)
      cout << q[i].getQsum() << "(" << setw(5) << q[i].getZahl() << ")  ";
   cout << endl;
   for (i=0; i < q.size(); i++) {
      cout << q[i].getQsum() << "(" << setw(5) << q[i].getZahl() << ")";
      cout << setw(i*11) << "";
      for (j=i; j < q.size(); j++) {
         string s = "";
         if (q[i] == q[j]) s+= "== ";
         if (q[i] != q[j]) s+= "!= ";
         if (q[i] <  q[j]) s+= "< ";
         if (q[i] <= q[j]) s+= "<= ";
         if (q[i] >  q[j]) s+= "> ";
         if (q[i] >= q[j]) s+= ">= ";
         cout << setw(11) << s;
      }
      cout << endl;
   }
}
```

Programm 4.3 liefert die folgende Tabellenausgabe:

```
          15(12345)  18(   99)  18( 1881)  15(54321)  19(  199)
15(12345)  == <= >=    != < <=    != < <=   == <= >=    != < <=
18(   99)             == <= >=   == <= >=    != > >=    != < <=
18( 1881)                        == <= >=    != > >=    != < <=
15(54321)                                   == <= >=    != < <=
19(  199)                                              == <= >=
```

4.2.2 Die Klasse pair

Um zwei Elemente als ein Paar zu verwalten (wie z. B. x- und y-Koordinaten einer Kurve), bietet die STL die Klasse `pair` an. Verwendet man diese Klasse, muss man folgende Headerdatei inkludieren:

```
#include <utility>
```

Der folgende Code legt z. B. zwei `pair`-Objekte an, die jeweils einen Namen einer Person mit deren Alter enthalten:

```
pair<string, int> person1("Hans", 37), person1("Antonia", 27);
```

Folgende Methoden bietet die Templateklasse `pair<T1,T2>` an:

`pair()`
: Default-Konstruktor; nur möglich, wenn sowohl `T1` als auch `T2` ihrerseits Default-Konstruktoren anbieten.

`pair(const T1& first, const T2& second)`
: Dieser Konstruktor erzeugt ein `pair`-Objekt aus `first` und `second`.

`pair(const pair& p)`
`pair& operator=(const pair& p)`
: Kopierkonstruktor und Zuweisungsoperator

`first, second`
: liefern die erste bzw. zweite Komponente des Paares.

`bool operator==(const pair& p1, const pair& p2)`
: nur möglich, wenn `T1` und `T2` auf Gleichheit geprüft werden können. Der Operator `==` liefert `true`, wenn die ersten beiden Elemente und die zweiten beiden Elemente von `p1` und `p2` gleich sind, und ansonsten `false`. Die Operatoren `==` und `!=` könnten z. B. wie folgt realisiert sein:

```
template <typename T1, typename T2>
bool operator==(const pair<T1, T2>& p1, const pair<T1, T2>& p2) {
  return p1.first == p2.first && p1.second == p2.second;
}
bool operator!=(...) { return !(p1 == p2); }
```

`bool operator<(const pair& p1, const pair& p2)`
: nur möglich, wenn `T1` und `T2` miteinander verglichen werden können. Der Operator `<` und abgeleitete Operatoren könnten z. B. wie folgt realisiert sein:

```
template <typename T1, typename T2>
bool operator<(const pair<T1, T2>& p1, const pair<T1, T2>& p2) {
  return p1.first < p2.first ||
         (!(p2.first < p1.first) && p1.second < p2.second);
}
bool operator<=(...) { return !(p2 < p1); }
bool operator> (...) { return   p2 < p1;  }
bool operator>=(...) { return !(p1 < p2); }
```

`make_pair(const T1& p1, const T2& p2)`
: globale Funktion; äquivalent zu `pair<T1, T2>(p1, p2)`

Programm 4.4 – `stlpair.cpp`:
Einfaches Demoprogramm zur pair-Klasse

```
#include <utility>
#include <iostream>
using namespace std;
template<typename F, typename S>
ostream& operator<< (ostream& os, const pair<F, S>& paar) {
   return os << "(" << paar.first << "," << paar.second << ")" << endl;
}
int main(void) {
   int  i, j;
   typedef pair<string, int> myPair;
   pair<string, int> person[6] = { myPair(),  // Default-Konstruktor
                                   myPair("Hans", 20),
                                   myPair("Emil", 30),
                                   myPair("Anton", 25),
                                   make_pair("Anton", 12) }; // make_pair-Aufruf
   person[5].first  = "Zorro";
   person[5].second = 10;
   for (i=0; i<5; i++) //............................... Sortieren nach Namen
      for (j=i+1; j<6; j++)
          if (person[i] > person[j])
              swap(person[i], person[j]); // allgemeine Routine zum Vertauschen
   cout << ".... Nach Namen sortiert:" << endl;
   for (i=0; i<6; i++)
      cout << person[i];
   for (i=0; i<5; i++) //............................... Sortieren nach Alter
      for (j=i+1; j<6; j++)
          if (person[i].second > person[j].second)
              swap(person[i], person[j]); // allgemeine Routine zum Vertauschen
   cout << ".... Nach Alter sortiert:" << endl;
   for (i=0; i<6; i++)
      cout << person[i];
}
```

Programm 4.4 liefert die folgende Ausgabe:

```
.... Nach Namen sortiert:
(,0)
(Anton,12)
(Anton,25)
(Emil,30)
(Hans,20)
(Zorro,10)
.... Nach Alter sortiert:
(,0)
(Zorro,10)
(Anton,12)
(Hans,20)
(Anton,25)
(Emil,30)
```

4.3 Container

In diesem Kapitel werden zentrale Operationen vorgestellt, über die Container verfügen, wobei nicht alle Container unbedingt alle diese Operationen anbieten.

4.3.1 Wichtige Methoden von Containern

Methoden zum Zugriff auf Anfang und Ende von Containern

Methode	gibt zurück …
`back()`	Wert des letzten Elements
`front()`	Wert des ersten Elements
`begin()`	Iterator auf das erste Element
`end()`	Iterator auf Ende-Element nach dem letztem (= `NULL`-Element)
`rbegin()`	Reverse-Iterator auf erstes Element von hinten (= letzte Element)
`rend()`	Reverse-Iterator auf Element vor dem ersten Element (= Ein Element vor dem `begin()`, ist auch `NULL`-Element)

Methoden zum Ermitteln der (maximale) Elementanzahl in Containern

Methode	gibt zurück …
`size()`	die Anzahl der Elemente im Container
`max_size()`	die maximale mögliche Größe für den aktuellen Container
`empty()`	`true`, falls der Container leer ist, sonst `false`

Methoden zum Einfügen von Elementen in Containern

`insert()`	Diese Methode hat je nach Containertyp unterschiedliche Parameter.

Container, die ihre Elemente unsortiert halten, bieten noch folgende Methoden an:

`push_back()`,	`push_front()`	Anhängen am Ende bzw. am Anfang
`pop_back()`,	`pop_front()`	Entfernen am Ende bzw. am Anfang

Methoden zum Löschen von Elementen in Containern

`erase()`	löscht ein Element bzw. Element-Bereich eines Containers
`clear()`	löscht alle Elemente des Containers

Methoden zum Suchen von Elementen in sortierten Containern

Sortierte Container bieten eine Methode `find()` zum Suchen eines Wertes innerhalb des Containers. Das Ergebnis der Suche ist ein Iterator. Wird kein entsprechendes Element gefunden, wird `end()` geliefert.

4.3.2 Wichtige Operatoren von Containern

Überladener Zuweisungsoperator =

Somit ist es möglich, dass man einem Container-Objekt ein anderes Container-Objekt vom gleichen Typ als Ganzes zuweisen kann.

Überladene Gleichheitsoperatoren == und !=

Zwei Container-Objekte des gleichen Typs sind gleich (==), wenn sie die gleiche Anzahl von Elementen besitzen und alle diese Elemente gleich sind, sonst sind sie ungleich (!=).

Überladene Vergleichsoperatoren <, <=, > und >=

Der entsprechende Vergleichsoperator liefert hier nur `true`, wenn dieser für alle Elemente der beiden Container-Objekte zutrifft. So liefert der Operator < nur dann `true`, wenn jedes Element des linken Container-Objekts kleiner als das korrespondierende Element im rechten Container-Objekt ist. Sollte das rechte Container-Objekt mehr Elemente als das linke beinhalten, werden diese beim Vergleich ignoriert.

Notwendige Konstrukte für eigene Klassen in Containern

Möchte man eigene definierte Typen, was üblicherweise Objekte zu eigenen Klassen sind, in Containern verwalten, sollten diese Klassen zumindest Folgendes anbieten:

- Standardkonstruktor
- Gleichheitsoperator ==
- Kleiner-Operator <

4.3.3 Typnamen in Containern

Alle Container stellen bestimmte implementierungsabhängige Typnamen zur Verfügung, die von anderen Komponenten der Bibliothek benötigt werden. Typnamen, die hier bei der Vorstellung von Methoden verwendet werden, sind:

- `iterator` – Iteratortyp
  kann für Container zu einer beliebigen Iteratorkategorie gehören mit Ausnahme von *Output-Iterator* (siehe auch Kapitel 4.4.2 auf Seite 231).
- `const_iterator` – nur lesender Iteratortyp
  kann für Container zu einer beliebigen Iteratorkategorie gehören mit Ausnahme von *Output-Iterator* (siehe auch Kapitel 4.4.2 auf Seite 231).
- `difference_type` – vorzeichenbehafteter ganzzahliger Typ
  ist der Abstand zwischen zwei Iteratoren, wie z. B. das Ergebnis einer Subtraktioin von zwei Iteratoren.
- `size_type` – vorzeichenloser ganzzahliger Typ
  kann alle nicht negativen Werte von `difference_type` aufnehmen und z. B. als Index auf Containerelemente benutzt werden.

4.3.4 Die sequenzielle Containerklasse vector

Um Elemente in einem dynamisch erweiterbaren Array zu verwalten, bietet die STL die Klasse `vector` an. Verwendet man diese Klasse, muss man folgende Headerdatei inkludieren:

```
#include <vector>
```

Der Syntax zur Erzeugung eines Vektors ist z. B. Folgende:

```
vector<T> name(elemZahl)
```

Hiermit wird ein Vektor *name* der Größe *elemZahl* angelegt, der Elemente vom Typ *T* aufnehmen kann. Um z. B. einen Vektor `meinArray` der Größe 20 anzulegen, der `double`-Zahlen speichern kann, müsste man folgenden Code angeben:

```
vector<double> meinArray(20);
```

Für die Zeiten bei `vector`-Objekten gilt:

- Zugriffe auf Elemente oder Anhängen von Elementen: *O(1)*
- Auffinden bestimmter Elemente oder Einfügen von Elementen: *O(n)*

Konstruktoren der Klasse vector

Folgende Konstruktoren bietet die Templateklasse `vector<T>` an:

`vector()` $O(1)$

`vector(const vector&)` $O(n)$
: Default-Konstruktor (legt leeren Vektor an) und Kopierkonstruktor.

`vector(size_type n)` $O(n)$
: legt Vektor der Größe `n` an. Hier wird für jedes angelegte Element der Default-Konstruktor aufgerufen, so dass bereits entsprechend viele Elemente existieren, was beim vorherigen Konstruktor nicht zutrifft.

`vector(size_type n, const T& elem)` $O(n)$
: legt Vektor mit `n` `elem`-Kopien an.

`vector(InputIterator anfang, InputIterator ende)` $O(n)$
: legt einen Vektor an, der mit dem Bereich von einschließlich `anfang` bis ausschließlich `ende` (wird nicht mehr kopiert) initialisiert wird.

Programm 4.5 – `vectorkonstr.cpp`:
Einfaches Demoprogramm zu den vector-Konstruktoren

```
#include <string>
#include <vector>
#include <iomanip>
#include <iostream>
using namespace std;
template <typename T>
void vectorAusgabe(vector<T>v, string text) {
   cout << setw(12) << text;
   for (unsigned i=0; i < v.size(); i++)
      cout << v[i] << ", ";
   cout << endl;
}
class K {
  public:
     K() { pi = 3.14; }
     double getPi() const { return pi; }
  private:
     double pi;
};
```

```
int main(void) {
   unsigned i;
   //...............................................1 (Default-Konstruktor)
   vector<string> leerVektor; // legt leeren Vektor für string-Objekte an
   cout << "leerVektor (Grösse): " << leerVektor.size() << endl;

   //................................................2 (Vektor der Grö"se n
   vector<double> gltPkt(10);  // legt Vektor der Grösse 10 für double an
   vectorAusgabe<double>(gltPkt, "gltPkt: ");
   vector<K>  piVektor(5); // legt Vektor der Grösse 5 für K-Objekte an
   cout << setw(12) << "piVektor: ";
   for (i=0; i < piVektor.size(); i++)
      cout << piVektor[i].getPi() << ", ";
   cout << endl;
   //.................................3 (Vektor mit n Kopien eines Elements)
   int zahl = 4711;
   vector<int> ganzZahl(10, zahl); // legt Vektor mit 10 mal dem Wert 4711 an
   vectorAusgabe<int>(ganzZahl, "ganzZahl: ");
   vector<string> gruss(5, string("Hallo")); // initialis. gruss mit 5 "Hallo"
   vectorAusgabe<string>(gruss, "gruss: ");
   //................................................4 (Kopierkonstruktor)
   vector<string> gruss2(gruss); // legt Vector an, der Kopie von gruss ist
   vectorAusgabe<string>(gruss2, "gruss2: ");
   //.............................5 (Teil-Kopierkonstruktor mit Iteratoren)
   gruss[3] = "Mister";
   vector<string> gruss3(&gruss[2], &gruss[4]); // legt Vector mit 2 Strings
                                                // aus gruss[2] und gruss[3] an
   vectorAusgabe<string>(gruss3, "gruss3: ");
}
```

Programm 4.5 liefert die folgende Ausgabe:

```
leerVektor (Grösse): 0
    gltPkt: 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  piVektor: 3.14, 3.14, 3.14, 3.14, 3.14,
  ganzZahl: 4711, 4711, 4711, 4711, 4711, 4711, 4711, 4711, 4711, 4711,
     gruss: Hallo, Hallo, Hallo, Hallo, Hallo,
    gruss2: Hallo, Hallo, Hallo, Hallo, Hallo,
    gruss3: Hallo, Mister,
```

Zuweisen von vector-Objekten

Um `vector`-Objekte einander zuweisen zu können, wird der folgende Zuweisungsoperator angeboten:

`vector& operator= (const vector&)` $O(n)$

Programm 4.6 – `vectorzuw.cpp`:
Demoprogramm zum Zuweisen von vector-Objekten

```
#include <vector>
#include <iostream>
using namespace std;
```

```
template <typename T>
ostream& operator<<(ostream& os, const vector<T>& v) {
   for (unsigned i=0; i < v.size(); i++)
      os << v[i] << ", ";
   return os;
}
int main(void) {
   unsigned i;
   vector<int> vekt1(5), vekt2(10), vekt3;
   for (i=0; i < vekt1.size(); i++)
      vekt1[i] = i;
   cout << "vekt1: " << vekt1 << endl;
   for (i=0; i < vekt2.size(); i++)
      vekt2[i] = (i+1)*10;
   cout << "vekt2: " << vekt2 << endl;
   vekt3 = vekt2; cout << "vekt3: " << vekt3 << endl;
   vekt2 = vekt1; cout << "vekt2: " << vekt2 << endl;
   vekt1 = vekt3; cout << "vekt1: " << vekt1 << endl;
}
```

Programm 4.6 liefert die folgende Ausgabe:

```
vekt1: 0, 1, 2, 3, 4,
vekt2: 10, 20, 30, 40, 50, 60, 70, 80, 90, 100,
vekt3: 10, 20, 30, 40, 50, 60, 70, 80, 90, 100,
vekt2: 0, 1, 2, 3, 4,
vekt1: 10, 20, 30, 40, 50, 60, 70, 80, 90, 100,
```

Zugriff auf vector-Elemente

Um auf einzelne Elemente in einem `vector`-Objekt lesend bzw. schreibend zuzugreifen, stehen die folgenden Methoden zur Verfügung:

`T& operator[size_type i]` $O(1)$

`const T& operator[size_type i] const` $O(1)$

liefert Referenz auf *i*. tes Element. Hier wird keine Überprüfung durchgeführt, ob der angegebene Index innerhalb des erlaubten Bereichs liegt.

`T& at(size_type i)` $O(1)$

`const T& at(size_type i) const` $O(1)$

liefert Referenz auf *i*. tes Element. Sollte sich *i* nicht innerhalb des erlaubten Bereichs befinden, wird hier – anders als beim Indexoperator `[]` – eine `out_of_range`-Exception ausgelöst, die man abfangen kann.

`T& front()` $O(1)$

`const T& front() const` $O(1)$

liefert Referenz auf erstes Element des Vektors; entspricht `vekt.at(0)`. Ein Aufruf dieser Methode führt bei einem leeren Vektor zum Programmabsturz.

`T& back()` $O(1)$
`const T& back() const` $O(1)$
liefert Referenz auf letztes Element des Vektors. Ein Aufruf dieser Methode führt bei einem leeren Vektor zum Programmabsturz. Diese Methode entspricht dem Aufruf `vekt.at(vekt.size()-1)`.

Programm 4.7 – `vectorzugr.cpp`: Zugriffsmöglichkeiten auf vector-Elemente

```
#include <vector>
#include <iostream>
using namespace std;
void vectorAusgabe(vector<int>v, string text) {
   cout << text;
   for (unsigned i=0; i < v.size(); i++)
      cout << v[i] << ", ";
   cout << endl;
}
int main(void) {
  vector<int> ganzZahl(10);
   for (unsigned i=0; i<ganzZahl.size(); i++)
      ganzZahl[i] = (i+1)*(i+1);
   vectorAusgabe(ganzZahl, "ganzZahl: ");
   cout << "Erstes Element: "  << ganzZahl.front() << endl
        << "Letztes Element: " << ganzZahl.back()  << endl;
   ganzZahl.front() = -1000;
   ganzZahl.back() = -1234;
   vectorAusgabe(ganzZahl, "ganzZahl: ");
   // ganzZahl.at(1000) = 4711; // Hier würde Programm abgebrochen
   // ganzZahl[1000] = 4711; // Hier Speicherüberschreibung
   vector<string> leerVektor; // legt leeren Vektor für string-Objekte an
   // cout << leerVektor.front(); // führt bei leeren Vektor zu Programmabbruch
   // cout << leerVektor.back();  // führt bei leeren Vektor zu Programmabbruch
}
```

Programm 4.7 liefert die folgende Ausgabe:

```
ganzZahl: 1, 4, 9, 16, 25, 36, 49, 64, 81, 100,
Erstes Element: 1
Letztes Element: 100
ganzZahl: -1000, 4, 9, 16, 25, 36, 49, 64, 81, -1234,
```

Methoden zum Zuweisen und Einfügen von vector-Elementen

`void assign(size_type n, const T& elem)` $O(n)$
weist dem Vektor `n` Mal das Element `elem` zu. Die vorher im Vektor enthaltenen Elemente gehen dabei alle verloren.

`void assign(InputIterator anfang, InputIterator ende)` $O(n)$
weist dem Vektor alle Elemente zu, die sich in dem Bereich von einschließlich `anfang` bis ausschließlich `ende` (wird nicht mehr zugewiesen) befinden. Die vorher im Vektor enthaltenen Elemente gehen dabei alle verloren.

`iterator insert(iterator pos, const T& elem)` $O(n)$
fügt das Element `elem` vor der Position `pos` ein und liefert einen Iterator auf dieses eingefügte Element.

`void insert(iterator pos, size_type n, const T& elem)` $O(n)$
fügt das Element `elem` `n` Mal vor der Position `pos` ein.

`void insert(iterator pos,InputIterator anf,InputIterator end)` $O(n)$
fügt vor der Position `pos` alle Elemente ein, die sich in dem Bereich von einschließlich `anf` bis ausschließlich `end` (wird nicht mehr eingefügt) befinden.

`void push_back(const T& elem)` $O(1)$
fügt das Element `elem` am Ende des Vektors an.

Methoden zum Löschen von vector-Elementen

`void clear()` $O(n)$
löscht alle Elemente des Vektors. Der Vektor hat danach die Größe 0.

`iterator erase(iterator pos)` $O(n)$
löscht das Element an Iterator-Position `pos`. Als Rückgabewert wird die Position des Elements geliefert, das sich nach dem gelöschten Element befindet.

`iterator erase(iterator anfang, iterator ende)` $O(n)$
löscht alle Elemente, die sich in dem Bereich von einschließlich `anfang` bis ausschließlich `ende` (wird nicht mehr gelöscht) befinden. Als Rückgabewert wird die Position des Elements geliefert, das sich nach dem zuletzt gelöschten Element befindet.

`void pop_back()` $O(1)$
löscht das letzte Element des Vektors.

Programm 4.8 – `vectorinserase.cpp`:
Zuweisen, Einfügen und Löschen von vector-Elementen

```
#include <vector>
#include <iostream>
using namespace std;
void ausgabe(vector<char>v, string text="") {
   vector<char>::iterator  it;
   cout << text;
   for (it = v.begin(); it != v.end(); it++)
      cout << *it;
}
int main(void) {
   unsigned i;
   vector<char> v1(1000), v2(200);

   //.............................................................assign
   v1.assign(6, 'x');         ausgabe(v1, "v1: "); cout << endl;
   v2.assign(&v1[2], &v1[5]); ausgabe(v2, "v2: "); cout << endl;
   //.............................................................clear
   v2.clear(); cout << "Grösse von v2 nun: " << v2.size() << endl;
```

```
   //.................................................push_back / erase
   vector<char> v3;
   for (i=0; i < 5; i++)
      v3.push_back('a'+ i);
   cout << "erase von vorne: ";
   while (v3.begin() != v3.end()) {
      v3.erase(v3.begin());
      ausgabe(v3);  cout << ",  ";
   }
   cout << endl;
   //.....................................................pop_back
   for (i=0; i < 5; i++)
      v3.push_back('a'+ i);
   cout << "pop_back: ";
   while (v3.begin() != v3.end()) {
      v3.pop_back();
      ausgabe(v3);  cout << ",  ";
   }
   cout << endl;
   //.....................................................insert
   for (i=0; i < 10; i++)
      v3.push_back('a'+ i);
   ausgabe(v3);  cout << endl;
   v3.insert(v3.begin()+2, '-');     ausgabe(v3);  cout << endl;
   v3.insert(v3.begin()+5, 10, '_'); ausgabe(v3);  cout << endl;
}
```

Programm 4.8 liefert die folgende Ausgabe:

```
v1: xxxxxx
v2: xxx
Grösse von v2 nun: 0
erase von vorne: bcde,  cde,  de,  e,  ,
pop_back: abcd,  abc,  ab,  a,  ,
abcdefghij
ab-cdefghij
ab-cd__________efghij
```

Größe von vector-Objekten

Methoden zum Erfragen bzw. Verändern der Größe von `vector`-Objekten sind:

`size_type size() const` $O(1)$
: liefert aktuelle Anzahl von Elementen im Vektor.

`size_type capacity() const` $O(1)$
: liefert Anzahl von Elementen, die Vektor insgesamt aufnehmen kann, bevor er vergrößert werden muss. Die Anzahl der noch freien Plätze lässt sich mit `capacity() - size()` ermitteln.

`size_type max_size() const` $O(1)$
: liefert (systemabhängige) maximal mögliche Größe für den jeweiligen Vektor.

`bool empty() const` $O(1)$
liefert `true`, wenn Vektor keine Elemente enthält und ansonsten `false`.

`void resize(size_type n, T elem = T())` $O(n)$
vergrößert bzw. verkleinert den Vektor auf Größe `n` mit Initialisierung. Bei einer Vergrößerung wird für neu hinzugefügten Elemente entweder der Default-Konstruktor aufgerufen oder sie werden mit einer Kopie von `elem` initialisiert.

`void reserve(size_type n)` $O(n)$
vergrößert den Vektor auf Größe `n` ohne Initialisierung. Ist `n` $<=$ `capacity()`, dann hat dieser Aufruf keine Auswirkung. Ansonsten wird der Vektor entsprechend vergrößert, so dass danach gilt: `n` $<=$ `capacity()`. Dieser Aufruf beeinflusst nicht die Größe, die von `size()` geliefert wird. Wenn `n > max_size()` ist, wird die Exception `length_error` geschickt.

Programm 4.9 – `vectorgroes.cpp`:
Größen von vector-Objekten

```
#include <vector>
#include <iomanip>
#include <iostream>
using namespace std;
template <typename T>
void ausgabe(T v, string text="") {
  typename T::iterator it;
   cout << text;
   for (it = v.begin(); it != v.end(); it++)
      cout << setw(3) << *it << ", ";
   cout << "(" << v.size() << ", " << v.capacity() << ", " << v.max_size()
        << ")"  << endl;
}
int main(void) {
   vector<char>   char_(5);
   vector<double> double_(5);

   ausgabe(char_, "  char_: ");
   ausgabe(double_, "double_: ");
   char_.assign(2, 'x');   ausgabe(char_, "  char_: ");
   double_.assign(2, 1.3); ausgabe(double_, "double_: ");
   char_.resize(10, 'y');   ausgabe(char_, "  char_: ");
   double_.resize(10, 9.9); ausgabe(double_, "double_: ");
   char_.resize(5);   ausgabe(char_, "  char_: ");
   double_.resize(5); ausgabe(double_, "double_: ");
   vector<int> v;
   for (unsigned i = 0; i < 5; i++)
      v.push_back(i);
   while ( !v.empty() ) {
      cout << v.back() << ", ";
      v.pop_back();
   }
}
```

Programm 4.9 liefert z. B. die folgende Ausgabe:

```
  char_:   ,   ,   ,   ,   , (5, 5, 4294967295)
double_:   0,   0,   0,   0,   0, (5, 5, 536870911)
  char_:   x,   x, (2, 2, 4294967295)
double_: 1.3, 1.3, (2, 2, 536870911)
  char_:   x,   x,   y,   y,   y,   y,   y,   y,   y,   y, (10, 10, 4294967295)
double_: 1.3, 1.3, 9.9, 9.9, 9.9, 9.9, 9.9, 9.9, 9.9, 9.9, (10, 10, 536870911)
  char_:   x,   x,   y,   y,   y, (5, 5, 4294967295)
double_: 1.3, 1.3, 9.9, 9.9, 9.9, (5, 5, 536870911)
4, 3, 2, 1, 0,
```

Vergleichen und Vertauschen von vector-Objekten

Die Klasse `vector` bietet folgende überladenen globalen Vergleichsoperatoren an:

`operator==(), operator!=()` $O(n)$

Zwei Vektoren sind gleich, wenn sie beide den gleichen Elementtyp besitzen, die gleiche Größe haben und die Werte jedes Elementpaares an der gleichen Stelle (Index) den gleichen Wert besitzen. Andernfalls sind die beiden Vektoren ungleich. Eine mögliche Realisierung dieser beiden Operatoren wäre z. B.:

```
template<typename It1, typename It2>
bool gleich(It1 a1, It1 end1, It2 a2) {
   for ( ; a1 != end1; ++a1, ++a2)
      if (!(*a1 == *a2))
         return false;
   return true;
}
template<typename T>
bool operator== (const vector<T>& v1, const vector<T>& v2) {
   return v1.size() == v2.size() && gleich(v1.begin(), v1.end(), v2.begin());
}
template<typename T>
bool operator!= (const vector<T>& v1, const vector<T>& v2) {
   return !(v1 == v2);
}
```

`operator<(), operator<=(), operator>(), operator>=()` $O(n)$

Der Vergleich von `v1 < v2` liefert `true`, wenn beim paarweisen Vergleich von `v1` und `v2` im ersten Paar, bei dem sich die beiden Werte unterscheiden, der Wert von `v1` kleiner als der von `v2` ist, andernfalls liefert dieser Vergleich `false`. Eine mögliche Realisierung dieser Operatoren wäre z. B.:

```
template<typename It1, typename It2>
bool vectorKleiner(It1 a1, It1 end1, It2 a2, It2 end2) {
   for ( ; a1 != end1 && a2 != end2 ; ++a1, ++a2) {
      if (*a1 < *a2) return true;
      if (*a2 < *a1) return false;
   }
   return a1 == end1 && a2 != end2;
}
```

```
template<typename T>
bool operator< (const vector<T>& v1, const vector<T>& v2) {
  return vectorKleiner(v1.begin(), v1.end(), v2.begin(), v2.end());
}
template<typename T>
bool operator<=(const vector<T>& v1, const vector<T>& v2) { return !(v2<v1); }
template<typename T>
bool operator> (const vector<T>& v1, const vector<T>& v2) { return v2 < v1;  }
template<typename T>
bool operator>=(const vector<T>& v1, const vector<T>& v2) { return !(v1<v2); }
```

Zum Vertauschen der Inhalte von zwei `vector`-Objekten steht die folgende Methode zur Verfügung:

`void swap(vector& v)` $O(n)$
: vertauscht die Elemente des eigenen Vektors mit denen von Vektor `v`.

Programm 4.10 – `vectorvergl.cpp`:
Demoprogramm zum Vergleichen und Vertauschen von vector-Objekten

```
#include <cstdlib>
#include <ctime>
#include <vector>
#include <iostream>
using namespace std;
//---------------------------------------------------------------------ausgabe
template <typename T>
void ausgabe(T v, size_t i) {
   typename T::iterator it;
   cout << "v[" << i << "]: ";
   for (it = v.begin(); it != v.end(); it++)
      cout << *it << ", ";
   cout << endl;
}

int main(void) {
  unsigned i, j;
  vector<int> v[5];
   srand(time(0));
   for (i=0; i< 5; i++)
      for (j=0; j < 7; j++)
         v[i].push_back(rand()%9+1);
   for (i=0; i< 5; i++)
      ausgabe(v[i], i);
   for (i=0; i< 4; i++) //... einfacher Bubble-Sort
      for (j=i+1; j < 5; j++)
         if (v[i] > v [j])
            v[i].swap(v[j]);
   cout << "...... nach Sortieren" << endl;
   for (i=0; i< 5; i++)
      ausgabe(v[i], i);
}
```

Programm 4.10 liefert z. B. die folgende Ausgabe:

```
v[0]: 7, 9, 2, 5, 3, 4, 6,
v[1]: 1, 3, 3, 3, 4, 8, 4,
v[2]: 7, 2, 7, 7, 7, 6, 4,
v[3]: 7, 6, 4, 4, 9, 9, 8,
v[4]: 3, 3, 4, 8, 2, 4, 1,
...... nach Sortieren
v[0]: 1, 3, 3, 3, 4, 8, 4,
v[1]: 3, 3, 4, 8, 2, 4, 1,
v[2]: 7, 2, 7, 7, 7, 6, 4,
v[3]: 7, 6, 4, 4, 9, 9, 8,
v[4]: 7, 9, 2, 5, 3, 4, 6,
```

Iterator-Methoden der Klasse vector

`iterator begin()` $O(1)$
`const_iterator begin() const` $O(1)$
liefern beide einen Iterator auf das erste Element des Vektors.

`iterator end()` $O(1)$
`const_iterator end() const` $O(1)$
liefern beide einen Iterator auf die Position hinter das letzte Element.

`reverse_iterator rbegin()` $O(1)$
`const_reverse_iterator rbegin() const` $O(1)$
liefern beide einen *reverse*-Iterator auf das letzte Element des Vektors.

`reverse_iterator rend()` $O(1)$
`const_reverse_iterator rend() const` $O(1)$
liefern beide einen *reverse*-Iterator auf die Position vor das erste Element.

Programm 4.11 – `vectoriter.cpp`:
Demoprogramm zu den Iterator-Methoden

```
#include <vector>
#include <iostream>
using namespace std;
//---------------------------------------------------------------------operator<<
template<typename T>
ostream& operator<< (ostream& os, vector<T> v) {
    typename vector<T>::iterator vorBegin;
    typename vector<T>::reverse_iterator vorRend;
    typename vector<T>::iterator rueckEnd;
    typename vector<T>::reverse_iterator rueckRbegin;

    os << "vorwärts (begin): {";
    for (vorBegin = v.begin(); vorBegin != v.end(); vorBegin++)
       os << *vorBegin << ",";
    os << "} " << endl << "vorwärts (rend):  {";
    for (vorRend = v.rend()-1; vorRend >= v.rbegin(); vorRend--)
       os << *vorRend << ",";
    os << "} " << endl << "rückwärts (end):    {";
```

```
    for (rueckEnd = v.end()-1; rueckEnd >= v.begin(); rueckEnd--)
       os << *rueckEnd << ",";
    os << "} " << endl << "rückwärts (rbegin): {";
    for (rueckRbegin = v.rbegin(); rueckRbegin < v.rend(); rueckRbegin++)
       os << *rueckRbegin << ",";
    os << "} " << endl;
    return os;
}
int main(void) {
   vector<int> v;
   for (unsigned i = 0; i < 5; i++)
      v.push_back(i);
   cout << v;
}
```

Programm 4.11 liefert die folgende Ausgabe:

```
vorwärts (begin): 0,1,2,3,4,
vorwärts (rend):  0,1,2,3,4,
rückwärts (end):    4,3,2,1,0,
rückwärts (rbegin): 4,3,2,1,0,
```

Spezielle Methoden für bool-Vektoren

Für `bool`-Vektoren werden zusätzlich noch die folgenden Methoden angeboten:

`void flip()`
: invertiert alle Wahrheitswerte im `vector<bool>`-Objekt.

`static void swap(bool& v[i], bool& v[j])`
: vertauscht die beiden Elemente `v[i]` und `v[j]` im `vector<bool>`-Objekt.

Programm 4.12 – `vectorbool.cpp`:
Demoprogramm zu speziellen Methoden für bool-Vektoren

```
#include <vector>
#include <iomanip>
#include <iostream>
using namespace std;
template<typename T>
ostream& operator<< (ostream& os, vector<T> v) {
   typename vector<T>::iterator it;
    os << "[";
    for (it = v.begin(); it != v.end(); it++)
       os << boolalpha << setw(6) << *it << ",";
    os << "]" << endl;
    return os;
}

int main(void) {
   srand(time(0));
   vector<bool> v;
   for (unsigned i = 0; i < 7; i++)
      v.push_back(rand()%2);
```

```
   cout << " Vor flip(): " << v;
   v.flip();
   cout << "Nach flip(): " << v;
   v.swap(v[2], v[5]);
   cout << "Nach swap(): " << v;
}
```

Programm 4.12 liefert z. B. die folgende Ausgabe:

```
 Vor flip(): [  true,  true,  true,  true,  true, false,  true,]
Nach flip(): [ false, false, false, false, false,  true, false,]
Nach swap(): [ false, false,  true, false, false, false, false,]
```

4.3.5 Die sequenzielle Container-Klasse deque

Die Klasse `deque` (*double-ended queue*) ist eine Kombination aus einem Vektor und zwei *Queues*. Sie erlaubt ein Einfügen und Löschen von Elementen sowohl am Anfang als auch am Ende und bietet auch die Vorteile des Direktzugriffs (z. B. Random-Acess-Iteratoren) von Vektoren. Verwendet man diese Klasse, muss man folgende Headerdatei inkludieren:

```
#include <deque>
```

Der Syntax zum Anlegen eines `deque`-Objekts ist z. B. Folgende:

```
deque<T> name(elemZahl)
```

Hiermit wird eine Deque mit dem Namen *name* angelegt, die Elemente vom Typ *T* aufnehmen kann. Ihre Anfangsgröße ist dabei *elemZahl*. Um z. B. eine Deque `meinDeque` der Größe 20 für `double`-Zahlen anzulegen, müsste man folgenden Code angeben:

```
deque<double> meinDeque(20);
```

Konstruktoren der Klasse deque

Folgende Konstruktoren bietet die Templateklasse `deque<T>` an:

`deque()` $O(1)$
: Default-Konstruktor (legt leere Deque an).

`deque(size_type n)` $O(n)$
: legt Deque der Größe `n` an. Hier wird für jedes angelegte Element der Default-Konstruktor aufgerufen, so dass bereits entsprechend viele Elemente existieren, was beim vorherigen Konstruktor nicht zutrifft.

`deque(size_type n, const T& elem)` $O(n)$
: legt Deque mit `n` `elem`-Kopien an.

`deque(const deque&)` $O(n)$
: Kopierkonstruktor

`deque(InputIterator anfang, InputIterator ende)` $O(n)$
: legt eine Deque an, die mit dem Bereich von einschließlich `anfang` bis ausschließlich `ende` (wird nicht mehr kopiert) initialisiert wird.

Programm 4.13 – `dequekonstr.cpp`:
Einfaches Demoprogramm zu den deque-Konstruktoren

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

using namespace std;

template <typename T>
void dequeAusgabe(deque<T>d, string text) {
   cout << setw(12) << text;
   for (unsigned i=0; i < d.size(); i++)
      cout << d[i] << ", ";
   cout << endl;
}
class K {
     double pi;
  public:
     K() { pi = 3.14; }
     double getPi() const { return pi; }
};
int main(void) {
   unsigned i;
   //...............................................1 (Default-Konstruktor)
   deque<string> leerDeque; // legt leere Deque für string-Objekte an
   cout << "leerDeque (Grösse): " << leerDeque.size() << endl;
   //...............................................2 Deque der Grösse n
   deque<double> gltPkt(10);  // legt Deque der Grösse 10 für double an
   dequeAusgabe<double>(gltPkt, "gltPkt: ");
   deque<K>  piDeque(5); // legt Deque der Grösse 5 für K-Objekte an
   cout << setw(12) << "piDeque: ";
   for (i=0; i < piDeque.size(); i++)
      cout << piDeque[i].getPi() << ", ";
   cout << endl;
   //.................................3 (Deque mit n Kopien eines Elements)
   int zahl = 4711;
   deque<int> ganzZahl(10, zahl); // legt Deque mit 10 mal den Wert 4711 an
   dequeAusgabe<int>(ganzZahl, "ganzZahl: ");
   deque<string> gruss(5, string("Hallo")); // initialis. gruss mit 5 "Hallo"
   dequeAusgabe<string>(gruss, "gruss: ");
   //...............................................4 (Kopierkonstruktor)
   deque<string> gruss2(gruss); // legt Deque an, die Kopie von gruss ist
   dequeAusgabe<string>(gruss2, "gruss2: ");
   //..............................5 (Teil-Kopierkonstruktor mit Iteratoren)
   gruss[3] = "Mister";
   deque<string> gruss3(&gruss[2], &gruss[4]); // legt Deque mit 2 Strings
                                               // aus gruss[2] und gruss[3] an
   dequeAusgabe<string>(gruss3, "gruss3: ");
}
```

Programm 4.13 liefert die folgende Ausgabe:

```
leerDeque (Grösse): 0
    gltPkt: 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   piDeque: 3.14, 3.14, 3.14, 3.14, 3.14,
  ganzZahl: 4711, 4711, 4711, 4711, 4711, 4711, 4711, 4711, 4711, 4711,
     gruss: Hallo, Hallo, Hallo, Hallo, Hallo,
    gruss2: Hallo, Hallo, Hallo, Hallo, Hallo,
    gruss3: Hallo, Mister,
```

Zuweisen von deque-Objekten

Um `deque`-Objekte einander zuweisen zu können, wird der folgende Zuweisungsoperator angeboten:

`deque& operator= (const deque&)` $O(n)$

Programm 4.14 – `dequezuw.cpp`:
Demoprogramm zum Zuweisen von deque-Objekten

```
#include <deque>
#include <iostream>
using namespace std;
template <typename T>
ostream& operator<<(ostream& os, const deque<T>& dq) {
   for (unsigned i=0; i < dq.size(); i++)
      os << dq[i] << ", ";
   return os;
}
int main(void) {
   unsigned i;
   deque<int> dq1(5), dq2(10), dq3;
   for (i=0; i < dq1.size(); i++)
      dq1[i] = i;
   cout << "dq1: " << dq1 << endl;
   for (i=0; i < dq2.size(); i++)
      dq2[i] = (i+1)*10;
   cout << "dq2: " << dq2 << endl;
   dq3 = dq2; cout << "dq3: " << dq3 << endl;
   dq2 = dq1; cout << "dq2: " << dq2 << endl;
   dq1 = dq3; cout << "dq1: " << dq1 << endl;
}
```

Programm 4.14 liefert die folgende Ausgabe:

```
dq1: 0, 1, 2, 3, 4,
dq2: 10, 20, 30, 40, 50, 60, 70, 80, 90, 100,
dq3: 10, 20, 30, 40, 50, 60, 70, 80, 90, 100,
dq2: 0, 1, 2, 3, 4,
dq1: 10, 20, 30, 40, 50, 60, 70, 80, 90, 100,
```

Zugriff auf deque-Elemente

Um auf einzelne Elemente in einem `deque`-Objekt lesend bzw. schreibend zuzugreifen, stehen die folgenden Methoden zur Verfügung:

`T& operator[size_type i]` $O(1)$
`const T& operator[size_type i] const` $O(1)$
: liefert Referenz auf *i*. tes Element. Hier wird keine Überprüfung durchgeführt, ob der angegebene Index innerhalb des erlaubten Bereichs liegt.

`T& at(size_type i)` $O(1)$
`const T& at(size_type i) const` $O(1)$
: liefert Referenz auf *i*. tes Element. Sollte sich *i* nicht innerhalb des erlaubten Bereichs befinden, wird hier – anders als beim Indexoperator `[]` – eine `out_of_range`-Exception ausgelöst, die man abfangen kann.

`T& front()` $O(1)$
`const T& front() const` $O(1)$
: liefert Referenz auf erstes Element der Deque; entspricht `dequ.at(0)`. Ein Aufruf dieser Methode führt bei einer leeren Deque zum Programmabsturz.

`T& back()` $O(1)$
`const T& back() const` $O(1)$
: liefert Referenz auf letztes Element der Deque. Ein Aufruf dieser Methode führt bei einer leeren Deque zum Programmabsturz. Diese Methode entspricht dem Aufruf `dequ.at(dequ.size()-1)`.

Programm 4.15 – `dequezugr.cpp`: Zugriffsmöglichkeiten auf deque-Elemente

```
#include <deque>
#include <iostream>
using namespace std;
void dequeAusgabe(deque<int>d, string text) {
   cout << text;
   for (unsigned i=0; i < d.size(); i++)
      cout << d[i] << ", ";
   cout << endl;
}
int main(void) {
   deque<int> ganzZahl(10);

   for (unsigned i=0; i<ganzZahl.size(); i++)
      ganzZahl[i] = (i+1)*(i+1);
   dequeAusgabe(ganzZahl, "ganzZahl: ");

   cout << "Erstes Element: "  << ganzZahl.front() << endl
        << "Letztes Element: " << ganzZahl.back()  << endl;
   ganzZahl.front() = -1000;
   ganzZahl.back() = -1234;
   dequeAusgabe(ganzZahl, "ganzZahl: ");
```

```
   // ganzZahl.at(1000) = 4711; // Hier würde Programm abgebrochen
   // ganzZahl[1000] = 4711; // Hier Speicherüberschreibung
   deque<string> leerDeque; // legt leere Deque für string-Objekte an
   // cout << leerDeque.front(); // führt bei leerer Deque zu Programmabbruch
   // cout << leerDeque.back();  // führt bei leerer Deque zu Programmabbruch
}
```

Programm 4.15 liefert die folgende Ausgabe:

```
ganzZahl: 1, 4, 9, 16, 25, 36, 49, 64, 81, 100,
Erstes Element: 1
Letztes Element: 100
ganzZahl: -1000, 4, 9, 16, 25, 36, 49, 64, 81, -1234,
```

Methoden zum Zuweisen und Einfügen von deque-Elementen

`void assign(size_type n, const T& elem)` $O(n)$
: weist der Deque `n` Mal das Element `elem` zu. Die vorher in der Deque enthaltenen Elemente gehen dabei alle verloren.

`void assign(InputIterator anfang, InputIterator ende)` $O(n)$
: weist der Deque alle Elemente zu, die sich in dem Bereich von einschließlich `anfang` bis ausschließlich `ende` (wird nicht mehr zugewiesen) befinden. Die vorher in der Deque enthaltenen Elemente gehen dabei alle verloren.

`iterator insert(iterator pos, const T& elem)` $O(n)$
: fügt das Element `elem` vor der Position `pos` ein und liefert einen Iterator auf dieses eingefügte Element.

`void insert(iterator pos, size_type n, const T& elem)` $O(n)$
: fügt das Element `elem` `n` Mal vor der Position `pos` ein.

`void insert(iterator pos,InputIterator anf,InputIterator end)` $O(n)$
: fügt vor der Position `pos` alle Elemente ein, die sich in dem Bereich von einschließlich `anf` bis ausschließlich `end` (wird nicht mehr eingefügt) befinden.

`void push_back(const T& elem)` $O(1)$
: fügt das Element `elem` am Ende der Deque an.

`void push_front(const T& elem)` $O(1)$
: fügt Element `elem` am Anfang der Deque ein (**bei `vector` nicht verfügbar**).

Methoden zum Löschen von deque-Elementen

`void clear()` $O(n)$
: löscht alle Elemente der Deque. Die Deque hat danach die Größe 0.

`void pop_back()` $O(1)$
: löscht das letzte Element der Deque.

`void pop_front()` $O(1)$
: löscht das erste Element der Deque **(bei `vector` nicht verfügbar**).

`iterator erase(iterator pos)` $O(n)$

löscht das Element an Iterator-Position `pos`. Als Rückgabewert wird die Position des Elements geliefert, das sich nach dem gelöschten Element befindet.

`iterator erase(iterator anfang, iterator ende)` $O(n)$

löscht alle Elemente, die sich in dem Bereich von einschließlich `anfang` bis ausschließlich `ende` (wird nicht mehr gelöscht) befinden. Als Rückgabewert wird die Position des Elements geliefert, das sich nach zuletzt gelöschtem Element befindet.

Programm 4.16 – `dequeinserase.cpp`: Zuweisen, Einfügen und Löschen von deque-Elementen

```
#include <deque>
#include <iostream>
using namespace std;
//--------------------------------------------------------------------------ausgabe
void ausgabe(deque<char>d, string text="") {
   deque<char>::iterator  it;
   cout << text;
   for (it = d.begin(); it != d.end(); it++)
      cout << *it;
}
//----------------------------------------------------------------------------main
int main(void)
{
   unsigned i;
   deque<char> d1(1000), d2(200);
   //.............................................................assign
   d1.assign(6, 'x');         ausgabe(d1, "d1: "); cout << endl;
   d2.assign(&d1[2], &d1[5]); ausgabe(d2, "d2: "); cout << endl;
   //.............................................................clear
   d2.clear(); cout << "Grösse von d2 nun: " << d2.size() << endl;
   //.............................................push_back / erase
   deque<char> d3;
   for (i=0; i < 5; i++)
      d3.push_back('a'+ i);
   cout << "erase von vorne: ";
   while (d3.begin() != d3.end()) {
      d3.erase(d3.begin());
      ausgabe(d3);  cout << ",  ";
   }
   cout << endl;
   //.............................................................pop_back
   for (i=0; i < 5; i++)
      d3.push_back('a'+ i);
   cout << "pop_back: ";
   while (d3.begin() != d3.end()) {
      d3.pop_back();
      ausgabe(d3);  cout << ",  ";
   }
   cout << endl;
```

```
   //.....................................................insert
   for (i=0; i < 10; i++)
      d3.push_back('a'+ i);
   ausgabe(d3);  cout << endl;
   d3.insert(d3.begin()+2, '-');     ausgabe(d3);  cout << endl;
   d3.insert(d3.begin()+5, 10, '_'); ausgabe(d3);  cout << endl;
   //.....................................................push_front
   deque<char> d4;
   cout << "push_front: ";
   for (i=0; i < 5; i++) {
      d4.push_front('a'+ i);
      ausgabe(d4);  cout << ",  ";
   }
   cout << endl;
   //.....................................................pop_front
   cout << "pop_front: ";
   for (i=0; i < 5; i++) {
      ausgabe(d4);  cout << ",  ";
      d4.pop_front();
   }
}
```

Programm 4.16 liefert die folgende Ausgabe:

```
d1: xxxxxx
d2: xxx
Grösse von d2 nun: 0
erase von vorne: bcde,  cde,  de,  e,  ,
pop_back: abcd,  abc,  ab,  a,  ,
abcdefghij
ab-cdefghij
ab-cd__________efghij
push_front: a,  ba,  cba,  dcba,  edcba,
pop_front: edcba,  dcba,  cba,  ba,  a,
```

Größe von deque-Objekten

Methoden zum Erfragen bzw. Verändern der Größe von `deque`-Objekten sind:

`size_type size() const` $O(1)$
: liefert aktuelle Anzahl von Elementen in der Deque.

`size_type max_size() const` $O(1)$
: liefert (systemabhängige) maximal mögliche Größe für die jeweilige Deque.

`bool empty() const` $O(1)$
: liefert `true`, wenn Deque keine Elemente enthält und ansonsten `false`.

`void resize(size_type n, T elem = T())` $O(n)$
: vergrößert bzw. verkleinert die Deque auf die Größe `n` mit Initialisierung. Bei einer Vergrößerung wird für neue Elemente der Default-Konstruktor aufgerufen oder sie werden mit einer Kopie von `elem` initialisiert.

Programm 4.17 – `dequegroes.cpp`:
Größen von deque-Objekten

```
#include <deque>
#include <iomanip>
#include <iostream>
using namespace std;
template <typename T>
void ausgabe(T d, string text="") {
   typename T::iterator it;
   cout << text;
   for (it = d.begin(); it != d.end(); it++)
      cout << setw(3) << *it << ", ";
   cout << "(" << d.size() << ", " << d.max_size() << ")"  << endl;
}
int main(void) {
   deque<double> d1(5);  ausgabe(d1, "d1: ");
   d1.assign(2, 1.3);    ausgabe(d1, "d1: ");
   d1.resize(10, 9.9);   ausgabe(d1, "d1: ");
   d1.resize(5);         ausgabe(d1, "d1: ");
   deque<int> d2;
   for (unsigned i = 0; i < 5; i++)
      d2.push_back(i);
   while ( !d2.empty() ) {
      cout << d2.front() << ", ";
      d2.pop_front();
   }
}
```

Programm 4.17 liefert z. B. die folgende Ausgabe:

```
d1:   0,   0,   0,   0,   0, (5, 4294967295)
d1: 1.3, 1.3, (2, 4294967295)
d1: 1.3, 1.3, 9.9, 9.9, 9.9, 9.9, 9.9, 9.9, 9.9, 9.9, (10, 4294967295)
d1: 1.3, 1.3, 9.9, 9.9, 9.9, (5, 4294967295)
0, 1, 2, 3, 4,
```

Vergleichen und Vertauschen von deque-Objekten

Die Klasse `deque` bietet folgende überladenen globalen Vergleichsoperatoren an:

`operator==(), operator!=()` $O(n)$

Zwei Deque sind gleich, wenn sie beide den gleichen Elementtyp besitzen, die gleiche Größe haben und die Werte jedes Elementpaares an der gleichen Stelle (Index) den gleichen Wert besitzen. Andernfalls sind die beiden Deques ungleich. Eine mögliche Realisierung dieser beiden Operatoren wäre z. B.:

```
template<typename It1, typename It2> bool gleich(It1 a1, It1 end1, It2 a2) {
   for ( ; a1 != end1; ++a1, ++a2)
      if (!(*a1 == *a2))
         return false;
   return true;
}
```

```
template<typename T>
bool operator== (const deque<T>& d1, const deque<T>& d2) {
   return d1.size() == d2.size() && gleich(d1.begin(), d1.end(), d2.begin());
}
template<typename T>
bool operator!= (const deque<T>& d1, const deque<T>& d2) {
   return !(d1 == d2);
}
```

`operator<(), operator<=(), operator>(), operator>=()` $O(n)$

Der Vergleich von `d1 < d2` liefert `true`, wenn beim paarweisen Vergleich von `d1` und `d2` im ersten Paar, bei dem sich die beiden Werte unterscheiden, der Wert von `d1` kleiner als der von `d2` ist, andernfalls liefert dieser Vergleich `false`. Eine mögliche Realisierung dieser Operatoren wäre z. B.:

```
template<typename It1, typename It2>
bool dequeKleiner(It1 a1, It1 end1, It2 a2, It2 end2) {
   for ( ; a1 != end1 && a2 != end2 ; ++a1, ++a2) {
      if (*a1 < *a2) return true;
      if (*a2 < *a1) return false;
   }
   return a1 == end1 && a2 != end2;
}
template<typename T>
bool operator< (const deque<T>& d1, const deque<T>& d2) {
  return dequeKleiner(d1.begin(), d1.end(), d2.begin(), d2.end());
}
template<typename T>
bool operator<=(const deque<T>& d1, const deque<T>& d2) { return !(d2<d1); }
template<typename T>
bool operator> (const deque<T>& d1, const deque<T>& d2) { return d2 < d1;  }
template<typename T>
bool operator>=(const deque<T>& d1, const deque<T>& d2) { return !(d1<d2); }
```

Zum Vertauschen der Inhalte von zwei `deque`-Objekten steht die folgende Methode zur Verfügung:

`void swap(deque &d)` $O(n)$

vertauscht die Elemente der eigenen Deque mit den Elementen von Deque `d`.

Programm 4.18 – `dequevergl.cpp`:
Demoprogramm zum Vergleichen und Vertauschen von deque-Objekten

```
#include <cstdlib>
#include <ctime>
#include <deque>
#include <iostream>
using namespace std;
template <typename T>
void ausgabe(T d, size_t i) {
   typename T::iterator it;
   cout << "d[" << i << "]: ";
```

```
   for (it = d.begin(); it != d.end(); it++)
      cout << *it << ", ";
   cout << endl;
}
int main(void) {
  unsigned i, j;
  deque<int> d[5];
   srand(time(0));
   for (i=0; i< 5; i++)
      for (j=0; j < 7; j++)
         d[i].push_back(rand()%9+1);
   for (i=0; i< 5; i++)
      ausgabe(d[i], i);
   for (i=0; i< 4; i++) //... einfacher Bubble-Sort
      for (j=i+1; j < 5; j++)
         if (d[i] > d[j])
            d[i].swap(d[j]);
   cout << "...... nach Sortieren" << endl;
   for (i=0; i< 5; i++)
      ausgabe(d[i], i);
}
```

Programm 4.18 liefert z. B. die folgende Ausgabe:

```
d[0]: 6, 8, 6, 1, 9, 8, 4,
d[1]: 9, 3, 9, 6, 8, 4, 7,
d[2]: 1, 6, 4, 9, 4, 4, 6,
d[3]: 1, 9, 1, 7, 3, 4, 9,
d[4]: 2, 1, 9, 7, 9, 5, 5,
...... nach Sortieren
d[0]: 1, 6, 4, 9, 4, 4, 6,
d[1]: 1, 9, 1, 7, 3, 4, 9,
d[2]: 2, 1, 9, 7, 9, 5, 5,
d[3]: 6, 8, 6, 1, 9, 8, 4,
d[4]: 9, 3, 9, 6, 8, 4, 7,
```

Iterator-Methoden der Klasse deque

`iterator begin()` $O(1)$

`const_iterator begin() const` $O(1)$

liefern beide einen Iterator auf das erste Element der Deque.

`iterator end()` $O(1)$

`const_iterator end() const` $O(1)$

liefern beide einen Iterator auf die Position hinter dem letzten Element der Deque.

`reverse_iterator rbegin()` $O(1)$

`const_reverse_iterator rbegin() const` $O(1)$

liefern beide einen *reverse*-Iterator auf das letzte Element der Deque.

`reverse_iterator rend()` $O(1)$

`const_reverse_iterator rend() const` $O(1)$

liefern beide *reverse*-Iterator auf die Position vor dem ersten Element der Deque.

Programm 4.19 – `dequeiter.cpp`:
Demoprogramm zu den Iterator-Methoden von Deques

```
#include <deque>
#include <iostream>
using namespace std;
//--------------------------------------------------------------------operator<<
template<typename T>
ostream& operator<< (ostream& os, deque<T> d) {
    typename deque<T>::iterator vorBegin;
    typename deque<T>::reverse_iterator vorRend;
    typename deque<T>::iterator rueckEnd;
    typename deque<T>::reverse_iterator rueckRbegin;

    os << "vorwärts (begin): {";
    for (vorBegin = d.begin(); vorBegin != d.end(); vorBegin++)
       os << *vorBegin << ",";
    os << "} " << endl;
    os << "vorwärts (rend):  {";
    for (vorRend = d.rend()-1; vorRend >= d.rbegin(); vorRend--)
       os << *vorRend << ",";
    os << "} " << endl;
    os << "rückwärts (end):    {";
    for (rueckEnd = d.end()-1; rueckEnd >= d.begin(); rueckEnd--)
       os << *rueckEnd << ",";
    os << "} " << endl;
    os << "rückwärts (rbegin): {";
    for (rueckRbegin = d.rbegin(); rueckRbegin < d.rend(); rueckRbegin++)
       os << *rueckRbegin << ",";
    os << "} " << endl;

    return os;
}
//-----------------------------------------------------------------------main
int main(void) {
   deque<int> d;
   for (unsigned i = 0; i < 5; i++)
      d.push_back(i);
   cout << d;
}
```

Programm 4.19 liefert die folgende Ausgabe:

```
vorwärts (begin): 0,1,2,3,4,
vorwärts (rend):  0,1,2,3,4,
rückwärts (end):    4,3,2,1,0,
rückwärts (rbegin): 4,3,2,1,0,
```

4.3.6 Die sequenzielle Container-Klasse list

Die Klasse `list` stellt eine doppelt verkettete Liste zur Verfügung. Verwendet man diese Klasse, muss man folgende Headerdatei inkludieren:

```
#include <list>
```

Der Syntax zum Anlegen eines `list`-Objekts ist z. B. Folgende:

```
list<T> name
```

Hiermit wird eine leere doppelt verkettete Liste mit dem Namen *name* angelegt.

Konstruktoren der Klasse list

Folgende Konstruktoren bietet die Templateklasse `list<T>` an:

`list()` $O(1)$

`list(const list&)` $O(n)$

Default-Konstruktor (legt leere Liste an) und Kopierkonstruktor.

`list(size_type n)` $O(n)$

legt Liste der Größe `n` an. Hier wird für jedes angelegte Element der Default-Konstruktor aufgerufen, so dass bereits entsprechend viele Elemente existieren, was beim vorherigen Konstruktor nicht zutrifft.

`list(size_type n, const T& elem)` $O(n)$

legt Liste mit `n` `elem`-Kopien an.

`list(InputIterator anfang, InputIterator ende)` $O(n)$

legt eine Liste an, die mit dem Bereich von einschließlich `anfang` bis ausschließlich `ende` (wird nicht mehr kopiert) initialisiert wird.

Zuweisen von list-Objekten

Um `list`-Objekte einander zuweisen zu können, wird der folgende Zuweisungsoperator angeboten:

`list& operator= (const list&)` $O(n)$

Zugriff auf list-Elemente

Um auf einzelne Elemente in einem `list`-Objekt lesend bzw. schreibend zuzugreifen, stehen die folgenden Methoden zur Verfügung:

`T& front()` $O(1)$

`const T& front() const` $O(1)$

liefern Referenz auf erstes Element der Liste. Ein Aufruf dieser Methode führt bei einer leeren Liste zum Programmabsturz.

`T& back()` $O(1)$

`const T& back() const` $O(1)$

liefern Referenz auf letztes Element der Liste. Ein Aufruf dieser Methode führt bei einer leeren Liste zum Programmabsturz.

Methoden zum Zuweisen und Einfügen von list-Elementen

`void assign(size_type n, const T& elem)` $O(n)$
weist der Liste `n` Mal das Element `elem` zu. Die vorher in der Liste enthaltenen Elemente gehen dabei alle verloren.

`void assign(InputIterator anfang, InputIterator ende)` $O(n)$
weist der Liste alle Elemente zu, die sich in dem Bereich von einschließlich `anfang` bis ausschließlich `ende` (wird nicht mehr zugewiesen) befinden. Die vorher in der Liste enthaltenen Elemente gehen dabei alle verloren.

`iterator insert(iterator pos, const T& elem)` $O(n)$
fügt das Element `elem` vor der Position `pos` ein und liefert einen Iterator auf dieses eingefügte Element.

`void insert(iterator pos, size_type n, const T& elem)` $O(n)$
fügt das Element `elem` `n` Mal vor der Position `pos` ein.

`void insert(iterator pos,InputIterator anf,InputIterator end)` $O(n)$
fügt vor der Position `pos` alle Elemente ein, die sich in dem Bereich von einschließlich `anf` bis ausschließlich `end` (wird nicht mehr eingefügt) befinden.

`void push_front(const T& elem)` $O(1)$
fügt das Element `elem` am Anfang der Liste ein.

`void push_back(const T& elem)` $O(1)$
fügt das Element `elem` am Ende der Liste an.

Methoden zum Löschen von list-Elementen

`void clear()` $O(n)$
löscht alle Elemente der Liste. Die Liste hat danach die Größe 0.

`iterator erase(iterator pos)` $O(1)$
löscht das Element an Iterator-Position `pos`. Als Rückgabewert wird die Position des Elements geliefert, das sich nach dem gelöschten Element befindet.

`iterator erase(iterator anfang, iterator ende)` $O(n)$
löscht alle Elemente, die sich in dem Bereich von einschließlich `anfang` bis ausschließlich `ende` befinden. Als Rückgabewert wird die Position des Elements geliefert, das sich nach dem zuletzt gelöschten Element befindet.

`void pop_back()` $O(1)$
löscht das letzte Element der Liste.

`void pop_front()` $O(1)$
löscht das erste Element der Liste.

Größe von list-Objekten

`size_type size() const` $O(1)$
liefert aktuelle Anzahl von Elementen in der Liste.

`size_type max_size() const` $O(1)$
liefert (systemabhängige) maximal mögliche Größe für die jeweilige Liste.

`bool empty() const` $O(1)$
: liefert `true`, wenn Liste keine Elemente enthält und ansonsten `false`.

`void resize(size_type n, T elem = T())` $O(n)$
: vergrößert bzw. verkleinert die Liste auf Größe `n` mit Initialisierung. Bei einer Vergrößerung wird für neu hinzugefügten Elemente entweder der Default-Konstruktor aufgerufen oder sie werden mit einer Kopie von `elem` initialisiert.

Vergleichen und Vertauschen von list-Objekten

Die Klasse `list` bietet folgende überladenen globalen Vergleichsoperatoren an:

`operator==(), operator!=()` $O(n)$
: Zwei Listen sind gleich, wenn sie beide den gleichen Elementtyp besitzen, die gleiche Größe haben und die Werte jedes Elementpaares an der gleichen Stelle (Index) den gleichen Wert besitzen. Andernfalls sind die beiden Deques ungleich. Eine mögliche Realisierung dieser beiden Operatoren wäre z. B.:

```
template<typename T>
bool operator== (const list<T>& l1, const list<T>& l2) {
   list<T>::const_iterator it1 = l1.begin();
   list<T>::const_iterator it2 = l2.begin();
   list<T>::const_iterator end1 = l1.end();
   list<T>::const_iterator end2 = l2.end();
   while (it1 != end1 && it2 != end2 && *it1 == *it2) {
      ++it1;
      ++it2;
   }
   return it1 == end1 && it2 == end2;
}

template<typename T>
bool operator!= (const list<T>& l1, const list<T>& l2) {
   return !(l1 == l2);
}
```

`operator<(), operator<=(), operator>(), operator>=()` $O(n)$
: Der Vergleich von `l1 < l2` liefert `true`, wenn beim paarweisen Vergleich von `l1` und `l2` im ersten Paar, bei dem sich die beiden Werte unterscheiden, der Wert von `l1` kleiner als der von `l2` ist, andernfalls liefert dieser Vergleich `false`. Eine mögliche Realisierung dieser Operatoren wäre z. B.:

```
template<typename It1, typename It2>
bool listKleiner(It1 a1, It1 end1, It2 a2, It2 end2) {
   for ( ; a1 != end1 && a2 != end2 ; ++a1, ++a2) {
      if (*a1 < *a2) return true;
      if (*a2 < *a1) return false;
   }
   return a1 == end1 && a2 != end2;
}
```

```
template<typename T>
bool operator< (const list<T>& l1, const list<T>& l2) {
  return listKleiner(l1.begin(), l1.end(), l2.begin(), l2.end());
}
template<typename T>
bool operator<=(const list<T>& l1, const list<T>& l2) { return !(l2 < l1); }
template<typename T>
bool operator> (const list<T>& l1, const list<T>& l2) { return   l2 < l1;  }
template<typename T>
bool operator>=(const list<T>& l1, const list<T>& l2) { return !(l1 < l2); }
```

Zum Vertauschen der Inhalte von zwei `list`-Objekten steht die folgende Methode zur Verfügung:

`void swap(list &l)` $O(n)$
: vertauscht die Elemente der eigenen Liste mit den Elementen von Liste `l`.

Programm 4.20 – `listvergl.cpp`:
Demoprogramm zum Vergleichen und Vertauschen von list-Objekten

```
#include <cstdlib>
#include <ctime>
#include <list>
#include <iostream>
using namespace std;
template <typename T>
void ausgabe(T l, size_t i) {
   typename T::iterator it;
   cout << "l[" << i << "]: ";
   for (it = l.begin(); it != l.end(); it++)
      cout << *it << ", ";
   cout << endl;
}

int main(void) {
  unsigned i, j;
  list<int> l[5];
   srand(time(0));
   for (i=0; i< 5; i++)
      for (j=0; j < 7; j++)
         l[i].push_back(rand()%9+1);
   for (i=0; i< 5; i++)
      ausgabe(l[i], i);
   for (i=0; i< 4; i++) //... einfacher Bubble-Sort
      for (j=i+1; j < 5; j++)
         if (l[i] > l[j])
            l[i].swap(l[j]);
   cout << "...... nach Sortieren" << endl;
   for (i=0; i< 5; i++)
      ausgabe(l[i], i);
}
```

Programm 4.20 liefert z. B. die folgende Ausgabe:

```
l[0]: 1, 4, 9, 1, 9, 7, 5,
l[1]: 3, 8, 2, 3, 8, 1, 2,
l[2]: 2, 3, 5, 7, 9, 8, 6,
l[3]: 8, 5, 1, 4, 7, 1, 7,
l[4]: 1, 3, 9, 1, 7, 8, 1,
...... nach Sortieren
l[0]: 1, 3, 9, 1, 7, 8, 1,
l[1]: 1, 4, 9, 1, 9, 7, 5,
l[2]: 2, 3, 5, 7, 9, 8, 6,
l[3]: 3, 8, 2, 3, 8, 1, 2,
l[4]: 8, 5, 1, 4, 7, 1, 7,
```

Iterator-Methoden der Klasse list

`iterator begin()` $O(1)$

`const_iterator begin() const` $O(1)$

liefern beide einen Iterator auf das erste Element der Liste.

`iterator end()` $O(1)$

`const_iterator end() const` $O(1)$

liefern beide einen Iterator auf die Position hinter dem letzten Element der Liste.

`reverse_iterator rbegin()` $O(1)$

`const_reverse_iterator rbegin() const` $O(1)$

liefern beide einen *reverse*-Iterator auf das letzte Element der Liste.

`reverse_iterator rend()` $O(1)$

`const_reverse_iterator rend() const` $O(1)$

liefern beide einen *reverse*-Iterator auf die Position vor dem ersten Element der Liste.

Löschen aller Vorkommen eines Werts und bedingtes Löschen von Werten

Hierzu bietet die Klasse `list` die beiden folgenden Methoden an:

`void remove(const T& wert)` $O(n)$

entfernt alle Vorkommen des Werts `wert` aus der Liste.

`void remove_if(bool fkt(T& elem))` $O(n)$

entfernt alle Vorkommen aus der Liste, für welche die Funktion `fkt()` `true` liefert. `fkt()` wird für jedes Element mit diesem als Argument aufgerufen.

Programm 4.21 – `listremove.cpp`:
Demoprogramm zu remove() und remove_if()

```
#include <string>
#include <list>
#include <iostream>
using namespace std;

int teiler; // globale Variable :-((
```

```
bool teilbar(int z) { return z > teiler && z % teiler == 0; }
template <typename T>
void ausgabe(T l) {
   for (typename T::iterator it = l.begin(); it != l.end(); it++)
      cout << *it << ", ";
   cout << endl;
}

int main(void)
{
   unsigned i;

   list<string> stringListe;
   stringListe.push_back(string("du"));
   stringListe.push_back(string("bist"));
   stringListe.push_back(string("du"));
   stringListe.push_back(string("und"));
   stringListe.push_back(string("ich"));
   stringListe.push_back(string("bin"));
   stringListe.push_back(string("ich"));  ausgabe(stringListe);
   stringListe.remove(string("du"));   ausgabe(stringListe);
   stringListe.remove(string("ich"));  ausgabe(stringListe);
   list<int> primZahlen;
   for (i=2; i < 50; i++)
      primZahlen.push_back(i);
   for (i=2; i < 25; i++) { // Sieb des Erathostenes
      teiler = i;
      primZahlen.remove_if(teilbar);
   }
   ausgabe(primZahlen);
}
```

Programm 4.21 liefert die folgende Ausgabe:

```
du, bist, du, und, ich, bin, ich,
bist, und, ich, bin, ich,
bist, und, bin,
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
```

Sortieren einer Liste

Hierzu bietet die Klasse `list` die beiden folgenden Methoden an, die beide stabil sortieren, was bedeutet, dass die Reihenfolge gleicher Elemente zueinander erhalten bleibt:

`void sort()` $O(n \log n)$
: sortiert die Liste entsprechend dem Operator `operator<()`.

`void sort(verglFkt(T& elem1, T& elem2))` $O(n \log n)$
: sortiert die Liste entsprechend der Funktion `verglFkt()`. Diese Funktion wird immer mit zwei zu vergleichenden Elemente aus der Liste aufgerufen. Über den Rückgabewert kann man hier steuern, wie diese beiden Elemente zueinander anzuordnen sind.

Programm 4.22 – `listsort.cpp`:
Demoprogramm zu den beiden sort()-Methoden

```
#include <cstdlib>
#include <ctime>
#include <string>
#include <list>
#include <iomanip>
#include <iostream>
using namespace std;
bool groesser(int l, int r) { return l > r; }

template <typename T>
void ausgabe(T l, string str) {
   cout << setw(20) << str;
   for (typename T::iterator it = l.begin(); it != l.end(); it++)
      cout << setw(2) << *it << ", ";
   cout << endl;
}

int main(void) {
   unsigned i;

   list<string> stringListe;
   stringListe.push_back(string("Zorro"));
   stringListe.push_back(string("Hans"));
   stringListe.push_back(string("Emil"));
   stringListe.push_back(string("Adam"));
   stringListe.push_back(string("Peter"));
   ausgabe(stringListe, string("unsortiert: "));
   stringListe.sort();
   ausgabe(stringListe, string("sortiert: "));
   srand(time(0));
   list<int> zahlen;
   for (i=1; i <= 10; i++)
      zahlen.push_back(rand()%50);
   ausgabe(zahlen, string("unsortiert: "));
   zahlen.sort();          ausgabe(zahlen, string("aufwaerts sortiert: "));
   zahlen.sort(groesser); ausgabe(zahlen, string("abwaerts sortiert: "));
}
```

Programm 4.22 liefert z. B. die folgende Ausgabe:

```
        unsortiert: Zorro, Hans, Emil, Adam, Peter,
          sortiert: Adam, Emil, Hans, Peter, Zorro,
        unsortiert: 43, 19, 45, 45, 30, 49,  7, 11,  4, 39,
aufwaerts sortiert:  4,  7, 11, 19, 30, 39, 43, 45, 45, 49,
 abwaerts sortiert: 49, 45, 45, 43, 39, 30, 19, 11,  7,  4,
```

Umkehren der Reihenfolge der Elemente in einer Liste

Hierzu bietet die Klasse `list` die folgende Methode an:

`void reverse()` $O(n)$
: dreht die Reihenfolge der Elemente in einer Liste um.

Programm 4.23 – `listreverse.cpp`: Demoprogramm zur Methode reverse()

```
#include <cstdlib>
#include <ctime>
#include <string>
#include <list>
#include <iomanip>
#include <iostream>
using namespace std;
template <typename T>
void ausgabe(T l, string str) {
   typename T::iterator it;
   cout << setw(20) << str;
   for (it = l.begin(); it != l.end(); it++)
      cout << setw(2) << *it << ", ";
   cout << endl;
}
int main(void) {
  unsigned i;
   list<string> stringListe;
   stringListe.push_back(string("Zorro"));
   stringListe.push_back(string("Hans"));
   stringListe.push_back(string("Emil"));
   stringListe.push_back(string("Adam"));
   stringListe.push_back(string("Peter"));
   ausgabe(stringListe, string("urspruenglich: "));
   stringListe.reverse();
   ausgabe(stringListe, string("umgedreht: "));
   srand(time(0));
   list<int> zahlen;
   for (i=1; i <= 10; i++)
      zahlen.push_back(rand()%50);
   ausgabe(zahlen, string("unsortiert: "));
   zahlen.sort();     ausgabe(zahlen, string("aufwaerts sortiert: "));
   zahlen.reverse();  ausgabe(zahlen, string("abwaerts sortiert: "));
}
```

Programm 4.23 liefert z. B. die folgende Ausgabe:

```
     urspruenglich: Zorro, Hans, Emil, Adam, Peter,
         umgedreht: Peter, Adam, Emil, Hans, Zorro,
        unsortiert: 36, 12,  3,  9, 30, 30, 21, 48, 49, 29,
aufwaerts sortiert:  3,  9, 12, 21, 29, 30, 30, 36, 48, 49,
 abwaerts sortiert: 49, 48, 36, 30, 30, 29, 21, 12,  9,  3,
```

Verschieben der Elemente aus einer Liste in eine andere Liste

Hierzu bietet die Klasse `list` die folgenden Methoden an:

`void splice(iterator pos, list& l)` $O(1)$
: leert die Liste `l`, indem es alle Elemente der Liste `l` vor der Position `pos` der eigenen Liste einfügt. Bei `l` muss es sich um eine andere Liste (`&l != this`) handeln.

`void splice(iterator pos, list& l, iterator liter)` $O(1)$
: enfernt aus der Liste `l` das Element, auf das der Iterator `liter` zeigt, und fügt es vor der Position `pos` in der eigenen Liste ein. Bei `pos == liter` hat diese Methode keine Auswirkung.

`void splice(iterator pos, list& l, iterator anf,iterator end)` $O(1)$
: entfernt aus der Liste `l` alle Elemente, die sich in dem Bereich von einschließlich `anf` bis ausschließlich `end` (wird nicht mehr entfernt) befinden, und fügt sie vor der Position `pos` in der eigenen Liste ein. `pos` darf kein Iterator aus dem Bereich von einschließlich `anf` bis ausschließlich `end` sein.

Programm 4.24 – `listsplice.cpp`:
Demoprogramm zu den splice()-Methoden

```
#include <list>
#include <iostream>
using namespace std;
template <typename T>
void ausgabe(T l, string str) {
   typename T::iterator it;
   cout << str;
   for (it = l.begin(); it != l.end(); it++)
      cout << *it << ", ";
   cout << endl;
}
int main(void) {
  unsigned i;
   list<char> eins, zwei, drei, vier;
   list<char>::iterator it1, it2, it3;
   for (i=0; i < 6; i++) {
      eins.push_back('a'+i);
      zwei.push_back('0'+i);
      drei.push_back('0'+i);
      vier.push_back('A'+i);
   }
   ausgabe(eins, string("eins: "));
   ausgabe(zwei, string("zwei: "));
   ausgabe(drei, string("drei: "));
   ausgabe(vier, string("vier: "));
   cout << "-------------- nach eins.splice(++eins.begin(), zwei);" << endl;
   eins.splice(++eins.begin(), zwei);
   ausgabe(eins, string("eins: "));
   ausgabe(zwei, string("zwei: "));
```

```
   cout << "---------nach: it1 = drei.begin(); it1++ it1++" << endl;
   cout << "               it2 = eins.begin(); it2-- it2-- it2-- it2--" << endl;
   cout << "               drei.splice(it1, zwei, it2);" << endl;
   it1 = drei.begin(); it1++; it1++;
   it2 = eins.end();   it2--; it2--; it2--; it2--;

   drei.splice(it1, eins, it2);
   ausgabe(eins, string("eins: "));
   ausgabe(drei, string("drei: "));

   cout << "---------nach: it1 = drei.begin(); it1++ it1++" << endl;
   cout << "               it2 = vier.begin(); it2++ it2++ it2++ it2++" << endl;
   cout << "               drei.splice(it1, vier, vier.begin(), it2);" << endl;
   it1 = drei.begin(); it1++; it1++;
   it2 = vier.begin(); it2++; it2++; it2++; it2++;

   drei.splice(it1, vier, vier.begin(), it2);
   ausgabe(drei, string("drei: "));
   ausgabe(vier, string("vier: "));
}
```

Programm 4.24 liefert die folgende Ausgabe:

```
eins: a, b, c, d, e, f,
zwei: 0, 1, 2, 3, 4, 5,
drei: 0, 1, 2, 3, 4, 5,
vier: A, B, C, D, E, F,
-------------- nach eins.splice(++eins.begin(), zwei);
eins: a, 0, 1, 2, 3, 4, 5, b, c, d, e, f,
zwei:
---------nach: it1 = drei.begin(); it1++ it1++
               it2 = eins.begin(); it2-- it2-- it2-- it2--
               drei.splice(it1, zwei, it2);
eins: a, 0, 1, 2, 3, 4, 5, b, d, e, f,
drei: 0, 1, c, 2, 3, 4, 5,
---------nach: it1 = drei.begin(); it1++ it1++
               it2 = vier.begin(); it2++ it2++ it2++ it2++
               drei.splice(it1, vier, vier.begin(), it2);
drei: 0, 1, A, B, C, D, c, 2, 3, 4, 5,
vier: E, F,
```

Mischen zweier Listen

Hierzu bietet die Klasse `list` die folgenden Methoden an:

`void merge(list& l)` $O(n)$

Bei dieser Methode müssen beide Listen sortiert (entsprechend `operator<`) und verschieden (`&l != this`) sein. Diese Methode entfernt dann alle Elemente aus der Liste `l` und sortiert sie in die eigene Liste ein.

`void merge(list& l, verglFkt(T& elem1, T& elem2))` $O(n)$

Hier müssen die beiden Listen auch wieder verschieden (`&l != this`) sein, aber entsprechend der Funktion `verglFkt()` sortiert sein. Diese Methode entfernt dann alle Elemente aus der Liste `l` und sortiert sie in die eigene Liste ein. Die Funktion `verglFkt()` wird immer mit zwei zu vergleichenden Elemente aus den Listen aufgerufen. Über den Rückgabewert kann man hier steuern, wie diese beiden Elemente zueinander anzuordnen sind.

Das Mischen erfolgt bei beiden Methoden stabil, was bedeutet, dass bei gleichen Elementen neu eingefügte Elemente nach denen der eigenen Elemente eingefügt werden.

Programm 4.25 – `listmerge.cpp`:
Demoprogramm zu den beiden merge()-Methoden

```
#include <string>
#include <list>
#include <iostream>
using namespace std;
bool groesser(string l, string r) { return l > r; }
template <typename T>
void ausgabe(T l, string str) {
   typename T::iterator it;
   cout << str;
   for (it = l.begin(); it != l.end(); it++)
      cout << *it << ", ";
   cout << endl;
}
int main(void) {
  list<string> mann1, mann2, frau1, frau2;
   mann1.push_back(string("Zorro")); mann1.push_back(string("Emil"));
   mann1.push_back(string("Adam"));  mann1.push_back(string("Peter"));
   mann2 = mann1;
   mann1.sort();
   ausgabe(mann1, string("mann1: "));
   frau1.push_back(string("Zara")); frau1.push_back(string("Emilia"));
   frau1.push_back(string("Berta"));
   frau2 = frau1;
   frau1.sort();
   ausgabe(frau1, string("frau1: "));
   mann1.merge(frau1);
   ausgabe(mann1, string("mann1: "));
   ausgabe(frau1, string("frau1: "));
   cout << "--------------------------------------------------" << endl;
   mann2.sort(groesser);
   frau2.sort(groesser);
   ausgabe(mann2, string("mann2: "));
   ausgabe(frau2, string("frau2: "));
   frau2.merge(mann2, groesser);
   ausgabe(mann2, string("mann2: "));
   ausgabe(frau2, string("frau2: "));
}
```

Programm 4.25 liefert die folgende Ausgabe:

```
mann1: Adam, Emil, Peter, Zorro,
frau1: Berta, Emilia, Zara,
mann1: Adam, Berta, Emil, Emilia, Peter, Zara, Zorro,
frau1:
-------------------------------------------------
mann2: Zorro, Peter, Emil, Adam,
frau2: Zara, Emilia, Berta,
mann2:
frau2: Zorro, Zara, Peter, Emilia, Emil, Berta, Adam,
```

Entfernen gleicher aufeinanderfolgender Elemente

Hierzu bietet die Klasse `list` die folgenden Methoden an:

`void unique()` $O(n)$

Kommen mehrere gleiche Elemente nacheinander in der Liste vor, so entfernt diese Methode jeweils bis auf das erste Element alle folgenden gleichen Elemente.

`void unique(gleich(T& elem1, T& elem2))` $O(n)$

Kommen mehrere gleiche Elemente nacheinander in der Liste vor, so entfernt diese Methode jeweils bis auf das erste Element alle folgenden gleichen Elemente. Anders als bei der vorherigen Methode kann man hier eine eigene Funktion `gleich()` angeben, die immer mit zwei zu vergleichenden Elemente aus der Liste aufgerufen wird. Über den Rückgabewert kann man hier steuern, ob diese beiden Elemente als gleich anzusehen sind, oder nicht.

Programm 4.26 – `listunique.cpp`:
Demoprogramm zu den beiden unique()-Methoden

```
#include <cstdlib>
#include <ctime>
#include <string>
#include <list>
#include <iostream>

using namespace std;

//.... gelten als gleich, wenn Distanz ihrer ASCII-Werte nicht grösser als 2
bool gleich(char l, char r) { int diff = abs(l-r); return diff <= 2; }

//-------------------------------------------------------------------------ausgabe
template <typename T>
void ausgabe(T l, string str) {
   typename T::iterator it;
   cout << str;
   for (it = l.begin(); it != l.end(); it++)
      cout << *it;
   cout << endl;
}
```

```
//-------------------------------------------------------------------------main
int main(void) {
   unsigned i;

   srand(time(0));
   list<char> buchst1, buchst2,
              buchst3, buchst4;
   for (i=1; i <= 40; i++)
      buchst1.push_back(rand()%10+'a');
   buchst4 = buchst3 = buchst2 = buchst1;

   ausgabe(buchst1, string("buchst1: "));
   buchst1.unique(); ausgabe(buchst1, string("buchst1: "));
   cout << "-------------------------------------------------" << endl;
   buchst2.sort();   ausgabe(buchst2, string("buchst2: "));
   buchst2.unique(); ausgabe(buchst2, string("buchst2: "));
   cout << "-------------------------------------------------" << endl;
   ausgabe(buchst3, string("buchst3: "));
   buchst3.unique(gleich);  ausgabe(buchst3, string("buchst3: "));
   cout << "-------------------------------------------------" << endl;
   buchst4.sort();          ausgabe(buchst4, string("buchst4: "));
   buchst4.unique(gleich);  ausgabe(buchst4, string("buchst4: "));
}
```

Programm 4.26 liefert z. B. die folgende Ausgabe:

```
buchst1: bcdefhcefaiidejgdijdjjbffihgifjbjeffbibi
buchst1: bcdefhcefaidejgdijdjbfihgifjbjefbibi
-------------------------------------------------
buchst2: abbbbbccddddeeeeffffffffgghhiiiiiiijjjjjj
buchst2: abcdefghij
-------------------------------------------------
buchst3: bcdefhcefaiidejgdijdjjbffihgifjbjeffbibi
buchst3: behcfaidjgdidjbfifjbjebibi
-------------------------------------------------
buchst4: abbbbbccddddeeeeffffffffgghhiiiiiiijjjjjj
buchst4: adgj
```

4.3.7 Die Container-Adaptorklasse stack

Bei der Klasse `stack` handelt es sich um eine *LIFO*-Datenstruktur (*Last-In-First-Out*). Verwendet man *Stacks*, muss man folgende Headerdatei inkludieren:

```
#include <stack>
```

Der Syntax zum Anlegen eines `stack`-Objekts ist z. B. Folgende:

```
stack<T> name
```

Hiermit wird ein leeres `stack`-Objekt *name* angelegt, das Elemente vom Typ *T* aufnehmen kann.

Konstruktoren und Methoden der Klasse stack

Folgende Konstruktoren und Methoden bietet die Templateklasse `stack` an:

`stack()` $O(1)$

`stack(const stack&)` $O(n)$

Default-Konstruktor (legt leeren Stack an) und Kopierkonstruktor.

`void push(const T& elem)` $O(1)$

legt das Element `elem` an oberster Stelle im Stack ab.

`void pop()` $O(1)$

entfernt das oberste Element aus dem Stack, liefert es aber nicht zurück.

`T& top()` $O(1)$

`const T& top() const` $O(1)$

liefern beide eine Referenz auf das oberste Element, entfernen dieses Element aber nicht aus dem Stack. Bei der zweiten Variante kann dieses Element nicht über die zurückgelieferte Referenz geändert werden.

`size_type size() const` $O(1)$

liefert aktuelle Anzahl von Elementen im Stack.

`bool empty() const` $O(1)$

liefert `true`, wenn der Stack keine Elemente enthält und ansonsten `false`.

Neben dem Kopierkonstruktor wird zusätzlich vom Compiler automatisch der Zuweisungsoperator generiert.

Programm 4.27 – `stack.cpp`:
Einfaches Demoprogramm zur Klasse stack

```
#include <stack>
#include <iostream>
using namespace std;

int main(void) {
   stack<int> s;

   for (unsigned i=0; i < 10; i++)
      s.push(i);
   cout << "Stack hat die Grösse " << s.size() << endl;
   while ( !s.empty() ) {
      cout << s.top() << ", ";
      s.pop();
   }
}
```

Programm 4.27 liefert die folgende Ausgabe:

```
Stack hat die Grösse 10
9, 8, 7, 6, 5, 4, 3, 2, 1, 0,
```

Vergleichen von stack-Objekten

Folgende überladene globalen Vergleichsoperatoren stehen für `stack`-Objekte zur Verfügung:

`operator==(), operator!=()`
: Zwei Stacks sind gleich, wenn sie beide die gleiche Größe haben und die Werte jedes Elementpaares an der gleichen Stelle den gleichen Wert besitzen. Andernfalls sind die beiden ungleich.

`operator<(), operator<=(), operator>(), operator>=()`
: Der Vergleich von zwei Stacks mit `operator<` liefert `true`, wenn beim paarweisen Vergleich im ersten Paar, bei dem sich die beiden Werte unterscheiden, der Wert im ersten Stack kleiner als der in der zweiten ist, andernfalls liefert dieser Vergleich `false`. Die anderen Operatoren lassen sich dann wieder aus dem `operator<` ableiten:

```
template<typename T>
bool operator<=(const stack<T>& s1, const stack<T>& s2) { return !(s2<s1); }
template<typename T>
bool operator> (const stack<T>& s1, const stack<T>& s2) { return s2 < s1;  }
template<typename T>
bool operator>=(const stack<T>& s1, const stack<T>& s2) { return !(s1<s2); }
```

Programm 4.28 – `stackvergl.cpp`:
Demoprogramm zum Vergleichen von stack-Objekten

```
#include <cstdlib>
#include <ctime>
#include <string>
#include <stack>
#include <iostream>
using namespace std;

template <typename T>
void ausgabe(T& s) {
   while ( !s.empty() ) {
      cout << s.top() << ", ";
      s.pop();
   }
}
int main(void) {
  string     str;
  stack<int> s1, s2;
   srand(time(0));
   for (unsigned i = 1; i <= 5; i++) {
      for (unsigned j=0; j < 3; j++) {
         s1.push(rand()%4+1);
         s2.push(rand()%4+1);
      }
      str.clear();
      if (s1 == s2) str += "== ";
```

```
      if (s1 != s2) str += "!= ";
      if (s1 <  s2) str += "< ";
      if (s1 <= s2) str += "<= ";
      if (s1 >  s2) str += "> ";
      if (s1 >= s2) str += ">= ";
      ausgabe(s1); cout << " " << str << endl;
      ausgabe(s2); cout << endl;
   }
}
```

Programm 4.28 liefert z. B. die folgende Ausgabe:

```
4, 2, 3,  != > >=
1, 4, 2,
-------------------
4, 3, 2,  != < <=
3, 3, 3,
-------------------
2, 1, 3,  != > >=
3, 4, 1,
-------------------
1, 1, 2,  != < <=
1, 2, 4,
-------------------
1, 3, 3,  == <= >=
1, 3, 3,
```

Interne Realisierung eines Stacks

Bei der Klasse `stack` handelt es sich um eine so genannte *Container-Adaptorklasse*, die sich intern unter Verwendung der sequenziellen Containerklassen `deque`, `vector` oder `list` realisieren lässt. Die Voreinstellung ist, dass die Klasse `stack` intern als `deque` realisiert wird:

```
  stack<T, deque<T> >  deqStack; // Voreinst. --> entspr.: stack<T> deqStack;
```

Möchte man eine andere Realisierung, muss man folgende Syntax verwenden:

```
  stack<T, vector<T> > vektorStack;
  stack<T, list<T> >   listStack;
```

Programm 4.29 gibt die Zeilen der Datei `stack.cpp` in umgekehrter Reihenfolge aus.

Programm 4.29 – `stackdeque.cpp`:
Interne Realisierung eines Stacks als deque

```
1  #include <fstream>
2  #include <list>
3  #include <stack>
4  #include <string>
5  #include <vector>
6  #include <iostream>
7  using namespace std;
```

```
 8
 9  int main(void)
10  {
11     ifstream eing("stack.cpp");
12     stack<string> stapel; // entspricht: stack<string, deque<string> > stapel;
13
14     //....Datei zeilenweise lesen und Zeilen im Stack speichern
15     string zeile;
16     while (getline(eing, zeile))
17        stapel.push(zeile + "\n");
18
19     //....Im Stack gespeicherten Zeilen ausgeben
20     while (!stapel.empty()) {
21        cout << stapel.top();
22        stapel.pop();
23     }
24  }
```

Alternativ könnte man den Stack in Programm 4.29 auch intern realisieren lassen als:

- Vektor, indem man die Zeile 12 wie folgt angibt:

```
12    stack<string, vector<string> > stapel;
```

- Liste, indem man die Zeile 12 wie folgt angibt:

```
12    stack<string, list<string> > stapel;
```

4.3.8 Die Container-Adaptorklasse queue

Eine weitere grundlegende Datenstruktur in der Informatik neben dem *Stack* ist die so genannte *Queue*, die eine Warteschlange simuliert, wie z. B. vor einem Postschalter oder einem Eintritt zu einer Veranstaltung.
Während ein Stack nach dem *LIFO*-Prinzip („*last in, first out*“) arbeitet, arbeitet eine Queue nach dem *FIFO*-Prinzip („*first in, first out*“).
Die Klasse `queue` ist eine eingeschränkte Forem einer `deque`, bei der man – anders als bei der Klasse `deque` – nur neue Elemente am einen Ende der Warteschlange einfügen und am anderen Ende entfernen kann. Grundsätzlich kann man überall dort, wo man die Klasse `queue` verwendet, auch die Klasse `deque` verwenden, wobei die Klasse `deque` über mehr Funktionalität verfügt. Um herauszustellen, dass man eine reine *FIFO*-Warteschlange benutzt, empfiehlt sich die Verwendung der Klasse `queue`.
Verwendet man die Klasse `queue`, muss man folgende Headerdatei inkludieren:

```
#include <queue>
```

Der Syntax zum Anlegen eines `queue`-Objekts ist z. B. Folgende:

```
queue<T> name
```

Hiermit wird ein leeres `queue`-Objekt mit den Namen *name* angelegt, das Elemente vom Typ *T* aufnehmen kann.

Konstruktoren und Methoden der Klasse queue

Folgende Konstruktoren und Methoden bietet die Templateklasse `queue` an:

`queue()` $O(1)$

`queue(const queue&)` $O(n)$

Default-Konstruktor (legt leere Queue an) und Kopierkonstruktor.

`void push(const T& elem)` $O(1)$

fügt das Element `elem` an letzter Stelle in der Queue ein.

`void pop()` $O(1)$

entfernt das Element an erster Stelle aus der Queue, liefert es aber nicht zurück.

`T& front()` $O(1)$

`const T& front() const` $O(1)$

liefern beide eine Referenz auf das an erster Stelle stehende Element der Queue, entfernen dieses Element aber nicht aus der Queue. Bei der zweiten Variante kann dieses Element nicht über die zurückgelieferte Referenz geändert werden.

`T& back()` $O(1)$

`const T& back() const` $O(1)$

liefern beide eine Referenz auf das an letzter Stelle stehende Element der Queue, entfernen dieses Element aber nicht aus der Queue. Bei der zweiten Variante kann dieses Element nicht über die zurückgelieferte Referenz geändert werden.

`size_type size() const` $O(1)$

liefert aktuelle Anzahl von Elementen in der Queue.

`bool empty() const` $O(1)$

liefert `true`, wenn die Queue keine Elemente enthält und ansonsten `false`.

Neben dem Kopierkonstruktor wird zusätzlich vom Compiler automatisch der Zuweisungsoperator generiert.

Programm 4.30 – `queue.cpp`:
Einfaches Demoprogramm zur Klasse queue

```
#include <queue>
#include <iostream>
using namespace std;
int main(void) {
   queue<int> gerade, ungerade;
   for (unsigned i=1; i <= 21; i++ )
      if (i%2==0)
         gerade.push(i);
      else
         ungerade.push(i);
   cout << "....gerade-Queue hat die Grösse " << gerade.size() << endl;
   while ( !gerade.empty() ) {
      cout << gerade.front() << " ";
      gerade.pop();
   }
   cout <<endl << "....ungerade-Queue hat die Grösse " << ungerade.size() << endl;
```

```
   while ( !ungerade.empty() ) {
      cout << ungerade.front() << " ";
      ungerade.pop();
   }
}
```

Programm 4.30 liefert die folgende Ausgabe:

```
....gerade-Queue hat die Grösse 10
2 4 6 8 10 12 14 16 18 20
....ungerade-Queue hat die Grösse 11
1 3 5 7 9 11 13 15 17 19 21
```

Vergleichen von queue-Objekten

Folgende globalen Vergleichsoperatoren stehen für `queue`-Objekten zur Verfügung:

`operator==(), operator!=()` $O(n)$

Zwei Queues sind nur dann gleich, wenn sie die gleiche Größe haben und die Werte jedes Elementpaares an der gleichen Stelle den gleichen Wert besitzen.

`operator<(), operator<=(), operator>(), operator>=()` $O(n)$

Der Vergleich von zwei Queues mit `operator<` liefert `true`, wenn beim paarweisen Vergleich im ersten Paar, bei dem sich die beiden Werte unterscheiden, der Wert in der ersten Queue kleiner als der in der zweiten ist, andernfalls liefert dieser Vergleich `false`. Die anderen Operatoren lassen sich dann wieder aus dem `operator<` ableiten:

```
template<typename T>
bool operator<=(const queue<T>& q1, const queue<T>& q2) { return !(q2<q1); }
template<typename T>
bool operator> (const queue<T>& q1, const queue<T>& q2) { return q2 < q1;  }
template<typename T>
bool operator>=(const queue<T>& q1, const queue<T>& q2) { return !(q1<q2); }
```

Programm 4.31 – `queuevergl.cpp`:
Demoprogramm zum Vergleichen von queue-Objekten

```
#include <cstdlib>
#include <string>
#include <queue>
#include <iostream>
using namespace std;
template <typename T>
void ausgabe(T& q) {
   while ( !q.empty() ) {
      cout << q.front() << " ";
      q.pop();
   }
}
int main(void) {
  string     str;
  queue<int> q1, q2;
```

```
   for (unsigned i = 1; i <= 5; i++) {
      for (unsigned j=0; j < 3; j++) {
         q1.push(rand()%4+1);
         q2.push(rand()%4+1);
      }
      str.clear();
      if (q1 <  q2) str += "< ";
      if (q1 <= q2) str += "<= ";
      if (q1 >  q2) str += "> ";
      if (q1 >= q2) str += ">= ";
      if (q1 == q2) str += "== ";
      if (q1 != q2) str += "!= ";
      ausgabe(q1); cout << " " << str << endl;
      ausgabe(q2); cout << endl;
   }
}
```

Programm 4.31 liefert z. B. die folgende Ausgabe:

```
1 1 3  < <= !=
3 2 2
-------------------
4 2 2  <= >= ==
4 2 2
-------------------
1 4 4  < <= !=
4 1 4
-------------------
3 3 2  < <= !=
4 3 3
-------------------
2 2 2  < <= !=
4 4 2
```

Interne Realisierung einer Queue

Bei der Klasse `queue` handelt es sich um eine so genannte *Container-Adaptorklasse*, die sich intern unter Verwendung der sequenziellen Containerklassen `deque` oder `list` realisieren lässt. Die Voreinstellung ist, dass die Klasse `queue` intern als `deque` realisiert wird, was wohl in den meisten Fällen auch gewünscht ist:

```
  queue<T, deque<T> >  deqQueue; // Voreinst. --> entspr.: queue<T> deqQueue;
```

Möchte man aber aus welchen Gründen auch immer eine andere Realisierung, muss man folgende Syntax verwenden:

```
  queue<T, list<T> >   listQueue;
```

Programm 4.32 gibt die Zeilen der Datei `queuedeq.cpp` aus.

Programm 4.32 – queuedeq.cpp:
Interne Realisierung einer Queue als deque

```
 1  #include <fstream>
 2  #include <queue>
 3  #include <deque>
 4  #include <list>
 5  #include <string>
 6  #include <iostream>
 7  using namespace std;
 8
 9  int main(void)
10  {
11     ifstream eing("queuedeq.cpp");
12     queue<string> q; // entspricht: queue<string, deque<string> > q;
13
14     //....Datei zeilenweise lesen und Zeilen in Queue speichern
15     string zeile;
16     while (getline(eing, zeile))
17        q.push(zeile + "\n");
18
19     //....In Queue gespeicherten Zeilen ausgeben
20     while (!q.empty()) {
21        cout << q.front();
22        q.pop();
23     }
24  }
```

Alternativ könnte man die Queue in Programm 4.32 auch intern als Liste realisieren lassen als, indem man die Zeile 12 wie folgt angibt:

```
12    queue<string, list<string> > q;
```

4.3.9 Die Container-Adaptorklasse priority_queue

Eine weitere Datenstruktur, die die STL über die Klasse `priority_queue` zur Verfügung stellt, ist die so genannte *Prioritäts-Warteschlange*. Eine Prioritäts-Warteschlange ist eine besondere Form einer `queue`, da hier die Elemente – unabhängig von der Reihenfolge des Einfügens – entsprechend eines vorgegebenen Sortierkriteriums in der Warteschlange eingefügt werden. Ein Beispiel für eine Prioritäts-Warteschlange wäre eine Auto-Warteschlange vor einer Ampel. Kommt hier ein Unfallwagen mit Blaulicht und Sirene an, so machen die davor stehenden Autos Platz und lassen den Unfallwagen vorbei an die erste Stelle in der Warteschlange. Verwendet man die Klasse `priority_queue`, muss man folgende Headerdatei inkludieren:

```
#include <queue>
```

Der Syntax zum Anlegen eines `priority_queue`-Objekts ist z. B. Folgende:

```
priority_queue<T> name
```

Hiermit wird ein leeres `priority_queue`-Objekt mit den Namen *name* angelegt, das Elemente vom Typ *T* aufnehmen kann.

Konstruktoren und Methoden der Klasse priority_queue

Folgende Konstruktoren und Methoden bietet die Templateklasse `priority_queue` an, wobei die Methoden denen von der Klasse `stack` ähneln, sich aber anders verhalten:

`priority_queue()` $O(1)$
: Default-Konstruktor: legt leere Prioritäts-Warteschlange an, wobei über `operator<` die Priorität festgelegt wird. Je höher die Priorität eines Elements ist, um so weiter vorne wird es in der Warteschlange eingeordnet.

`priority_queue(const priority_queue&` $O(n)$`)`
: Kopierkonstruktor

`priority_queue(InputIterator anf, InputIterator end)` $O(n)$
: legt Prioritäts-Warteschlange an, die die Elemente aus dem Bereich von einschließlich `anf` bis ausschließlich `end` (wird nicht mehr kopiert) enthält.

`void push(const T& elem)` $O(1)$
: fügt `elem` in der Prioritäts-Warteschlange entsprechend seiner Priorität ein.

`void pop()` $O(1)$
: entfernt das höchstpriore Element an der ersten Stelle aus der Prioritäts-Warteschlange, liefert es aber nicht zurück.

`const T& top() const` $O(1)$
: liefert eine `const`-Referenz auf das an erster Stelle stehende Element der Prioritäts-Warteschlange, entfernt dieses aber nicht aus der Warteschlange.

`size_type size() const` $O(1)$
: liefert aktuelle Anzahl von Elementen in der Prioritäts-Warteschlange.

`bool empty() const` $O(1)$
: liefert `true`, wenn Prioritäts-Warteschlange leer ist und sonst `false`.

Anders als bei den Klassen `stack` und `queue` sind für Prioritäts-Warteschlangen keine Vergleichsoperatoren verfügbar, da ein Vergleichen von zwei Prioritäts-Warteschlangen wenig Sinn macht, und zudem aufgrund der internen Heapstruktur sehr aufwändig wäre.

Programm 4.33 – `prioqueue1.cpp`:
Einfaches Demoprogramm zur Klasse priority_queue

```
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <queue>
using namespace std;
int main(void) {
   srand(time(0));
   priority_queue<int> prioq;
   cout << "Eingefügt in folg. Reihenfolge: ";
   for (unsigned i = 0; i < 20; i++) {
      unsigned z = rand()%10;
      cout << z << " ";
      prioq.push(z);
   }
```

```
   cout << endl;
   cout << "Reihenfolge in priority_queue:  ";
   while (!prioq.empty()) {
      cout << prioq.top() << " ";
      prioq.pop();
   }
}
```

Programm 4.33 liefert die folgende Ausgabe:

```
Eingefügt in folg. Reihenfolge: 8 3 6 9 0 2 1 6 1 2 1 7 1 5 0 3 0 5 6 7
Reihenfolge in priority_queue:  9 8 7 7 6 6 6 5 5 3 3 2 2 1 1 1 0 0 0
```

Interne Realisierung einer Prioritäts-Warteschlange

Bei der Klasse `priority_queue` handelt es sich um eine so genannte *Container-Adaptorklasse*, die sich intern unter Verwendung der sequenziellen Containerklassen `vector` oder `deque` realisieren lässt. Die Voreinstellung ist, dass die Klasse `priority_queue` intern als `vector` realisiert wird:

```
priority_queue<T, vector<T> > pq; // entspr.: priority_queue<T> pq;
```

Möchte man eine andere `deque`-Realisierung, muss man folgende Syntax verwenden:

```
priority_queue<T, deque<T> > pq;
```

Programm 4.34 gibt die Zeilen der Datei `namen.txt` absteigend sortiert aus.

Programm 4.34 – `prioqueue2.cpp`:
Interne Realisierung einer Prioritäts-Warteschlange als vector

```
 1  #include <fstream>
 2  #include <queue>
 3  #include <vector>
 4  #include <deque>
 5  #include <string>
 6  #include <iostream>
 7  using namespace std;
 8
 9  int main(void)
10  {
11     ifstream eing("namen.txt");
12     priority_queue<string> pq; // entspricht:
13                                // priority_queue<string, vector<string> > pq;
14
15     //....Datei zeilenweise lesen und Zeilen in Prioritäts-Warteschl. speichern
16     string zeile;
17     while (getline(eing, zeile))
18        pq.push(zeile + "\n");
19
20     //....In Prioritäts-Warteschlange gespeicherten Zeilen ausgeben
21     while (!pq.empty()) {
22        cout << pq.top();
```

```
23        pq.pop();
24     }
25  }
```

Alternativ könnte man die Prioritäts-Warteschlange in Programm 4.34 auch intern als Deque realisieren lassen als, indem man die Zeile 12 wie folgt angibt:

```
12    priority_queue<string, deque<string> > pq;
```

Hat die Datei `namen.txt` den folgenden Inhalt:

```
Berta
Michael
Zorro
Aaron
Anton
```

so würde Programm 4.34 in beiden Fällen Folgendes ausgeben:

```
Zorro
Michael
Berta
Anton
Aaron
```

Wie wir zuvor gesehen haben, ist die Voreinstellung, dass Elemente in einer Prioritäts-Warteschlange abwärts sortiert werden. Um eine andere Sortierung zu erreichen, gibt es mehrere Möglichkeiten, die nachfolgend vorgestellt werden.

Überladen von operator< in eigener Wrapper-Klasse (für andere Sortierung)

Eine Möglichkeit, eine andere Sortierreihenfolge in einer Prioritäts-Warteschlange zu erreichen, ist eine eigene Wrapper-Klasse, in der man den Operator `operator<` entsprechend der gewünschten Reihenfolge überlädt, wie es in Programm 4.35 gezeigt ist.

Programm 4.35 – `prioqueue3.cpp`:
Aufsteigendes Sortieren mittels einer eigenen Wrapper-Klasse

```
#include <fstream>
#include <queue>
#include <string>
#include <iostream>
using namespace std;

//.....................................................Wrapper-Klasse myString
class myString {
  public:
     myString(const string s) : m_str(s) { }
     string getString() const                { return m_str;          }
     bool operator< (const myString& r) const { return m_str > r.m_str; }
  private:
     string m_str;
};
```

```
//.....................................................................main
int main(void) {
   ifstream eing("namen.txt");
   priority_queue<myString> pq;
     //....Datei zeilenweise lesen und Zeilen in Prioritäts-Warteschl. speichern
   string zeile;
   while (getline(eing, zeile))
      pq.push(zeile + "\n");
     //....In Prioritäts-Warteschlange gespeicherten Zeilen ausgeben
   while (!pq.empty()) {
      cout << pq.top().getString();
      pq.pop();
   }
}
```

Programm 4.35 gibt dann den Inhalt der Datei `namen.txt` aufsteigend sortiert aus:

```
Aaron
Anton
Berta
Michael
Zorro
```

Programm 4.36 zeigt, wie man unter Verwendung einer Wrapper-Klasse eine Prioritäts-Warteschlange auch nach mehreren Schlüsseln sortieren lassen kann.

Programm 4.36 – `prioqueue4.cpp`:
Sortieren einer Prioritäts-Warteschlange nach mehreren Schlüsseln

```
#include <string>
#include <queue>
#include <iostream>
using namespace std;

class CArtikel
{
     friend bool operator< (const CArtikel& l, const CArtikel& r) {
        return (l.m_regal > r.m_regal ||
                (l.m_regal == r.m_regal && l.m_nr > r.m_nr));
     }
     friend ostream& operator<<(ostream& os, const CArtikel& art) {
        return os << art.m_regal << art.m_nr << ": " << art.m_name;
     }
  public:
     CArtikel(char regal = 'A', int nr = 1, string name="")
          : m_regal(regal), m_nr(nr), m_name(name) { }
  private:
     char   m_regal;
     int    m_nr;
     string m_name;
};
```

```
int main(void) {
   priority_queue<CArtikel> lager;
   lager.push(CArtikel('X', 3, "Radiergummi"));
   lager.push(CArtikel('B', 2, "Schulhefte"));
   lager.push(CArtikel('M', 1, "Malstifte"));
   lager.push(CArtikel('B', 1, "Notizbloecke"));
   lager.push(CArtikel('X', 2, "Locher"));
   lager.push(CArtikel('M', 2, "Bleistifte"));
   lager.push(CArtikel('X', 1, "Tintenkiller"));
   while (!lager.empty()) {
      cout << lager.top() << endl;
      lager.pop();
   }
}
```

Programm 4.36 liefert die folgende Ausgabe:

```
B1: Notizbloecke
B2: Schulhefte
M1: Malstifte
M2: Bleistifte
X1: Tintenkiller
X2: Locher
X3: Radiergummi
```

Verwenden alternativer Konstruktoren (für andere Sortierung)

`priority_queue<T, vector<T>, name<T> >()`

`priority_queue<T, deque<T>, name<T> >()`

legen leere Prioritäts-Warteschlnge fest, in der die Sortierung entsprechend dem überladenen `operator()` in der Klasse `name` erfolgen soll.

`priority_queue<T, vector<T>, name<T> >`
`(InputIterator anf, InputIterator end)`

`priority_queue<T, deque<T>, name<T> >`
`(InputIterator anf, InputIterator end)`

legen Prioritäts-Warteschlange an, die Elemente aus dem Bereich von einschließlich `anf` bis ausschließlich `end` (wird nicht mehr kopiert) enthält. Die Sortierung erfolgt entsprechend dem überladenen `operator()` in der Struktur bzw. Klasse `name`.

Bei beiden Methoden wird der eigene überladene `operator()` in der Struktur bzw. Klasse `name` hier immer mit zwei zu vergleichenden Elemente aus der Prioritäts-Wartschlange aufgerufen. Über den Rückgabewert kann man hier steuern, wie diese beiden Elemente zueinander anzuordnen sind.

Programm 4.37 – `prioqueue5.cpp`:
Demoprogramm zu den priority_queue()-Konstruktoren mit eigener Sortierung

```
#include <cstdlib>
#include <ctime>
#include <functional>  // wegen der Standard-Funktion greater
#include <string>
```

```
#include <queue>
#include <iostream>
using namespace std;
template <typename T>
void ausgabe(T& pq, string text) {
   cout << text;
   while (!pq.empty()) {
      cout << pq.top() << " ";
      pq.pop();
   }
   cout << endl;
}
template <typename T>
class aufwaerts {
  public:
    bool operator() (T x, T y) const { return x > y; }
};
int main(void) {
   srand(time(0));
   priority_queue<int> pq1;
   priority_queue<int, vector<int>, greater<int> >  pq2;
   priority_queue<int, deque<int>, aufwaerts<int> > pq3;
   cout << "Einfüge-Reihenfolge: ";
   for (unsigned i = 0; i < 20; i++) {
      unsigned z = rand()%10;
      cout << z << " ";   pq1.push(z);   pq2.push(z);   pq3.push(z);
   }
   cout << endl;
   ausgabe(pq1, "pq1:  ");
   ausgabe(pq2, "pq2:  ");
   ausgabe(pq3, "pq3:  ");
   cout << "-----------------------------------------------" << endl;
   int x[] = { 2, 5, 9, 1, 3, 8 };
   priority_queue<int> pq4(x, x+6);
   priority_queue<int, deque<int>, greater<int> >    pq5(x, x+6);
   priority_queue<int, vector<int>, aufwaerts<int> > pq6(x, x+6);
   ausgabe(pq4, "pq4:  ");
   ausgabe(pq5, "pq5:  ");
   ausgabe(pq6, "pq6:  ");
}
```

Programm 4.37 liefert z. B. die folgende Ausgabe:

```
Einfüge-Reihenfolge: 4 5 1 9 2 9 6 0 1 2 3 0 4 6 4 6 6 8 0 5
pq1:  9 9 8 6 6 6 6 5 5 4 4 4 3 2 2 1 1 0 0 0
pq2:  0 0 0 1 1 2 2 3 4 4 4 5 5 6 6 6 6 8 9 9
pq3:  0 0 0 1 1 2 2 3 4 4 4 5 5 6 6 6 6 8 9 9
-----------------------------------------------
pq4:  9 8 5 3 2 1
pq5:  1 2 3 5 8 9
pq6:  1 2 3 5 8 9
```

Zugriff auf eine Prioritäts-Warteschlange über einen Vektor

In einer Prioritäts-Warteschlange kann nur auf das erste Element, aber nicht auf ein beliebiges Element zugegriffen werden. Jedoch ist es möglich, mittels eines Vektors das Verhalten einer Prioritäts-Warteschlange nachzubilden, so das man dann über den Vektor auch auf beliebige Elemente in der simulierten Prioritäts-Warteschlange zugreifen kann. Dazu muss man wissen, dass Prioritäts-Warteschlangen intern als so genannte *Heaps* realisiert sind. Ein Heap ist eine spezielle Organisation von Daten innerhalb eines durch zwei Iteratoren definierten Bereichs, wobei die Anordnung der Elemente über eine Vergleichsfunktion `verglFkt` festgelegt wird. Für Heaps stehen die folgenden Heap-Algorithmen zur Verfügung:

`void make_heap(RandomAccessIterator anf, RandomAccessIterator ende, verglFkt)`
: legt durch Umordnen der Elemente einen Heap zu dem Bereich von einschließlich `anf` bis ausschließlich `ende` an.

`void push_heap(RandomAccessIterator anf, RandomAccessIterator ende, verglFkt)`
: fügt das Element `*(ende-1)` zu dem Heap hinzu, der durch den Bereich von einschließlich `anf` bis ausschließlich `ende` festgelegt wird. Anders ausgedrückt: Es plaziert das letzte Elemente an seine geeignete Position im Heap.

`void pop_heap(RandomAccessIterator anf, RandomAccessIterator ende, verglFkt)`
: plaziert das größte Element `*anf` an die Position `*(ende-1)` und organisiert den restlichen Bereich so um, dass er sich wieder in der entsprechenden Heap-Ordnung befindet.

`void sort_heap(RandomAccessIterator anf, RandomAccessIterator ende, verglFkt)`
: sortiert den Bereich von einschließlich `anf` bis ausschließlich `ende` in einer normalen Reihenfolge, so dass es sich hierbei nicht mehr um einen Heap handelt. Danach machen Aufrufe wie `push_heap()` oder `pop_heap()` auf diesen Bereich keinen Sinn mehr.

Eine Prioritäts-Warteschlange ist prinzipiell nichts anderes als eine Wrapper-Klasse um einen Heap.

Programm 4.38 ist ein erster fehlerhafter Versuch, mittels eines Vektors eine Prioritäts-Warteschlange nachzubilden. In Programm 4.38 wird in der Klasse `PV` der Name `c` benutzt, der nach Standard-C++ das Container-Objekt ist, das von der Prioritäts-Warteschlange verwendet wird.

Programm 4.38 – `prioqueue6.cpp`:
Fehlerhafter Versuch, mittels eines Vektors eine Prioritäts-Warteschlange nachzubilden

```
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <queue>
using namespace std;
```

```
class PV : public priority_queue<int> {
  public:
        // c = Container-Objekt, das Prioritäts-Warteschl. realisiert
     vector<int>& getContainer() { return c; }
};
int main(void) {
  unsigned i;
   PV pv;
   for (i = 0; i < 20; i++)
      pv.push(rand() % 10);
   vector<int> v = pv.getContainer();
   cout << "     Vektorausgabe: ";
   for (i = 0; i < v.size(); i++)
       cout << v[i] << " ";
   cout << endl;
   cout << "Prio-Warteschlange: ";
   while (!pv.empty()) {
      cout << pv.top() << " ";
      pv.pop();
   }
}
```

Programm 4.38 liefert z. B. die folgende Ausgabe:

```
     Vektorausgabe: 9 9 8 7 6 5 8 4 6 2 4 0 5 1 0 1 1 2 4 0
Prio-Warteschlange: 9 9 8 8 7 6 6 5 5 4 4 4 2 2 1 1 1 0 0 0
```

An dieser Ausgabe ist erkennbar, dass der Vektor die Elemente nicht in der Reihenfolge wie in der Prioritäts-Warteschlange enthält, da diese ja nicht in Form eines Heaps für den Vektor angeordnet wurden.
Programm 4.39 behebt dieses Manko, indem die Klasse PV sich einen Vektor als Member hält und bei jedem `push()` und `pop()` sich den Heap für diesen Member-Vektor neu organisieren lässt. Die Methode `getVector()` liefert dabei die Daten des Heaps in Form eines sortierten Vektors, wobei sie zuvor die Daten des Heaps in einen lokalen Vektor überträgt, den sie dann mittels `sort_heap()` normal sortieren lässt.

Programm 4.39 – `prioqueue7.cpp`:
1. Möglichkeit zur Nachbildung einer Prioritäts-Warteschlange mittels eines Vektors

```
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iterator>
#include <queue>
#include <vector>
#include <iostream>
using namespace std;
//.....................................................Templateklasse PV
template<class T, class verglFkt>
class PV {
  public:
```

```
     PV(verglFkt vf = verglFkt()) :  vergl(vf) {}
     void push(const T& elem) {
        v.push_back(elem); // am Ende einfügen
        push_heap(v.begin(), v.end(), vergl); // Heap neu organisieren
     }
     void pop() {
        pop_heap(v.begin(), v.end(), vergl); // 1. Elem. ans Ende legen
        v.pop_back(); // Letztes Element aus Heap entfernen
     }
     const T& top()     { return v.front(); }
     bool empty() const { return v.empty(); }
     int size() const   { return v.size();  }
     vector<T> getVector() {
        vector<T> r(v.begin(), v.end());
        sort_heap(r.begin(), r.end(), vergl);
        reverse(r.begin(), r.end()); // Umgekehrte Reihenfolge in Prio.-WS
        return r;
     }
  private:
     vector<T> v;
     verglFkt  vergl;
};
//.....................................................................main
int main(void) {
   unsigned i;
   srand(time(0));

   PV<int, less<int> >    pq1;
   PV<int, greater<int> > pq2;

   cout << "Eintrag-Reihenfolge: ";
   for (i = 0; i < 20; i++) {
      unsigned z = rand()%10;
      pq1.push(z);
      pq2.push(z);
      cout << z << " ";
   }
   cout << endl << "-------------------------------------------------" << endl;
   const vector<int>& v1 = pq1.getVector();
   cout << "     Vektorausgabe1: ";
   for (i = 0; i < v1.size(); i++)
      cout << v1[i] << " ";
   cout << endl << "Prio-Warteschlange1: ";
   while (!pq1.empty()) {
      cout << pq1.top() << " ";
      pq1.pop();
   }
   cout << endl << "-------------------------------------------------" << endl;
   const vector<int>& v2 = pq2.getVector();
   cout << "     Vektorausgabe2: ";
```

```
   for (i = 0; i < v2.size(); i++)
      cout << v2[i] << " ";
   cout << endl << "Prio-Warteschlange2: ";
   while (!pq2.empty()) {
      cout << pq2.top() << " ";
      pq2.pop();
   }
}
```

Programm 4.39 liefert z. B. die folgende Ausgabe:

```
Eintrag-Reihenfolge: 7 2 9 6 3 2 1 0 6 2 1 3 9 7 8 5 0 7 5 5
-------------------------------------------------
     Vektorausgabe1: 9 9 8 7 7 7 6 6 5 5 5 3 3 2 2 2 1 1 0 0
Prio-Warteschlange1: 9 9 8 7 7 7 6 6 5 5 5 3 3 2 2 2 1 1 0 0
-------------------------------------------------
     Vektorausgabe2: 0 0 1 1 2 2 2 3 3 5 5 5 6 6 7 7 7 8 9 9
Prio-Warteschlange2: 0 0 1 1 2 2 2 3 3 5 5 5 6 6 7 7 7 8 9 9
```

Leitet man in Programm 4.39 die Templateklasse `PV` von der Klasse `priority_queue` ab, kann man diese Klasse kürzer implementieren, wie es nachfolgend gezeigt ist:

```
//.... siehe Programm 4.39
//.....................................................Templateklasse PV
template<typename T, typename verglFkt>
class PV : public priority_queue<T, vector<T>, verglFkt> {
  public:
     vector<T> getVector() {
        vector<T> r(this->c.begin(),this->c.end());
        sort_heap(r.begin(), r.end(), this->comp); // comp liefert Vergleichsfkt.
        reverse(r.begin(), r.end()); // Umgekehrte Reihenfolge in Prio.-WS
        return r;
     }
};
//.... siehe Programm 4.39
```

4.3.10 Die assoziativen Container-Klassen set und multiset

- Ein *Set* ist eine Menge von Objekten, bei der jedes Element *nur einmal* vorkommen darf. Da die Elemente zu jedem Zeitpunkt sortiert vorliegen, kann sehr schnell festgestellt werden, ob ein Wert im *Set* vorhanden ist oder nicht.
- Ein *Multiset* entspricht einem *Set*, jedoch mit dem Unterschied, dass in einem *Multiset* mehrere Elemente den gleichen Wert besitzen dürfen.

Verwendet man *Sets* oder *Multisets*, muss man folgende Headerdatei inkludieren:

```
#include <set>
```

Der Syntax zum Anlegen eines `set`- bzw. `multiset`-Objekts ist z. B. Folgende:

```
  set<T> name
  multiset<T> name
```

Hiermit wird ein `set`- bzw. `multiset`-Objekt mit den Namen *name* angelegt.

Konstruktoren der Klassen set und multiset

Konstruktoren der Templateklassen `set<T>` und `multiset<T>` sind:

`set()` $O(1)$
`multiset()` $O(1)$
: Default-Konstruktoren (legt leeren Set bzw. Multiset an).

`set(InputIterator anfang, InputIterator ende)` $O(n)$
`multiset(InputIterator anfang, InputIterator ende)` $O(n)$
: legen einen Set bzw. Multiset an, der mit dem Bereich von einschließlich `anfang` bis ausschließlich `ende` (wird nicht mehr kopiert) initialisiert wird.

`set(const set&)` $O(n)$
`multiset(const multiset&)` $O(n)$
: Kopierkonstruktoren

Programm 4.40 – `set1.cpp`:
Demoprogramm zu den set/multiset-Konstruktoren

```
#include <cstdlib>
#include <ctime>
#include <string>
#include <iterator>
#include <set>
#include <iostream>
using namespace std;

int main(void)
{
   unsigned i;
   char z;

   srand(time(0));
   set<char> menge1;            set<char>::iterator it;
   multiset<char> multiMenge1;  multiset<char>::iterator itMulti;
   for (i=0; i < 50; i++) {
      cout << (z = 'a'+rand()%26);
      menge1.insert(z);
      multiMenge1.insert(z);
   }
   cout << endl;
   for (it = menge1.begin(); it != menge1.end(); it++)
      cout << *it;
   cout << endl;
   for (itMulti = multiMenge1.begin(); itMulti != multiMenge1.end(); itMulti++)
      cout << *itMulti;
   cout << endl << "--------------------------------------------------" << endl;
   string gruss = "Hallo wie gehts denn so?";
   cout << gruss << endl;
   set<char> menge2(gruss.begin(), gruss.end());
   multiset<char> multiMenge2(gruss.begin(), gruss.end());
```

```
   for (it = menge2.begin(); it != menge2.end(); it++)
      cout << *it;
   cout << endl;
   for (itMulti = multiMenge2.begin(); itMulti != multiMenge2.end(); itMulti++)
      cout << *itMulti;
}
```

Programm 4.40 liefert z. B. die folgende Ausgabe:

```
fkuwcrnfpwkzffzorxodqhkehsmbrfwyqsxujmaaikcprbekau
abcdefhijkmnopqrsuwxyz
aaabbccdeeffffhhijkkkkkmmnooppqqrrrrssuuuwwwxxyzz
--------------------------------------------------
Hallo wie gehts denn so?
 ?Hadeghilnostw
    ?Hadeeeghillnnoosstw
```

Konstruktoren mit Festlegung der Sortierung

Voreinstellung ist, dass die Elemente in Sets bzw. Multisets aufwärts sortiert werden. Mit den hier vorgestellten Konstruktoren kann man eine andere Sortierung erreichen, wobei die Sortierung entsprechend dem überladenen `operator()` in der Struktur bzw. Klasse `name` erfolgt.

`set<T, name>()` $O(1)$

`multiset<T, name>()` $O(1)$

Default-Konstruktoren (legen leeren Set bzw. Multiset an).

`set<T, name>(InputIterator anfang, InputIterator ende)` $O(n \log n)$

`multiset<T,name>(InputIterator anfang, InputIterator ende)` $O(n \log n)$

legen einen Set bzw. Multiset an, der mit dem Bereich von einschließlich `anfang` bis ausschließlich `ende` (wird nicht mehr kopiert) initialisiert wird.

Der eigene überladene `operator()` in der Struktur bzw. Klasse `name` wird hier immer mit zwei zu vergleichenden Elemente aus der Set bzw. Multiset aufgerufen. Über den Rückgabewert kann man hier steuern, wie diese beiden Elemente zueinander anzuordnen sind.
Die STL bietet eine ganze Reihe von Algorithmen, die für sortierte Bereiche, welche ja in Sets vorliegen, ausgelegt sind. In Programm 4.41 werden folgende STL-Algorithmen verwendet:

Funktionsname	erzeugt einen Set mit den Elementen …
`set_union(...)`	(alle) aus beiden Sets (Vereinigung)
`set_intersection(...)`	die nur in beiden Sets vorkommen (Durchschnitt)
`set_difference(...)`	die nur im ersten Set vorkommen (Differenz)
`set_symmetric_difference(...)`	die nur in einer der beiden Sets vorkommen

Programm 4.41 – `set2.cpp`:
Demoprogramm zu den set-Konstruktoren mit eigener Sortierung

```
#include <string>
#include <iterator>
#include <set>
#include <iostream>
using namespace std;
```

```
template <typename T>
void ausgabe(T l, const char *str1, const char *str2)
{
   cout << str1;
   for (typename T::iterator it = l.begin(); it != l.end(); it++)
      cout << *it << " ";
   cout << str2 << endl;
}

struct abwaerts
{
  public:
    bool operator() (string str1, string str2) const { return str1 > str2; }
};

int main(void)
{
   const unsigned N = 5;
   string name1[N] = {"hans", "fritz", "emil", "toni", "gerta"  };
   string name2[N] = {"hans", "anni",  "emil", "toni", "gunter" };

   set<string> A(name1, name1 + N);
   set<string> B(name2, name2 + N);
   set<string> C;
   ausgabe(A, "Menge A: ", "");   ausgabe(B, "Menge B: ", "");
   cout << "-----------------------------------------------------------" << endl;
   set_union(A.begin(), A.end(), B.begin(), B.end(), inserter(C, C.begin()));
   ausgabe(C, " A + B: ", " (Vereinigung)");
   C.clear();
   set_intersection(A.begin(), A.end(), B.begin(),B.end(), inserter(C,C.begin()));
   ausgabe(C, " A / B: ", " (Durchschnitt)");
   C.clear();
   set_difference(A.begin(), A.end(), B.begin(), B.end(), inserter(C, C.begin()));
   ausgabe(C, " A - B: ", " (Differenz)");
   C.clear();
   set_difference(B.begin(), B.end(), A.begin(), A.end(), inserter(C, C.begin()));
   ausgabe(C, " B - A: ", " (Differenz)");
   C.clear();
   set_symmetric_difference(A.begin(), A.end(), B.begin(), B.end(),
                            inserter(C, C.begin()));
   ausgabe(C, "A -- B: ", " (Symetr. Differenz)");
   cout << "-----------------------------------------------------------" << endl;
   set<string, abwaerts> D;
   set_union(A.begin(), A.end(), B.begin(), B.end(), inserter(D, D.begin()));
   ausgabe(D,  "    A + B: ", " (abwaerts sortiert)");
   set<string, abwaerts> E(A.begin(), A.end());
   for (unsigned i=0; i < N; i++)
       E.insert(name2[i]);
   ausgabe(E,  "A + name2: ", " (abwaerts sortiert)");
}
```

Programm 4.41 liefert die folgende Ausgabe:

```
Menge A: emil fritz gerta hans toni
Menge B: anni emil gunter hans toni
-----------------------------------------------------------
 A + B: anni emil fritz gerta gunter hans toni  (Vereinigung)
 A / B: emil hans toni  (Durchschnitt)
 A - B: fritz gerta  (Differenz)
 B - A: anni gunter  (Differenz)
A -- B: anni fritz gerta gunter  (Symetr. Differenz)
-----------------------------------------------------------
    A + B: toni hans gunter gerta fritz emil anni  (abwaerts sortiert)
A + name2: toni hans gunter gerta fritz emil anni  (abwaerts sortiert)
```

Abwärts Sortieren mit dem Funktionsobjekt greater

Die Voreinstellung bei Sets ist, dass sie mittels Aufruf des vordefinierten Funktionsobjekts `less` aufsteigend sortiert werden. Um Sets absteigend sortieren zu lassen kann man auch das vordefinierte Funktionsobjekt `greater` angeben, wie es in Programm 4.42 gezeigt ist.

Programm 4.42 – `set3.cpp`:
Absteigend Sortieren mit Funktionsobjekt greater

```
#include <string>
#include <iterator>
#include <set>
#include <iostream>
using namespace std;
template<typename T1, typename T2>
ostream& operator<< (ostream& os, set<T1, T2> s) {
   typename set<T1, T2>::iterator it;
   for (it = s.begin(); it != s.end(); it++)
      os << *it << " ";
   return os;
}
int main(void) {
  string name[] = { "Hans", "Fritz", "Anton", "Zorro", "Gerta" };
   set<string>                  auf1(name, name + sizeof(name)/sizeof(*name));
   set<string, less<string> >   auf2(name, name + sizeof(name)/sizeof(*name));
   set<string, greater<string> > ab(name, name + sizeof(name)/sizeof(*name));
   cout << "aufwaerts: " << auf1 << endl;
   cout << "aufwaerts: " << auf2 << endl;
   cout << "abwaerts:  " << ab  << endl;
}
```

Programm 4.42 liefert die folgende Ausgabe:

```
aufwaerts: Anton Fritz Gerta Hans Zorro
aufwaerts: Anton Fritz Gerta Hans Zorro
abwaerts:  Zorro Hans Gerta Fritz Anton
```

Methoden zum Einfügen von Elementen in Sets bzw. Multisets

Während bei Sets folgende Methodenaufrufe keine Auswirkung haben, wenn ein entsprechendes Element bereits im Set vorhanden ist, fügen sie in jedem Fall die entsprechenden Elemente in einer Multiset ein:

`pair<iterator, bool> insert(const T& elem)` *(bei set)* $O(n \log n)$
`iterator insert(const T& elem)` *(bei multiset)* $O(n \log n)$

fügt das Element `elem` ein. Bei einem Set wird ein Paar zurückgegeben, in dem der boolesche Wert anzeigt, ob das Element eingefügt werden konnte, oder nicht (weil es schon vorhanden war). Bei einem erfolgreichen Einfügen zeigt der Iterator des zurückgegebenen Paares auf das neue eingefügte Element. Sollte kein Einfügen möglich gewesen sein, dann zeigt der Iterator auf das bereits vorhandene Element des Sets. Bei einem Multiset wird lediglich der Iterator auf das neu eingefügte Element zurückgegeben.

`iterator insert(iterator pos, const T& elem)` $O(n \log n)$

fügt das Element `elem` ein und liefert einen Iterator auf dieses eingefügte Element. Da durch das Einfügen die Sortierung erhalten bleiben muss, ist die Position `pos` lediglich ein Vorschlag, der eventuell ein schnelleres Einfügen erlaubt.

`void insert(InputIterator anf, InputIterator end)` $O(n \log n)$

fügt alle Elemente ein, die sich in dem Bereich von einschließlich `anf` bis ausschließlich `end` (wird nicht mehr eingefügt) befinden.

Programm 4.43 – `set4.cpp`:
Demoprogramm zu insert() bei Sets

```
#include <cstdlib>
#include <ctime>
#include <iterator>
#include <set>
#include <iostream>
using namespace std;

int main(void)
{
   set<int> lotto; // int-Set
   set<int>::iterator it;
   pair<set <int>::iterator, bool> p;
   for (unsigned i=1; i<=6; i++)
      do {
         p = lotto.insert( rand()%49+1 );
      } while ( !p.second ); // doppelte Zahl unterbinden
   cout << "Die Lottozahlen sind: ";
   for (it = lotto.begin(); it != lotto.end(); it++)
      cout << *it << ", ";
}
```

Programm 4.43 liefert z. B. die folgende Ausgabe:

```
Die Lottozahlen sind: 5, 12, 26, 30, 43, 48,
```

Methoden zum Löschen von Elementen in Sets bzw. Multisets

`void clear()` $O(n)$
: löscht alle Elemente der Set. Die Set hat danach die Größe 0.

`void erase(iterator pos)` $O(1)$
: löscht das Element an Iterator-Position `pos`.

`void erase(iterator anfang, iterator ende)` $O(log\ n)$
: löscht alle Elemente, die sich in dem Bereich von einschließlich `anfang` bis ausschließlich `ende` (wird nicht mehr gelöscht) befinden.

`size_type erase(const T& elem)` $O(log\ n)$
: löscht alle Elemente des Sets, die identisch zu `elem` sind, und liefert die Anzahl der gelöschten Elemente als Rückgabewert.

Programm 4.44 – `set5.cpp`:
Wortstatistik zu einer Datei mittels erase() bei Multisets

```
#include <cctype>
#include <fstream>
#include <iterator>
#include <string>
#include <set>
#include <iomanip>
#include <iostream>
using namespace std;
int main(void) {
  ifstream datei("set5.cpp"); // Input-Stream auf Datei
  istreambuf_iterator<char> it(datei), end; // auf Dateianfang und für Ende
   multiset<string> worte; // string-Multiset
   string str;
   for ( ; it != end; it++) {
      if (isalpha(*it))
         str += tolower(*it);
      else {
         if (isalpha(str[0]))
            worte.insert(str);
         str.clear();
      }
   }
   multiset<string>::iterator msetIt = worte.begin();
   while ( !worte.empty() ) {
      msetIt = worte.begin();
      string s = *msetIt;
      cout << setw(20) << setiosflags(ios::left) << s << ":"
           << setw(10) << setiosflags(ios::right) << worte.erase(s)
                       << resetiosflags(ios::right) << endl;
   }
}
```

Programm 4.44 liefert die folgende Ausgabe:

```
auf                 :         2
begin               :         2
cctype              :         1
char                :         1
clear               :         1
cout                :         1
cpp                 :         1
datei               :         3
.............................
.............................
using               :         1
void                :         1
while               :         1
worte               :         6
```

Methoden zum Suchen von Elementen in Sets bzw. Multisets

Alle nachfolgend vorgestellten Methoden liefern einen Iterator auf das Ende (`end()`) des Sets bzw. Multisets zurück, wenn kein entsprechendes Element gefunden wird:

`iterator find(const T& elem)` $O(\log n)$

`const_iterator find(const T& elem) const` $O(\log n)$

sucht im Set bzw. Multiset nach dem ersten Element, das identisch zu `elem` ist, und liefert einen Iterator auf dieses gefundene Element zurück.

`iterator lower_bound(const T& elem)` $O(\log n)$

`const_iterator lower_bound(const T& elem) const` $O(\log n)$

sucht im Set bzw. Multiset nach dem ersten Element, dessen Wert nicht kleiner als der von `elem` ist, und liefert einen Iterator auf dieses gefundene Element zurück.

`iterator upper_bound(const T& elem)` $O(\log n)$

`const_iterator upper_bound(const T& elem) const` $O(\log n)$

sucht im Set bzw. Multiset nach dem ersten Element, dessen Wert größer als der von `elem` ist, und liefert einen Iterator auf dieses gefundene Element zurück.

`pair<iterator, iterator> equal_range(const T& elem)` $O(\log n)$

`pair<const_iterator, const_iterator>`
`equal_range(const T& elem) const` $O(\log n)$

liefert ein Paar von Iteratoren zurück, wobei der erste Iterator dem Iterator entspricht, den auch `lower_bound()` liefern würde, und der zweite dem Iterator, den `upper_bound()` zurückgeben würde.

Das Ergebnis von `equal_range()` entspricht:
`make_pair(lower_bound(elem), upper_bound(elem))`.

Programm 4.45 – `set6.cpp`:
Demoprogramm zu find(), lower_bound(), upper_bound() und equal_range()

```
#include <cstdlib>
#include <ctime>
#include <iterator>
```

```
#include <utility>
#include <set>
#include <string>
#include <iostream>
using namespace std;

int main(void) {
  unsigned z = 0;
  srand(time(0));
  multiset<int> zahlen;
  multiset<int>::iterator it, it1, it2;
   for (unsigned i=1; i<=20; i++) // mit Zufallszahlen füllen
      zahlen.insert( rand()%10);
   for (it = zahlen.begin(); it != zahlen.end(); it++)
       cout << *it;
   cout << endl;
   it1 = zahlen.lower_bound(7); // Anfang und Ende für 7-Folge suchen mit
   it2 = zahlen.upper_bound(7); // lower_bound() und upper_bound()
   for (it = zahlen.begin(); it != it2; it++)
      cout << ( (*it >= *it1) ? "-" : " " );
   cout << endl;
   pair<multiset<int>::iterator, multiset<int>::iterator> p; // Anfang/Ende suchen
   p = zahlen.equal_range(7);                                // mit equal_range()
   for (it = zahlen.begin(); it != zahlen.end(); it++)
      cout << ( (*it >= *(p.first) && (*it < *(p.second) ) ) ? "x" : " " );
   cout << endl;
   it = zahlen.find(7); // Zählen, wie oft die Zahl 7 vorkommt
   while (it != zahlen.end() && *it == 7)
      z++;
      it++;
   }
   cout << "7 kommt " << z << " mal vor" << endl;
}
```

Programm 4.45 liefert z. B. die folgende Ausgabe:

```
00223334445556777899
              ---
              xxx
7 kommt 3 mal vor
```

Anzahl von gleichen Elementen in set- bzw. multiset-Objekten

`size_type count(const T& elem) const` $O(\log n)$

gibt zurück, wie oft das Element `elem` im Set bzw. Multiset vorhanden ist. In Programm 4.45 hätte man sich z. B. das explizite Zählen des Vorkommens der Zahl 7 ersparen und statt dessen die folgende Anweisung angeben können:

```
cout << "7 kommt " << zahlen.count(7) << " mal vor" << endl;
```

Bei Sets liefert diese Methode den Wert 0 oder 1.

Größe von set- bzw. multiset-Objekten

Methoden zum Erfragen der Größe von `set`- bzw. `multiset`-Objekten sind:

`size_type size() const` $O(1)$
: liefert aktuelle Anzahl von Elementen im Set bzw. Multiset.

`size_type max_size() const` $O(1)$
: liefert (systemabhängige) maximal mögliche Größe für die jeweilige Set bzw. Multiset.

`bool empty() const` $O(1)$
: liefert `true`, wenn Set bzw. Multiset keine Elemente enthält und sonst `false`.

Vergleichen von set- bzw. multiset-Objekten

Folgende überladenen globalen Vergleichsoperatoren stehen für `set`- bzw. `multiset`-Objekten zur Verfügung:

`operator==(), operator!=()` $O(n)$
: Zwei Sets bzw. Multisets sind gleich, wenn sie beide den gleichen Elementtyp besitzen, die gleiche Größe haben und die Werte jedes Elementpaares an der gleichen Stelle den gleichen Wert besitzen. Andernfalls sind die beiden ungleich.

`operator<(), operator<=(), operator>(), operator>=()` $O(n)$
: Der Vergleich von zwei Sets bzw. Multisets liefert `true`, wenn beim paarweisen Vergleich im ersten Paar, bei dem sich die beiden Werte unterscheiden, der Wert in der ersten Set bzw. Multiset kleiner als der in der zweiten ist, andernfalls liefert dieser Vergleich `false`. Eine mögliche Realisierung dieser Operatoren wäre z. B.:

```
template<typename It1, typename It2>
bool setKleiner(It1 a1, It1 end1, It2 a2, It2 end2) {
   for ( ; a1 != end1 && a2 != end2 ; ++a1, ++a2) {
      if (*a1 < *a2) return true;
      if (*a2 < *a1) return false;
   }
   return a1 == end1 && a2 != end2;
}

template<typename T>
bool operator< (const set<T>& v1, const set<T>& v2) {
  return setKleiner(v1.begin(), v1.end(), v2.begin(), v2.end());
}

template<typename T>
bool operator<=(const set<T>& v1, const set<T>& v2) { return !(v2<v1); }

template<typename T>
bool operator> (const set<T>& v1, const set<T>& v2) { return v2 < v1;  }

template<typename T>
bool operator>=(const set<T>& v1, const set<T>& v2) { return !(v1<v2); }
```

Die Multiset-Realisierung ist dann entsprechend.

Vertauschen von set- bzw. multiset-Objekten

Zum Vertauschen der Inhalte von zwei `set`- bzw. `multiset`-Objekten steht die folgende Methode zur Verfügung:

`void swap(set &s)` $O(1)$

`void swap(multiset &m)` $O(1)$

vertauscht die Elemente der eigenen Set bzw. Multiset mit den Elementen von Set `s` bzw. Multiset `m`.

Iterator-Methoden der Klassen set und multiset

`iterator begin()` $O(1)$

`const_iterator begin() const` $O(1)$

liefert einen Iterator auf das erste Element der Set bzw. Multiset.

`iterator end()` $O(1)$

`const_iterator end() const` $O(1)$

liefert Iterator auf die Position hinter dem letzten Element der Set bzw. Multiset.

`reverse_iterator rbegin()` $O(1)$

`const_reverse_iterator rbegin() const` $O(1)$

liefert einen *reverse*-Iterator auf das letzte Element der Set bzw. Multiset.

`reverse_iterator rend()` $O(1)$

`const_reverse_iterator rend() const` $O(1)$

liefert einen *reverse*-Iterator auf die Position vor dem ersten Element der Set bzw. Multiset.

Der Datentyp value_type

Jeder Set bzw. Multiset definiert einen Typ `value_type`, der benutzt werden kann, um Elemente zu erzeugen, die in einer Set bzw. Multiset eingefügt werden können. Um z. B. ein Element `name` zu erzeugen, das zu einem `set<string>`-Objekt hinzugefügt werden kann, könnte man Folgendes angeben:

```
  set<string>::value_type name("Fritz");
```

Verwendet man solche `value_type`-Objekte, ist es ratsam, an diese einen eigenen Typnamen zu vergeben, wie z. B.:

```
typedef set<string>::value_type   stringSetTyp;
//.... Nun ist folgende Definition möglich:
   stringSetTyp name("Fritz");
```

Es ist auch möglich, anonyme `value_type`-Objekte zu erzeugen, um diese z. B. an eine Funktion zu übergeben, wie z. B.:

```
    fkt(set<string>::value_type("Fritz"));
//... bzw. bei eigenen Typnamen auch:
    fkt(stringSetTyp("Fritz"));
```

Programm 4.46 – `set7.cpp`:
Demoprogramm zu value_type

```
#include <string>
#include <iterator>
#include <set>
#include <iostream>
using namespace std;

int main(void) {
   typedef set<string>::value_type   stringSetTyp; // Typdef. für string-Elem.
   typedef multiset<int>::value_type intSetTyp;    // Typdef. für int-Elem.

    //.... Anlegen von Elementen, die in einem Set eingefügt werden können
   set<string>::value_type string1("Eins");
   stringSetTyp string2("Zwei");
   set<string>::value_type string3("Drei");

    //.... Einfügen der zuvor erzeugten Elemente
   set<string> strings;
   strings.insert(string1);
   strings.insert(string2);
   strings.insert(string3);

    //.... Ausgeben aller Elemente des Sets
   set<string>::iterator it;
   for (it = strings.begin(); it != strings.end(); it++)
      cout << *it << ", ";
   cout << endl;

    //.... Übergeben von anonymen value_type-Objekten
   cout << set<string>::value_type("anonym1, ");
   cout << stringSetTyp("anonym2");
   cout << endl;

    //.... Anlegen von Elementen, die in einem Multiset eingefügt werden können
   multiset<int>::value_type z[] = { 1, 2, 3, 1, 2, 4, 5, 3, 2, 1 };

    //.... Anlegen einer Multiset mit zuvor erzeugten Elementen
   multiset<int> zahlen(z, z + sizeof(z)/sizeof(*z));

    //.... Hinzufügen weiterer Elemente zur Multiset
   intSetTyp nochEineZahl(7); // neues value_type
   zahlen.insert(nochEineZahl);
   zahlen.insert(intSetTyp(6)); // anonymes value_type

    //.... Ausgeben aller Elemente des Multisets
   multiset<int>::iterator it2;
   for (it2 = zahlen.begin(); it2 != zahlen.end(); it2++)
      cout << *it2 << ", ";
}
```

Programm 4.46 liefert die folgende Ausgabe:

```
Drei, Eins, Zwei,
anonym1, anonym2
1, 1, 1, 2, 2, 2, 3, 3, 4, 5, 6, 7,
```

Modifizieren von Elementen in set bzw. multiset

Sollen in einem `set` oder `multiset` neben einem konstanten Element, das als Schlüssel für die Sortierung verwendet wird, weitere Datenelemente verwaltet werden, wobei diese jedoch anders als der Schlüssel änderbar sein sollen, so muss man Folgendes beachten:

- Um eine Änderung der Reihenfolge der Elemente zu unterbinden, sind auf Sets bzw. Multisets nur Operationen zugelassen, die keine Änderungen an den jeweiligen Schlüsseln zulassen. Dies wird meist dadurch realisiert, dass nur Iteratoren von den Typen `const_iterator` bzw. `const_reverse_iterator` für Sets bzw. Multisets angeboten werden.
- Diese konstanten Iteratoren ermöglichen damit auch keine Änderungen der anderen Elemente, die nicht als Schlüssel dienen.

Möchte man nun Elemente in Sets bzw. Multisets ändern, hat man zwei Möglichkeiten:

1. Verwendung von `const_cast` oder
2. entsprechendes Element zunächst entfernen und anschließend wieder mit dem neuen Wert im `set` bzw. `multiset` einfügen.

Programm 4.47 demonstriert beide Möglichkeiten.

Programm 4.47 – `set8.cpp`:
Modifizieren von Elementen in set bzw. multiset

```
#include <cstdlib>
#include <ctime>
#include <string>
#include <set>
#include <iomanip>
#include <iostream>
using namespace std;
class Spieler {
  public:
    Spieler(string nam) : name(nam), punkte(0) { }
    void addPunkte(int pkte) { punkte += pkte; }
    string getName()   const { return name;    }
    int    getPunkte() const { return punkte;  }
  private:
    const string name;
    int          punkte;
};
struct Vergleich {
 public:
   bool operator()(const Spieler& x, const Spieler& y) const {
      return x.getName() < y.getName();
   }
};
```

```
int main()
{
   int    i, n;
   string name[] = { "Hans", "Zorro", "Anna", "Michel" };
   const int anzahl = sizeof(name)/sizeof(*name);

   srand(time(0));

   //............................... 1. Möglichkeit: Verwenden von const_cast
   set<Spieler, Vergleich>  spielRunde1(name, name + anzahl);
   set<Spieler, Vergleich>::iterator it;
   for (i=1, n=1; i<=100; i++) {
      it = spielRunde1.find(Spieler(name[rand() % anzahl]));
      // it->addPunkte( rand()%6 + 1 ); // geht nicht
      const_cast<Spieler&>(*it).addPunkte( rand()%6 + 1 );
   }
   for (it = spielRunde1.begin(); it != spielRunde1.end(); ++it)
      cout << n++ << ". " << setiosflags(ios::left) << setw(10) << it->getName()
           << resetiosflags(ios::left) << setw(4) << it->getPunkte() << endl;
   cout << "----------------------------" << endl;

   //............................. 2. Möglichkeit: Entfernen und neu einfuegen
   set<Spieler, Vergleich>  spielRunde2(name, name + anzahl);
   for (i=1, n=1; i<=100; i++) {
      it = spielRunde2.find(Spieler(name[rand() % anzahl]));
      Spieler temp(*it);
      temp.addPunkte( rand()%6 + 1 );
      spielRunde2.erase(it++);
      spielRunde2.insert(it, temp); // Vorschlag fuer Einfuegeposition
   }
   for (it = spielRunde2.begin(); it != spielRunde2.end(); ++it)
      cout << n++ << ". " << setiosflags(ios::left) << setw(10) << it->getName()
           << resetiosflags(ios::left) << setw(4) << it->getPunkte() << endl;
}
```

Programm 4.47 liefert z. B. die folgende Ausgabe:

```
1. Anna        85
2. Hans        86
3. Michel      56
4. Zorro      111
----------------------------
1. Anna        80
2. Hans       119
3. Michel      90
4. Zorro       70
```

4.3.11 Die assoziativen Container-Klassen map und multimap

Maps sind so genannte *assoziative Arrays*. Anders als bei Sets wird bei Maps immer ein Paar bestehend aus einem Schlüssel und den Daten abgespeichert, wobei die einzelnen Elemente nach ihren Schlüsseln sortiert sind. So könnte man z. B. ein Telefonbuch als Map realisieren, wobei der Name der Schlüssel und die entsprechende Telefonnummer und Adresse die zugehörigen Daten zu diesem Schlüssel sind.
Der Hauptunterschied zwischen einer *Map* und einer *Multimap* ist, dass in einer Multimap zu einem Schlüssel mehrere Einträge vorhanden sein dürfen, was bei einer Map nicht möglich ist. Anders als bei Maps können sich in Multimaps auch mehrere identische Werte befinden, denen identische Schlüssel zugeordnet sind.
Verwendet man *Maps* oder *Multimaps*, muss man folgende Headerdatei inkludieren:

```
#include <map>
```

Der Syntax zum Anlegen eines `map`- bzw. `multimap`-Objekts ist z. B. Folgende:

```
map<schlüsselTyp, datenTyp> name
multimap<schlüsselTyp, datenTyp> name
```

Hiermit wird ein `map`- bzw. `multimap`-Objekt mit den Namen *name* angelegt.

Konstruktoren der Klassen map und multimap

Folgende Konstruktoren bieten die Templateklassen `map<schlTyp, datenTyp>` und `multimap<schlTyp, datenTyp>` an:

`map()` $O(1)$

`multimap()` $O(1)$

Default-Konstruktoren (legt leere Map bzw. Multimap an).

`map(InputIterator anfang, InputIterator ende)` $O(n)$

`multimap(InputIterator anfang, InputIterator ende)` $O(n)$

legen eine Map bzw. Multimap an, die mit dem Bereich von einschließlich `anfang` bis ausschließlich `ende` (wird nicht mehr kopiert) initialisiert wird.

`map(const map&)` $O(n)$

`multimap(const multimap&)` $O(n)$

Kopierkonstruktoren

Programm 4.48 – `map1.cpp`:
Demoprogramm zu den map()-Konstruktoren

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

using namespace std;
```

```
int main(void)
{
   typedef pair<string,int> paar;
   paar zahlen[] = { paar("eins",  1), paar("zwei",  2), paar("drei", 3),
                     paar("eins", 11), paar("zwei", 22) };
   unsigned anz = sizeof(zahlen) / sizeof(*zahlen);

   map<string,int> map1;       // Leere Map
   multimap<string,int> mmap1; // Leere Multimap
   map<string,int>::iterator it1;       // Map-Iterator
   multimap<string,int>::iterator mit1; // Multimap-Iterator
   for (unsigned i=0; i < anz; i++) {
      map1.insert(zahlen[i]);
      mmap1.insert(zahlen[i]);
   }
   cout << "SingleMap: "; //.....Ausgabe des Map-Inhalts
   for (it1 = map1.begin(); it1 != map1.end(); it1++)
      cout << "(" << it1->first << ": " << it1->second << "), ";
   cout << endl;
   cout << "MultiMap:  "; //.....Ausgabe des Multimap-Inhalts
   for (mit1 = mmap1.begin(); mit1 != mmap1.end(); mit1++)
      cout << "(" << mit1->first << ": " << mit1->second << "), ";
   cout << endl;

   map<string,int> map2(zahlen, zahlen+4);       // Map initialisiert
   multimap<string,int> mmap2(zahlen, zahlen+4); // Multimap initialisiert
   map<string,int>::iterator it2;       // Map-Iterator
   multimap<string,int>::iterator mit2; // Multimap-Iterator

   cout << "SingleMap2: "; //.....Ausgabe des Map-Inhalts
   for (it2 = map2.begin(); it2 != map2.end(); it2++)
      cout << "(" << it2->first << ": " << it2->second << "), ";
   cout << endl;
   cout << "MultiMap2:  "; //.....Ausgabe des Multimap-Inhalts
   for (mit2 = mmap2.begin(); mit2 != mmap2.end(); mit2++)
      cout << "(" << mit2->first << ": " << mit2->second << "), ";
   cout << endl;
}
```

Programm 4.48 liefert die folgende Ausgabe, an der erkennbar ist, dass die Elemente in den Maps entsprechend ihrem Schlüssel sortiert sind:

```
SingleMap: (drei: 3), (eins: 1), (zwei: 2),
MultiMap:  (drei: 3), (eins: 1), (eins: 11), (zwei: 2), (zwei: 22),
SingleMap2: (drei: 3), (eins: 1), (zwei: 2),
MultiMap2:  (drei: 3), (eins: 1), (eins: 11), (zwei: 2),
```

Konstruktoren mit Festlegung der Sortierung

Die Voreinstellung ist, dass die Elemente in Maps bzw. Multimaps nach ihren Schlüsseln aufwärts sortiert werden. Mit den hier vorgestellten Konstruktoren kann man eine andere Sortierung erreichen, wobei die Sortierung der Schlüssel intern entsprechend dem überladenen `operator()` in der Struktur bzw. Klasse `name` erfolgt.

`map<S, D, name>()` $O(1)$

`multimap<S, D, name>()` $O(1)$

Default-Konstruktoren (legen leere Map bzw. Multimap an), wobei die Sortierung der Schlüssel intern entsprechend dem überladenen `operator()` in der Struktur bzw. Klasse `name` erfolgt.

`map<S, D, name>(InputIterator anfang, InputIterator ende)` $O(n \log n)$

`multimap<S,D,name>(InputIterator anfang,InputIterator ende)` $O(n \log n)$

legen eine Map bzw. Multimap an, die mit dem Bereich von einschließlich `anfang` bis ausschließlich `ende` (wird nicht mehr kopiert) initialisiert wird. Die Sortierung der Schlüssel erfolgt dabei intern entsprechend dem überladenen `operator()` in der Struktur bzw. Klasse `name`.

Der eigene überladene `operator()` in der Struktur bzw. Klasse `name` wird hier immer mit zwei zu vergleichenden Schlüsseln aus der Map bzw. Multimap aufgerufen. Über den Rückgabewert kann man hier steuern, wie diese beiden Schlüssel intern zueinander anzuordnen sind.

Programm 4.49 – `map2.cpp`:
Demoprogramm zu den map/multimap-Konstruktoren mit eigener Sortierung

```
#include <string>
#include <iterator>
#include <map>
#include <iostream>
using namespace std;
struct abwaerts {
  public:
    bool operator() (string str1, string str2) const { return str1 > str2; }
};
int main(void)
{
  typedef pair<string,int> paar;
  paar zahlen[] = { paar("eins",  1), paar("zwei",  2), paar("drei", 3),
                    paar("eins", 11), paar("zwei", 22) };
  unsigned anz = sizeof(zahlen) / sizeof(*zahlen);
   map<string,int,abwaerts> map1;       // Leere Map
   multimap<string,int,abwaerts> mmap1; // Leere Multimap
   map<string,int>::iterator it1;       // Map-Iterator
   multimap<string,int>::iterator mit1; // Multimap-Iterator
   for (unsigned i=0; i < anz; i++) {
      map1.insert(zahlen[i]);
      mmap1.insert(zahlen[i]);
   }
```

```
   cout << "SingleMap: "; //.....Ausgabe des Map-Inhalts
   for (it1 = map1.begin(); it1 != map1.end(); it1++)
      cout << "(" << it1->first << ": " << it1->second << "), ";
   cout << endl;
   cout << "MultiMap:  "; //.....Ausgabe des Multimap-Inhalts
   for (mit1 = mmap1.begin(); mit1 != mmap1.end(); mit1++)
      cout << "(" << mit1->first << ": " << mit1->second << "), ";
   cout << endl;
}
```

Programm 4.49 liefert nun eine Ausgabe, die umgekehrt zur Ausgabe von Programm 4.49 ist:

```
SingleMap: (zwei: 2), (eins: 1), (drei: 3),
MultiMap:  (zwei: 2), (zwei: 22), (eins: 1), (eins: 11), (drei: 3),
```

Methoden zum Einfügen von Elementen in Maps bzw. Multimaps

Während bei Maps folgende Methodenaufrufe keine Auswirkung haben, wenn ein gleicher Schlüssel bereits in der Map vorhanden ist, fügen sie in jedem Fall das entsprechende Schlüssel/Werte-Paar in einer Multimap ein. Bei den folgenden Methoden steht `value_type` für `pair<const` *`schlüsselTyp, datenTyp`*`>`:

`pair<iterator, bool>`
`insert(const value_type& elem_paar)` *`(bei map)`* $O(n \log n)$

`iterator insert(const value_type& elem_paar)` *`(bei multimap)`* $O(n \log n)$
fügt das Schlüssel/Werte-Paar `elem_paar` ein. Bei einer Map wird ein Paar zurückgegeben, in dem der boolesche Wert anzeigt, ob `elem_paar` eingefügt werden konnte, oder nicht (weil ein entsprechender Schlüssel schon vorhanden war). Bei einem erfolgreichen Einfügen zeigt der Iterator des zurückgegebenen Paares auf das neue eingefügte Paar. Sollte kein Einfügen möglich gewesen sein, dann zeigt der Iterator auf das bereits vorhandene Paar in der Map. Bei einer Multimap wird lediglich der Iterator auf das neu eingefügte Element zurückgegeben.

`iterator insert(iterator pos, const value_type& elem_paar)` $O(n \log n)$
fügt das Schlüssel/Werte-Paar `elem_paar` ein und liefert einen Iterator auf dieses eingefügte Element. Da durch das Einfügen die Sortierung erhalten bleiben muss, ist die Position `pos` lediglich ein Vorschlag, der eventuell ein schnelleres Einfügen erlaubt.

`void insert(InputIterator anf, InputIterator end)` $O(n \log n)$
fügt alle Schlüssel/Werte-Paare ein, die sich in dem Bereich von einschließlich `anf` bis ausschließlich `end` (wird nicht mehr eingefügt) befinden.

Programm 4.50 – `map3.cpp`:
Demoprogramm zu insert() bei Maps

```
#include <cstdlib>
#include <ctime>
#include <iterator>
#include <string>
```

```
#include <map>
#include <iostream>
using namespace std;
int main(void) {
  unsigned z;
  srand(time(0));
  typedef pair<string,int> paar;
  string zahlen[] = { "", "eins", "zwei", "drei", "vier", "fuenf", "sechs" };
   map<string,int> single;
   multimap<string,int> multi;
   map<string,int>::iterator sit;
   multimap<string,int>::iterator mit;
   pair<map <string,int>::iterator, bool> p;
   for (unsigned i=1; i<=6; i++) {
      do {
         z = rand()%6+1;
         p = single.insert( paar(zahlen[z], z) );
      } while ( !p.second ); // doppelte Zahl unterbinden
      z = rand()%6+1;
      multi.insert( paar(zahlen[z], z) );
   }
   cout << "single: ";
   for (sit = single.begin(); sit != single.end(); sit++)
       cout << sit->first << ", ";
   cout << endl;
   cout << "multi:  ";
   for (mit = multi.begin(); mit != multi.end(); mit++)
       cout << mit->first << ", ";
   cout << endl;
}
```

Programm 4.50 liefert z. B. die folgende Ausgabe:

```
single: drei, eins, fuenf, sechs, vier, zwei,
multi:  fuenf, sechs, vier, vier, zwei, zwei,
```

Methoden zum Löschen von Elementen in Maps bzw. Multimaps

`void clear()` $O(n)$
: löscht alle Schlüssel/Werte-Paare der Map.

`void erase(iterator pos)` $O(1)$
: löscht das Schlüssel/Werte-Paar an Iterator-Position `pos`.

`void erase(iterator anfang, iterator ende)` $O(\log n)$
: löscht alle Schlüssel/Werte-Paare, die sich in dem Bereich von einschließlich `anfang` bis ausschließlich `ende` (wird nicht mehr gelöscht) befinden.

`size_type erase(const S& schlüssel)` $O(\log n)$
: löscht alle Schlüssel/Werte-Paare der Map, die den Schlüssel `schlüssel` haben, und liefert die Anzahl der gelöschten Elemente als Rückgabewert.

Programm 4.51 – `map4.cpp`:
Demoprogramm zu der Methode erase() bei Multimaps

```
#include <cstdlib>
#include <ctime>
#include <iterator>
#include <map>
#include <iostream>
using namespace std;
int main(void) {
  unsigned i, z=0;
  srand(time(0));
  typedef pair<char,int> paar;

   multimap<char,int> mmap; // Leere Multimap
   multimap<char,int>::iterator it; // Multimap-Iterator
   for (i=0; i < 10; i++)
      mmap.insert(paar('a'+rand()%5, rand()%10));
   while ( !mmap.empty() ) {
      for (it = mmap.begin(); it != mmap.end(); it++)
         cout << it->first << ":" << it->second << ",";
      cout << endl;
      mmap.erase('a'+z);
      z++;
   }
}
```

Programm 4.51 liefert z. B. die folgende Ausgabe:

```
a:0,a:6,b:0,b:6,c:3,d:7,d:5,e:6,e:7,e:2,
b:0,b:6,c:3,d:7,d:5,e:6,e:7,e:2,
c:3,d:7,d:5,e:6,e:7,e:2,
d:7,d:5,e:6,e:7,e:2,
e:6,e:7,e:2,
```

Methoden zum Suchen und Zählen von Elementen in Maps bzw. Multimaps

Alle nachfolgend vorgestellten Methoden, die einen Iterator zurückgeben, liefern einen Iterator auf das Ende (`end()`) der Map bzw. Multimap zurück, wenn kein entsprechendes Element gefunden wird:

`iterator find(const S& schlüssel)` $O(\log n)$
`const_iterator find(const S& schlüssel) const` $O(\log n)$
: suchen beide in der Map bzw. Multimap nach dem ersten Element, das den Schlüssel `schlüssel` besitzt, und liefern einen Iterator auf dieses gefundene Element zurück.

`iterator lower_bound(const S& schlüssel)` $O(\log n)$
`const_iterator lower_bound(const S& schlüssel) const` $O(\log n)$
: sucht in der Map bzw. Multimap nach dem ersten Element, dessen Schlüssel nicht kleiner als `schlüssel` ist, und liefert einen Iterator auf dieses gefundene Element zurück.

`iterator upper_bound(const S& schlüssel)` $O(\log n)$

`const_iterator upper_bound(const S& schlüssel) const` $O(\log n)$

sucht in der Map bzw. Multimap nach dem ersten Element, dessen Schlüssel größer als `schlüssel` ist, und liefert einen Iterator auf dieses gefundene Element zurück.

`pair<iterator, iterator> equal_range(const S& schlüssel)` $O(\log n)$

`pair<const_iterator, const_iterator> equal_range(const S& schlüssel) const` $O(\log n)$

liefert ein Paar von Iteratoren zurück, wobei der erste Iterator dem Iterator entspricht, den auch `lower_bound()` liefern würde, und der zweite dem Iterator, den `upper_bound()` zurückgeben würde. Das Ergebnis entspricht `make_pair(lower_bound(schlüssel), upper_bound(schlüssel))`.

`size_type count(const S& schlüssel) const` $O(\log n)$

gibt zurück, wie viele Elemente in der Map bzw. Multimap den Schlüssel `schlüssel` besitzen.

Programm 4.52 – `map5.cpp`:
Demo zu find(), lower_bound(), upper_bound(), equal_range() und count()

```
#include <cstdlib>
#include <ctime>
#include <map>
#include <iostream>
using namespace std;
int main(void) {
  unsigned z = 0;
  srand(time(0));
   multimap<char,int> zeichen;
   multimap<char,int>::iterator it, it1, it2;
   for (unsigned i=1; i<=15; i++) {
      z = 'A' + rand() % 6; // Zufallsschlüssel: 'A', 'B', 'C'. 'D', 'E', 'F'
      zeichen.insert( pair<char,int>(char(z), z) );
   }
   for (it = zeichen.begin(); it != zeichen.end(); it++)
       cout << it->first << "(" << it->second << ")";
   cout << endl;
   it1 = zeichen.lower_bound('B'); // 'B'-Folge suchen
   it2 = zeichen.upper_bound('B'); // ..............
   for (it = zeichen.begin(); it != it2; it++)
      cout << ( (*it >= *it1) ? "-----" : "     " );
   cout << endl;
   pair<map<char,int>::iterator, map<char,int>::iterator> p; // Anfang/Ende mit
   p = zeichen.equal_range('B');                          // equal_range() suchen
   for (it = zeichen.begin(); it != p.second; it++)
      cout << ( (*it >= *(p.first)) ? "xxxxx" : "     " );
   cout << endl;
   z = zeichen.count('B'); // Zählen, wie oft Schlüssel 'B' vorkommt
   cout << "Schlüssel 'B' kommt " << z << " mal vor" << endl;
}
```

Programm 4.52 liefert z. B. die folgende Ausgabe:

```
A(65)A(65)A(65)A(65)B(66)B(66)B(66)C(67)C(67)D(68)E(69)E(69)E(69)F(70)F(70)
                    ---------------
                    xxxxxxxxxxxxxxx
Schlüssel 'B' kommt 3 mal vor
```

Größe von map- bzw. multimap-Objekten

Methoden zum Erfragen der Größe von `map`- bzw. `multimap`-Objekten sind:

`size_type size() const` $O(1)$
: liefert aktuelle Anzahl von Elementen in der Map bzw. Multimap.

`size_type max_size() const` $O(1)$
: liefert (systemabhängige) maximal mögliche Größe für die jeweilige Map bzw. Multimap.

`bool empty() const` $O(1)$
: liefert `true`, wenn Map bzw. Multimap keine Elemente enthält, sonst `false`.

Vergleichen und Vertauschen von map- bzw. multimap-Objekten

Folgende überladenen globalen Vergleichsoperatoren stehen für `map`- bzw. `multimap`-Objekten zur Verfügung:

`operator==(), operator!=()` $O(n)$
: Zwei Maps bzw. Multimaps sind gleich, wenn sie beide die gleiche Größe haben und die Schlüssel/Werte-Paare jedes Elementpaares an der gleichen Stelle den gleichen Wert besitzen. Andernfalls sind die beiden ungleich.

`operator<(), operator<=(), operator>(), operator>=()` $O(n)$
: Der Kleiner-Vergleich von zwei Maps bzw. Multimaps liefert `true`, wenn beim paarweisen Vergleich im ersten Paar, bei dem sich die Schlüssel und/oder die Werte unterscheiden, der Schlüssel bzw. der Wert in der ersten Map bzw. Multimap kleiner als der in der zweiten ist, andernfalls liefert dieser Vergleich `false`. Die anderen Operatoren lassen sich dann wieder aus dem `operator<` ableiten:

```
template<typename S, typename T>
bool operator<=(const map<S,T>& m1, const map<S,T>& m2) { return !(m2<m1); }
template<typename S, typename T>
bool operator> (const map<S,T>& m1, const map<S,T>& m2) { return m2 < m1;  }
template<typename S, typename T>
bool operator>=(const map<S,T>& m1, const map<S,T>& m2) { return !(m1<m2); }
```

Für Multimaps gilt dann Entsprechendes.
Zum Vertauschen der Inhalte von zwei `map`- bzw. `multimap`-Objekten steht die folgende Methode zur Verfügung:

`void swap(map& m)` $O(1)$

`void swap(multimap& mm)` $O(1)$
: vertauscht die Elemente der eigenen Map bzw. Multimap mit den Elementen von Map `m` bzw. Multimap `mm`.

Programm 4.53 – `map6.cpp`:
Demoprogramm zum Vergleichen und Vertauschen von map-Objekten

```
#include <cstdlib>
#include <ctime>
#include <iterator>
#include <vector>
#include <map>
#include <iostream>
using namespace std;
//.....................................................................ausgabe
void ausgabe(vector<map<char,int> >v) {
   map<char,int>::iterator it;
   for (unsigned i=0; i<5; i++) {
      for (it = v[i].begin(); it != v[i].end(); it++)
         cout << it->first << "(" << it->second << "), ";
      cout << endl;
   }
}
//.....................................................................main
int main(void)
{
   unsigned i, j;
   srand(time(0));

   vector<map<char,int> > v(5); // Vektor mit map-Elementen
   for (i=0; i<5; i++)
      for (j=0; j<5; j++)
         v[i].insert( pair<char,int>(char('A' + rand() % 6), rand()%10) );
   ausgabe(v);
   cout << "---------------------------------------" << endl;
   for (i=0; i<4; i++)
      for (j=i+1; j<5; j++)
         if (v[i] > v[j])
            v[i].swap(v[j]);
   ausgabe(v);
}
```

Programm 4.53 liefert z. B. die folgende Ausgabe:

```
B(8), C(3), E(6), F(9),
A(4), B(9), C(1), E(4),
B(1), E(2), F(2),
A(4), B(3), E(3), F(8),
A(4), B(3), C(3), D(9), F(4),
---------------------------------------
A(4), B(3), C(3), D(9), F(4),
A(4), B(3), E(3), F(8),
A(4), B(9), C(1), E(4),
B(1), E(2), F(2),
B(8), C(3), E(6), F(9)
```

Iterator-Methoden der Klassen map und multimap

`iterator begin()` $O(1)$

`const_iterator begin() const` $O(1)$

liefern beide einen Iterator auf das erste Element der Map bzw. Multimap.

`iterator end()` $O(1)$

`const_iterator end() const` $O(1)$

liefern beide einen Iterator auf die Position hinter dem letzten Element der Map bzw. Multimap.

`reverse_iterator rbegin()` $O(1)$

`const_reverse_iterator rbegin() const` $O(1)$

liefern beide einen *reverse*-Iterator auf das letzte Element der Map bzw. Multimap.

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

liefern beide einen *reverse*-Iterator auf die Position vor dem ersten Element der Map bzw. Multimap.

Der Indexoperator operator[] bei der Klasse map

Die Klasse `map` bietet anders als die Klasse `multimap` den Indexoperator `[]` an, um über einen Schlüssel auf einen Wert in der Map zuzugreifen:

`data_type& operator[](const S& schlüssel)` (*nur bei map*) $O(n \log n)$

liefert eine Referenz auf den Wert (Objekt), das dem Schlüssel `schlüssel` in der Map zugeordnet ist. Sollte die Map den angegebenen Schlüssel nicht enthalten, wird automatisch ein neues Datenelement zu diesem Schlüssel angelegt, das entweder den Default-Wert erhält oder aber mittels des entsprechenden Default-Konstruktors angelegt wird. Multimaps bieten den Indexoperator nicht an, was ja auch nachvollziehbar ist, denn welcher Wert sollte dort zurückgeliefert werden, wenn der entsprechende Schlüssel mehrfach in der Map vorhanden ist.

Programm 4.54 – `map7.cpp`:
Demoprogramm zum Indexoperator bei map-Objekten

```
#include <string>
#include <map>
#include <iomanip>
#include <iostream>
using namespace std;

int main(void) {
  map<string,int> m; // Map anlegen
  map<string, int>::iterator it; // Map-Iterator
   m["Zorro"] = 49;  // Mittels Indexoperator eintragen
   m["Emil"]  = 30;
   m.insert(make_pair("Hans", 22)); // Mittels insert eintragen
   for (it = m.begin(); it != m.end(); ++it)
      cout << setw(6) << it->first << ": " << it->second << endl;
   cout << "--------------------------------" << endl;
```

```
   if (m["Gerta"] == 28)  // Legt Gerta-Schlüssel mit Default-Wert 0 an
      cout << "...Gerta ist nicht 28 Jahre alt" << endl;
   for (it = m.begin(); it != m.end(); ++it)
      cout << setw(6) << it->first << ": " << it->second << endl;
}
```

Programm 4.54 liefert die folgende Ausgabe:

```
  Emil: 30
  Hans: 22
 Zorro: 49
--------------------------------
  Emil: 30
 Gerta: 0
  Hans: 22
 Zorro: 49
```

Programm 4.55 erstellt wie Programm 4.44 eine Wortstatistik zu einer Datei, wobei es allerdings eine Map statt einer Multiset dazu verwendet. Als Schlüssel in der Map verwendet es die extrahierten Worte, deren Werte (Zähler) es dann mittels dem Indexoperator erhöht.

Programm 4.55 – `map8.cpp`:
Wortstatistik zu einer Datei unter Verwendung einer Map

```
#include <cctype>
#include <iterator>
#include <map>
#include <string>
#include <iomanip>
#include <iostream>
using namespace std;
int main(void) {
  ifstream datei("map8.cpp"); // Input-Stream auf Datei
  istreambuf_iterator<char> it(datei), end; // auf Dateianfang und fuer Ende
   map<string, int> worte; // Map (Wort als Schluessel und Zaehler als Wert)
   string str;
   for ( ; it != end; it++) {
      if (isalpha(*it))
         str += tolower(*it);
      else {
         if (isalpha(str[0]))
            worte[str]++;
         str.clear();
      }
   }
   map<string, int>::iterator wit;
   for (wit = worte.begin(); wit != worte.end(); wit++)
      cout << setw(15) << setiosflags(ios::left)    << wit->first << ":"
           << setw(10) << setiosflags(ios::right)   << wit->second
                       << resetiosflags(ios::right) << endl;
   cout << "....include kommt " << worte["include"] << " mal vor" << endl;
}
```

Programm 4.55 liefert die folgende Ausgabe:

```
als             :         2
auf             :         2
begin           :         1
cctype          :         1
.........................
.........................
wert            :         1
wit             :         6
wort            :         1
worte           :         4
zaehler         :         1
....include kommt 9 mal vor
```

Der Datentyp value_type

Jede Map bzw. Multimap definiert einen Typ `value_type`, der benutzt werden kann, um Elemente zu erzeugen, die in eine Map bzw. Multimap eingefügt werden können.
Um z. B. ein Element `alter` zu erzeugen, das zu einem `map<string,int>`-Objekt hinzugefügt werden kann, könnte man Folgendes angeben:

```
map<string,int>::value_type alter("Fritz", 27);
```

Verwendet man solche `value_type`-Objekte, ist es ratsam, an diese einen eigenen Typnamen zu vergeben, wie z. B.:

```
typedef map<string,int>::value_type personAlter;
  //.... Nun ist folgende Definition möglich:
personAlter name("Fritz", 27);
```

Es ist auch möglich, anonyme `value_type`-Objekte zu erzeugen, um diese z. B. an eine Funktion zu übergeben, wie z. B.:

```
fkt(map<string,int>::value_type("Fritz", 27));
//... bzw. bei eigenen Typnamen auch:
fkt(personAlter("Fritz", 27));
```

Programm 4.56 – `map9.cpp`:
Demoprogramm zu value_type

```
#include <string>
#include <iterator>
#include <map>
#include <iomanip>
#include <iostream>
using namespace std;

void ausgabe(map<string,int>::value_type elem)
{
   cout << elem.first << ": " << elem.second << endl;
}
```

```
int main(void)r
 {
   typedef map<string, int>::value_type nameAlter;
   typedef multimap<int, bool>::value_type primZahl;

   //.... Anlegen von Elementen, die in einer Map eingefügt werden können
   map<string, int>::value_type name1("Hansi", 38);
   nameAlter name2("Anton", 54);
   map<string, int>::value_type name3("Berta", 25);

   //.... Einfügen der zuvor erzeugten Elemente
   map<string, int> personen;
   personen.insert(name1);
   personen.insert(name2);
   personen.insert(name3);

   //.... Ausgeben aller Elemente der Map
   map<string, int>::iterator it;
   for (it = personen.begin(); it != personen.end(); it++)
      ausgabe(*it);

   //.... Übergeben von anonymen value_type-Objekten
   ausgabe(map<string,int>::value_type("Micha", 44));
   ausgabe(nameAlter("Erwin", 97));

   //.... Anlegen von Elementen, die in einer Map eingefügt werden können
   map<int, bool>::value_type z[] = { make_pair(1, false),
                                      make_pair(2, true), make_pair(3, true),
                                      make_pair(5, true), make_pair(7, true) };

   //.... Anlegen einer Map mit zuvor erzeugten Elementen
   map<int, bool> zahlen(z, z + sizeof(z)/sizeof(*z));
   //.... Hinzufügen weiterer Elemente zur Map
   for (int i=0; i < 10; i++)
      if (i != 2 && i%2==0)
         zahlen.insert(make_pair(i, false));
   primZahl nochEineZahl(9, false); // neues value_type
   zahlen.insert(nochEineZahl);
   zahlen.insert(primZahl(20, false)); // anonymes value_type
   zahlen[37] = true;

   //.... Ausgeben aller Elemente der Map
   cout << "....Primzahlen sind mit true gekennzeichnet" << endl;
   map<int, bool>::iterator it2;
   for (it2 = zahlen.begin(); it2 != zahlen.end(); it2++)
      cout << setw(2) << it2->first << ": " << boolalpha << it2->second << endl;
}
```

Programm 4.56 liefert die folgende Ausgabe:

```
Anton: 54
Berta: 25
Hansi: 38
Micha: 44
Erwin: 97
....Primzahlen sind mit true gekennzeichnet
 0: false
 1: false
 2: true
 3: true
 4: false
 5: true
 6: false
 7: true
 8: false
 9: false
20: false
37: true
```

4.3.12 Übungen

Primzahlen mit dem Sieb des Eratosthenes

Das *Sieb des Eratosthenes* zur Ermittlung von Primzahlen verwendet folgende Vorgehensweise: Zunächst wird auf die Zahl 2 positioniert, und dann alle Vielfachen von 2 herausgestrichen (fette Zahlen verbleiben im Array):

```
1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  ...
   ^
```

Im nächsten Schritt wird um eine Zahl weiterpositioniert, in unserem Fall also auf 3. Ist diese Zahl noch vorhanden, so handelt es sich um eine Primzahl, von der wiederum alle Vielfachen zu streichen sind:

```
1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  ...
      ^
```

Die nächste Primzahl nach 3 wäre dann 5 (noch nicht gestrichen), von der wieder alle Vielfachen zu streichen sind usw. Dieses Verfahren wird wiederholt, bis die Zahl `n` erreicht ist. Die im Array nicht gestrichenen Zahlen sind dann Primzahlen.

Erstellen Sie ein Programm `primsieb.cpp`, das Primzahlen nach diesem Verfahren ermittelt, wobei es dazu den Datentyp `vector` verwendet, der boolesche Werte enthält: `false` zeigt dabei an, dass die entsprechende Zahl gestrichen wurde.

Möglicher Ablauf des Programms `primsieb.cpp`:

```
Primzahlen bis wohin: 1000 ⏎
       2       3       5       7      11      13      17      19      23      29
     ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
     877     881     883     887     907     911     919     929     937     941
     947     953     967     971     977     983     991     997
```

Sortieren von Dateien

Erstellen Sie ein Programm `sortfile.cpp`, das den Inhalt (Zeilen) der Datei, die als erstes Argument angegeben ist, sortiert und in die als zweites Argument angegebene Datei ausgibt. Dabei sollten Sie die einzelnen Zeilen in einen `string`-Vektor einlesen. Ein Vergleichen von `string`-Variablen ist im übrigen mit den Operatoren <, > usw. möglich. Hat z. B. die Datei `namen.txt` den folgenden Inhalt:

```
Mueller, Hans
Wolters, Toni
Baller, Marion
Aller, Michael
```

und man ruft
./sortfile namen.txt namensort.txt
auf, so sollte danach die Datei `namensort.txt` folgenden Inhalt haben:

```
Aller, Michael
Baller, Marion
Mueller, Hans
Wolters, Toni
```

Folgen von Nullen und Einsen

Erstellen Sie ein Programm `nulleins.cpp` das zunächst eine zufällige Folge von Nullen und Einsen erzeugt. Diese Folge soll dann in weiteren Schritten wie folgt komprimiert werden: Aus zwei gleichen aufeinanderfolgenden Ziffern wird eine 0 und aus zwei unterschiedlichen aufeinanderfolgenden Ziffern wird eine 1. Diese Schritte werden solange wiederholt, bis nur noch eine Ziffer übrigbleibt.
Verwenden Sie zur Lösung dieser Aufgabenstellung zwei `deque`-Objekte.
Mögliche Abläufe des Programms `nulleins.cpp` sind:

```
Läge der 0/1-Folge: 9 ⏎
110101011
01111
101
11
```

```
Länge der 0/1-Folge: 71 ⏎
11001111101001001000111010101010100011100100010111100011100001011000010
00001110100111111001101101001011011
000111001110101010
010001111
10101
111
01
1
```

Sortieren und Mischen von Listen

Erstellen Sie ein Programm `listmisch.cpp`, das zunächst in zwei unterschiedlichen Listen jeweils Zufallszahlen zwischen 0 und 9 ablegt, und dann diese beiden Listen ausgibt. Anschließend soll es diese beiden Listen sortieren und ausgeben, bevor es dann diese beiden Listen mischt und die gemischte Liste ausgibt. Am Ende soll dieses Programm noch in der gemischten Liste alle gleichen aufeinderfolgenden Zahlen durch eine Zahl ersetzen und diese Liste dann erneut ausgeben.
Ein möglicher Ablauf des Programms `listmisch.cpp` ist z. B.:

```
       zahlen1 (unsortiert): 7,4,9,0,6,8,3,8,4,0,
       zahlen2 (unsortiert): 3,1,3,2,8,6,7,5,8,3,
         zahlen1 (sortiert): 0,0,3,4,4,6,7,8,8,9,
         zahlen2 (sortiert): 1,2,3,3,3,5,6,7,8,8,
                   gemischt: 0,0,1,2,3,3,3,3,4,4,5,6,6,7,7,8,8,8,8,9,
 gemischt (keine doppelten): 0,1,2,3,4,5,6,7,8,9,
```

Eine Ringliste

Im Jahre 67 n. Chr. wurde die galiläische Stadt Jotapata, unter Führung des jüdischen Historikers Josephus (37-100) ein Zentrum des antirömischen Widerstands, nach 47tägiger Belagerung von Kaiser Vespasian eingenommen. Josephus und 40 Soldaten zogen sich in eine Zisterne zurück. Um der Sklaverei zu entgehen, wollten die Soldaten sich selbst umbringen. Josephus beschwor sie vergebens, davon abzulassen. Damit er wenigstens seinen Freund und sich selbst rettet, schlug Josephus als Tötungsritual den alten römischen Brauch der *decimatio* (Aussonderung jedes Zehnten) vor. An welche Stelle des Kreises stellte er seinen Freund und sich, um zu überleben? Erstellen Sie ein Programm `josephus.cpp`, das die Nummern der Personen in einer Liste (`list`-Objekt) einordnet, bevor es die einzelnen Nummern nach der oben beschriebenen Regel aus der Liste mittels eines Iterators löscht, wobei es jede gelöschte Nummer immer ausgibt. Nach jedem Löschen einer Nummer gilt der Kreis als wieder geschlossen.
Mögliche Abläufe des Programms `josephus.cpp` sind:

```
Wie viele Personen: 41 ⏎
Der wievielte soll immer ausgesondert werden: 10 ⏎
In folgender Reihenfolge wird ausgesondert:
10, 20, 30, 40, 9, 21, 32, 2, 14, 26, 38, 11, 24, 37, 12, 27, 1, 17, 34, 8, 29, 6,
28, 7, 33, 16, 41, 25, 18, 5, 3, 39, 4, 15, 23, 13, 36, 22, 31, 19, 35,
Wie viele Personen: 100 ⏎
Der wievielte soll immer ausgesondert werden: 2 ⏎
In folgender Reihenfolge wird ausgesondert:
2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44,
46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86,
88, 90, 92, 94, 96, 98, 100, 3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55,
59, 63, 67, 71, 75, 79, 83, 87, 91, 95, 99, 5, 13, 21, 29, 37, 45, 53, 61, 69, 77,
85, 93, 1, 17, 33, 49, 65, 81, 97, 25, 57, 89, 41, 9, 73,
```

Rückwärtige Ausgabe einer Datei

Erstellen Sie ein Programm `stackrueck.cpp`, das unter Verwendung der Klasse `stack` den Inhalt der Datei, deren Name auf der Kommandozeile angegeben ist, rückwärts ausgibt. Hat z. B. die Datei `namen.txt` den folgenden Inhalt:

```
Mueller, Hans
Wolters, Toni
Baller, Marion
Aller, Michael
```

und man ruft **./stackrueck namen.txt** auf, so sollte Folgendes ausgegeben werden:

```

leahciM ,rellA
noiraM ,rellaB
inoT ,sretloW
ztirF ,sreblA
snaH ,relleuM
```

Kubikzahlen über Polyas Sieb

Erstellen Sie ein Programm `polya.cpp`, das das so genannte *Polya Sieb* realisiert:

- Die natürlichen Zahlen werden zunächst in einer Reihe hingeschrieben.

```
        1   2   3   4   5   6   7   8   9   10   11   12   13....
```

- Danach streicht man – beginnend mit der 3 – jede dritte Zahl heraus.

```
Durch Herausstreichen jeder dritten Zahl erhält man dann:
        1   2   4   5   7   8   10   11   13....
```

- Schließlich betrachtet man die Summenfolge der so verbleibenden Zahlen.

```
Die Summenfolge ist dann:
        1
        3 = 1+2
        7 = 1+2+4
       12 = 1+2+4+5
       19 = 1+2+4+5+7
       27 = 1+2+4+5+7+8
       37 = 1+2+4+5+7+8+10
       48 = 1+2+4+5+7+8+10+11
       ..................
```

- Streicht man nun beginnend mit der zweiten Zahl jede zweite Zahl aus dieser Summenfolge und
- bildet dann erneut die Summenfolge, so ergibt sich eine Zahlenfolge, die die Kubikzahlen enthält.

Sie sollten Polyas Sieb dadurch realisieren, dass sie die Zahlen immer wieder zwischen zwei Queues hin und herschieben, wobei beim Übertragen aus einer in die andere Queue immer das nächste der oben erwähnten Kriterien angewendet wird.

Ein möglicher Programmablauf von `polya.cpp` ist z. B.:

```
Wie viele Zahlen: 25 ⏎
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, .... 19, 20, 21, 22, 23, 24, 25,
1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19, 20, 22, 23, 25,
1, 3, 7, 12, 19, 27, 37, 48, 61, 75, 91, 108, 127, 147, 169, 192, 217,
1, 7, 19, 37, 61, 91, 127, 169, 217,
1, 8, 27, 64, 125, 216, 343, 512, 729,
```

Das Phänomen gleicher Geburtstage

Eines der verblüffensten Beispiele aus der Wahrscheinlichkeitstheorie ist die relativ niedrige Anzahl von Personen, die notwendig sind, damit in einer solchen Gruppe zwei Personen am gleichen Tag Geburtstag haben.

Dazu notwendige Formeln aus der Wahrscheinlichkeitstheorie:

Nach Vorgabe von `n` wird die Wahrscheinlichkeit `q` bestimmt, dass alle `n` Personen verschiedene Geburtstage haben.

Die gesuchte Wahrscheinlichkeit ist dann $p = 1 - q$.

Die Wahrscheinlichkeit, dass bei einer Gruppe von `n` Personen alle an einem verschiedenen Tag des Jahres Geburtstag haben, ist:

$$q = \frac{365}{365} \cdot \frac{364}{365} \cdot \ldots \cdot \frac{365 - n + 1}{365}$$

Erstellen Sie nun ein Programm *gebwahr.cpp* das zunächst einliest, wie viele Personen `n` sich in einer Gruppe befinden, bevor es dann die Wahrscheinlichkeit ausgibt, dass in dieser Gruppe zwei Personen am gleichen Tag Geburtstag haben. Allerdings sollte dieses Programm nicht die mathematische Formel dabei verwenden, sondern dies simulieren, indem es `x` Simulationsläufe für eine solche Gruppe von `n` Personen durchführt.

Das Programm *gebwahr.cpp* soll dabei eine Set verwenden, in der es versucht, für jede Person eine Zufallszahl zwischen 0 und 365 einzutragen. Sollte dies nicht möglich sein, ist diese Zahl vorhanden und für diesen Durchlauf hatten zwei Personen am gleichen Tag Geburtstag.

Mögliche Abläufe des Programms `gebwahr.cpp`:

```
Wie viele Personen: 23 ⏎
Wie viele Simulationen: 100000 ⏎
Die Wahrscheinlichkeit, dass mind. 2 Personen am gleichen Tag
Geburtstag haben ist: 50.65%
```

```
Wie viele Personen: 43 ⏎
Wie viele Simulationen: 10000 ⏎
Die Wahrscheinlichkeit, dass mind. 2 Personen am gleichen Tag
Geburtstag haben ist: 92.32%
```

Umsatzberechnung (Maps in einer Map)

Erstellen Sie ein Programm `mapumsatz.cpp`, das aus einer Datei die Umsätze der unterschiedlichen Abteilungen liest, diese dann zusammenaddiert und entsprechend aufbereitet ausgibt. Hat die Datei `umsatz.txt` z. B. den folgenden Inhalt:

```
12/2005:  23453.56
03/2006:   4444.12
01/2006:   6353.42
12/2005:  63674.65
02/2006:  90963.73
12/2005:  52523.84
02/2006:   3453.55
01/2006:  12342.33
11/2005:  62353.43
03/2006:  53535.64
```

und man ruft das Programm wie folgt auf
./mapumsatz umsatz.txt
so sollte es die folgende Ausgabe liefern:

```
.........................2005
          11:     62353.43
          12:    139652.05
      Gesamt:    202005.48
.........................2006
          01:     18695.75
          02:     94417.28
          03:     57979.76
      Gesamt:    171092.79
```

Verwenden Sie bei diesem Programm eine Map mit dem jeweiligen Jahr als Schlüssel, wobei in dieser Map dann aber die einzelnen Summen für die Monate wieder als Maps realisiert sein sollten, wie z. B.:

```
   map<int, map<int, double> > jahrUmsatz;
```

Zum Durchlaufen benötigen Sie nun zwei Iteratoren:

```
   map<int, map<int, double> >::iterator jit; // für Jahre
   map<int, double>::iterator mit; // für Monate im jeweiligen Jahr
```

4.4 Iteratoren

4.4.1 Allgemeines zu Iteratoren

Für Iteratoren gilt Folgendes:

Iteratoren sind abstrakte Zeiger,

die das schrittweise Iterieren über alle Elemente eines abstrakten Datentyps (ADTs) oder Containers ermöglichen, wobei sie unabhängig von der jeweiligen Implementierung der ADTs bzw. Containern sind. Für den Benutzer verhalten sie sich dabei wie Zeiger.

Zeiger sind einfache Iteratoren

Deshalb ist es auch möglich, dass man z. B. die STL-Algorithmen `reverse()` (Reihenfolge der Elemente in einem Bereich umkehren) oder `sort()` auf normale C-Arrays anwenden kann:

```
int a[100];
...
reverse(a+1, a+4); // Inhalt eines Bereiches im Array a umkehren
sort(a, a+20);     // Ersten 20 Werte im Array a sortieren
```

Iteratoren machen STL-Algorithmen unabhängig von STL-Containern

Algorithmen sind Templates, deren Parameter Iteratoren sind, so dass sie nicht auf einen bestimmten Datentyp (Container) eingeschränkt sind. Der C++-Erfinder *Bjarne Stroustrup* machte hierzu folgende Aussage: *„Iteratoren sind der Klebstoff, der Container und Algorithmen zusammenhält“*.

begin() und end() bei Containern, die Iteratoren anbieten

Container, die Iteratoren anbieten, stellen grundsätzlich die beiden folgenden Methoden zur Verfügung:

- `begin()` zeigt auf das erste Element des Containers.
- `end()` zeigt hinter das letzte Element des Containers.

Programm 4.57 – `iterator1.cpp`:
Einfaches Demoprogramm zu Iteratoren

```
 1  #include <vector>
 2  #include <iostream>
 3  using namespace std;
 4
 5  int main(void)
 6  {
 7     vector<char> v(10); // deklariert einen Vector mit 10 char-Elementen
 8     vector<char>::iterator it; // definiert einen char-Iterator für vector
 9
10     for (unsigned i=0; i < 10; i++)
11        v[i] = 'a' + i;
12     for (it=v.begin(); it < v.end(); it++)
13        cout << *it << ", ";
14     cout << endl;
15  }
```

Programm 4.57 liefert die folgende Ausgabe:

```
a, b, c, d, e, f, g, h, i, j,
```

In Programm 4.57 wird Folgendes ersichtlich:

- Da der in Zeile 8 definierte Iterator auf einen Vektor anzuwenden ist, in dem Zeichen gespeichert sind, muss man dies entsprechend bei der Definition – wie in Zeile 8 gezeigt – angeben.
- Die `for`-Schleife in Zeile 12 weist dem Iterator mit `v.begin()` die Position des ersten Elements zu, und durchläuft den Vektor bis seine Ende (`v.end()`) erreicht ist. Es ist zu beachten, dass hier auf `<` und nicht auf `<=` geprüft wird, da `v.end()` nicht auf das letzte Element, sondern hinter das letzte Element zeigt.
- Um über einen Iterator auf das Element zuzugreifen, auf das er aktuell zeigt, muss dieser wie ein Zeiger dereferenziert werden, was in Zeile 13 mittels `*` auch geschieht.

const-Iteratoren

Neben Iteratoren, die eine Änderung der Elemente erlauben, existieren meist auch `const`-Iteratoren, die keine Änderung über sie erlauben:

```
<ContainerName>::iterator       // erlaubt Änderung der Elemente
<ContainerName>::const_iterator // erlaubt keine Änderung der Elemente
```

Programm 4.58 z. B. würde alle auf der Kommandzeile angegebenen Strings nacheinander (mit Komma getrennt) ausgeben.

Programm 4.58 – `iterator2.cpp`:
Einfaches Demoprogramm zu const-Iteratoren

```
#include <string>
#include <vector>
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {
   const vector<string> v(argv, argv+argc); // deklariert const-Vektor
   vector<string>::const_iterator it; // nicht erlaubt wäre:
                                      // vector<string>::iterator it;
   for (it=v.begin(); it < v.end(); it++)
      cout << *it << ", ";
}
```

Überladene Operatoren bei Iteratoren

Alle Iteratoren stellen folgende überladenen Operatoren zur Verfügung:

- `operator==()` und `operator!=()`
  errlauben das Vergleichen von Iteratoren.
- `operator++()` (Präfix) und `operator++(int)` (Postfix)
  positionieren den Iterator im Container weiter auf das nächste Element, wie z. B.:

```
vector<char> v(10); // deklariert einen Vector mit 10 char-Elementen
vector<char>::iterator it; // definiert einen char-Iterator für vector
it = v.begin(); // setzt Iterator auf das erste Element des Vektors v
```

```
while (it != v.end() { // alle Elemente des Vektors v durchlaufen
   cout << *it << ", ";
   ++it; // Iterator auf nächstes Element positionieren
}
```

- `operator->()` und `operator*()`

  Mittels dieser Dereferenzierungsoperatoren kann man auf das Element zugreifen, auf das ein Iterator gerade zeigt. Programm 4.59 ist ein Demonstrationsprogramm zu den Dereferenzierungsoperatoren. Es zeigt aber auch ein Funktionstemplate, mit dem man eine frei wählbare Methode für alle Elemente in einem beliebigen Container (hier `vector`) aufrufen lassen kann. Dazu muss diesem Funktionstemplate `methAufruf()` ein Zeiger auf eine Methode übergeben werden.

Programm 4.59 – `iterator3.cpp`:
Demoprogramm zu den Dereferenzierungsoperatoren bei Iteratoren

```
#include <vector>
#include <iterator>
#include <iomanip>
#include <iostream>
using namespace std;
template<class Container, class fktZgr>
void methAufruf(Container& c, fktZgr f) {
   typename Container::iterator it = c.begin();
   for ( ; it != c.end() ; it++)
     ((*it).*f)();
     // nicht mögl.: it->*f)(); da Iteratoren keinen operator->* anbieten
}
class K {
     friend ostream& operator<<(ostream& os, const K& k) {
        return os << k.w;
     }
  public:
     K()      : w(0) {}
     K(int x) : w(x) {}
     void ausgabe()   { cout << setw(2) << w << ", "; }
     void verdoppel() { w *= 2;                       }
  private:
     int w;
};
int main(void) {
   vector<K> v(5);
   for (int i = 0; i < 5; i++)
      v[i] = K(i);
   // Explizites schrittw. Durchlaufen der Elemente mit Methodenaufrufe
   vector<K>::iterator it;
   for (it = v.begin(); it != v.end(); it++) {
      (*it).ausgabe();
      it->verdoppel();
   }
   cout << endl;
```

```
   for (it = v.begin(); it != v.end(); it++)
      it->ausgabe();
   cout << endl;

   // Methodenaufrufe für alle Elemente mittels Funktionstemplate
   methAufruf(v, &K::verdoppel);
   methAufruf(v, &K::ausgabe);    cout << endl;
}
```

Programm 4.59 liefert die folgende Ausgabe:

```
 0,  1,  2,  3,  4,
 0,  2,  4,  6,  8,
 0,  4,  8, 12, 16,
```

Iteratoren mit Zeigerarithmetik bei manchen Containern

Bei Containern wie z. B. `vector` und `deque`, deren Elemente zusammenhängend gespeichert sind, wird bei den Iteratoren auch Zeigerarithmetik unterstützt: `[]`, `-`, `+`, `-`, `+=`, `-=`, `<`, `<=`, `>` und `>=`.

Programm 4.60 – `iterator4.cpp`:
Iteratoren mit Zeigerarithmetik

```
#include <vector>
#include <iterator>
#include <iostream>
using namespace std;

int main(void)
{
   vector<int> v(10);

   for (int i = 0; i < 10; i++)
      v[i] = i+1;

   vector<int>::iterator it;
   for (it = v.begin(); it < v.end(); it += 2) // Ausgabe jeder 2. Zahl
      cout << *it << ", ";
   cout << endl;

   for (it = v.end()-5; it >= v.begin(); it--) // Rückwärtsausgabe ab 6
      cout << *it << ", ";
   cout << endl;
}
```

Programm 4.60 würde die folgende Ausgabe liefern:

```
1, 3, 5, 7, 9,
6, 5, 4, 3, 2, 1,
```

4.4.2 Iterator-Kategorien

Die Iteratoren der STL sind in bestimmte Kategorien unterteilt:

Input-Iterator – Einmaliges Vorwärtslesen (read-only)

Ein Input-Iterator iteriert vorwärts über einen Input-Stream oder einen Container, wobei nur ein lesender Zugriff über ihn erlaubt ist. Über einen Input-Iterator kann man weder Daten schreiben noch kann man mehrmals über die Elemente iterieren.
Algorithmen, die Input-Iteratoren verwenden, sind so genannte „Einweg-Algorithmen“.
Als Operationen auf Input-Iteratoren sind erlaubt:

- Zuweisungsoperator und Kopierkonstruktor
- Inkrementieren mit ++ (Post- und Präfix)
- Vergleichen mit ==, != (und eventuell <, <=, > und >=)
- Dereferenzieren mit * (nur lesend) und ->

Ein spezieller Konstruktor definiert hier den Wert, der sich hinter dem letzten Element befindet.
Vordefinierte Input-Iteratoren sind `istream_iterator` und `istreambuf_iterator`, um aus einem `istream`-Objekt zu lesen.
Ein Algorithmus, der Input-Iteratoren verwendet, ist z. B. `copy()`, der beispielsweise wie in Programm 4.61 realisiert sein könnte.

Programm 4.61 – `iterator5.cpp`:
Mögliche Realisierung des STL-Algorithmus copy()

```
#include <vector>
#include <list>
#include <iostream>
using namespace std;

template<typename InputIterator, typename OutputIterator>
OutputIterator
my_copy(InputIterator anfang, InputIterator ende, OutputIterator ziel) {
   while (anfang != ende)
      *ziel++ = *anfang++;
   return ziel;
}

int main(void) {
   unsigned i;
   vector<int> v(5), v2(5);
   vector<int>::iterator it;
   for (i = 0; i < v.size(); i++)
      v[i] = i+1;
   my_copy(v.begin()+1, v.end()-1, v2.begin());

   cout << "v2(" << v2.size() << "): ";
   for (it = v2.begin(); it != v2.end(); it++)
      cout << *it << ", ";
   cout << endl;
```

```
   list<int> l, l2(5);
   list<int>::iterator lit;
   for (i = 0; i < 5; i++)
      l.push_back(i+1);
   my_copy(l.begin(), l.end(), l2.begin());
   cout << "l2(" << l2.size() << "): ";
   for (lit = l2.begin(); lit != l2.end(); lit++)
      cout << *lit << ", ";
   cout << endl;
}
```

Programm 4.61 würde die folgende Ausgabe liefern:

```
v2(5): 2, 3, 4, 0, 0,
l2(5): 1, 2, 3, 4, 5,
```

Output-Iterator – Einmaliges Vorwärtsschreiben (write-only)

Ein Output-Iterator iteriert vorwärts über einen Output-Stream oder einen Container, wobei nur ein schreibender Zugriff über ihn erlaubt ist. Über einen Output-Iterator kann man weder Daten lesen noch kann man mehrmals über die Elemente iterieren.
Algorithmen, die Output-Iteratoren verwenden, sind so genannte „Einweg-Algorithmen“.
Als Operationen auf Output-Iteratoren sind erlaubt:

- Zuweisungsoperator und Kopierkonstruktor
- Inkrementieren mit ++ (Post- und Präfix)
- Dereferenzieren mit * (nur schreibend)

Vergleichsoperationen stehen im Allgemeinen nicht zur Verfügung, da gleichzeitig niemals mehr als ein Output-Iterator für einen Container benutzt werden soll.
Anders als bei Input-Iteratoren steht hier kein Konstruktor für den Wert zur Verfügung, der sich hinter dem letzten Element befindet.
Vordefinierte Output-Iteratoren sind *Inserter-Iteratoren* (siehe auch Seite 245) und auch `ostream_iterator` bzw. `ostreambuf_iterator`, um in ein `ostream`-Objekt zu schreiben.
Ein Algorithmus, der Output-Iteratoren verwendet, ist z. B. `fill_n()`, der wie in Programm 4.62 realisiert sein könnte.

Programm 4.62 – `iterator6.cpp`:
Mögliche Realisierung des STL-Algorithmus fill_n()

```
#include <vector>
#include <list>
#include <string>
#include <iostream>
using namespace std;

template<typename OutputIterator, typename Size, typename T>
OutputIterator my_fill_n(OutputIterator ziel, Size n, const T& wert) {
   while (n-- > 0)
      *ziel++ = wert;
   return ziel;
}
```

```
int main(void) {
   vector<int> v(9);
   vector<int>::iterator it;
   my_fill_n(v.begin()+2, 5, 100);
   cout << "v(" << v.size() << "): ";
   for (it = v.begin(); it != v.end(); it++)
      cout << *it << ", ";
   cout << endl;

   list<string> l(6);
   list<string>::iterator lit;
   my_fill_n(l.begin(), 4, "Hallo");
   cout << "l(" << l.size() << "): ";
   for (lit = l.begin(); lit != l.end(); lit++)
      cout << *lit << ", ";
   cout << endl;
}
```

Programm 4.62 würde die folgende Ausgabe liefern:

```
v(9): 0, 0, 100, 100, 100, 100, 100, 0, 0,
l(6): Hallo, Hallo, Hallo, Hallo, , ,
```

Forward-Iterator – Mehrmaliges Vorwärtslesen und -schreiben (read/write)

Ein Forward-Iterator besitzt sowohl die Funktionalität eines Input-Iterators als auch die eines Output-Iterators und erlaubt anders als diese ein mehrmaliges Iterieren des gleichen Bereichs, indem man erneut von einer zuvor gemerkten Position weitere Iterationen startet. Deshalb können „Mehrweg-Algorithmen“ mit Forward-Iteratoren realisiert werden. Ein Forward-Iterator kann sich allerdings ausschließlich vorwärts bewegen.

Als Operationen auf Forward-Iteratoren sind erlaubt:

- Zuweisungsoperator und Kopierkonstruktor
- Inkrementieren mit `++` (Post- und Präfix)
- Vergleichen mit `==`, `!=` (und eventuell `<`, `<=`, `>` und `>=`)
- Dereferenzieren mit `*` oder `->`

Es gibt keine vordefinierten Iteratoren, die nur Forward-Iteratoren sind.

Ein Algorithmus, der Forward-Iteratoren verwendet, ist z. B. `unique()`, der bei gleichen aufeinanderfolgenden Elementen alle bis auf eines entfernt. Dieser Algorithmus `unique()` könnte z. B. wie in Programm 4.63 realisiert sein.

Programm 4.63 – `iterator7.cpp`:
Mögliche Realisierung des STL-Algorithmus unique()

```
#include <vector>
#include <iostream>
using namespace std;
template<typename ForwardIterator>
ForwardIterator
my_unique(ForwardIterator anf, ForwardIterator ende) {
   if (anf == ende)
      return ende;
```

```
   ForwardIterator next = anf;
   while (++next != ende && *anf != *next)
      anf = next;
   if (next == ende)
      return ende;
   while (++next != ende)
      if (*anf != *next)
         *++anf = *++next;
   return ++anf;
}
int main(void) {
  unsigned i;
  vector<int> v;
   for (i=0; i < 20; i++)
      v.push_back(rand() % 6);
   sort(v.begin(), v.end());
   for (i=0; i < v.size(); i++)
      cout << v[i] << ", ";
   cout << endl;
   vector<int>::iterator it, ende = my_unique(v.begin(), v.end());
   for (it = v.begin(); it != ende; it++)
      cout << *it << ", ";
}
```

Programm 4.63 würde z. B. die folgende Ausgabe liefern:

```
0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 2, 3, 3, 4, 4, 4, 4, 4, 5, 5,
0, 1, 2, 3, 4, 5,
```

Forward-Iteratoren unterscheiden sich von Output-Iteratoren dahingehend, dass Output-Iteratoren keine Vergleiche unterstützen, Deshalb kann man nicht in allen Fällen, wo Output-Iteratoren vorgegeben sind, diese durch Forward-Iteratoren ersetzen.

Bidirectional-Iterator – Mehrmaliges Vorwärts-/Rückwärtslesen/-schreiben

Ein Bidirectional-Iterator besitzt zum einen die ganze Funktionalität eines Forward-Iterators, kann aber auch zusätzlich noch mittels des Dekrementierungs-Operators `-` auch rückwärts bewegt werden.
Mit Bidirectional-Iteratoren sind „Mehrweg-Algorithmen“ in beide Richtungen realisierbar. Der `list`-Container stellt z. B. einen Bidirectional-Iterator zur Verfügung.
Als Operationen auf Bidirectional-Iteratoren sind erlaubt:

- Zuweisungsoperator und Kopierkonstruktor
- Inkrementieren bzw. Dekrementieren mit ++ bzw. −− (Post- und Präfix)
- Vergleichen mit ==, != (und eventuell <, <=, > und >=)
- Dereferenzieren mit * oder ->

Ein Algorithmus, der Bidirectional-Iteratoren verwendet, ist z. B. `reverse()`, der die Reihenfolge der Elemente in einem Bereich umkehrt. Dieser Algorithmus `reverse()` könnte z. B. wie in Programm 4.64 realisiert sein. Auf `iterator_traits` wird in Kapitel 4.4.3 auf Seite 236 näher eingegangen.

Programm 4.64 – `iterator8.cpp`:
Mögliche Realisierung des STL-Algorithmus reverse()

```
#include <vector>
#include <iostream>
using namespace std;

template<typename BidirectionalIterator>
void my_reverse(BidirectionalIterator anf, BidirectionalIterator ende) {
   while (anf != ende && anf != --ende) {
      typename iterator_traits<BidirectionalIterator>::value_type tmp = *anf;
      *anf  = *ende;
      *ende = tmp;
      ++anf;
   }
}
int main(void) {
  unsigned i;
  vector<char> v;
   for (i=0; i < 20; i++)
      v.push_back('a'+rand() % 26);
   for (i=0; i < v.size(); i++)
      cout << v[i];
   cout << endl;
   my_reverse(v.begin(), v.end());
   for (i=0; i < v.size(); i++)
      cout << v[i];
   cout << endl;
}
```

Programm 4.64 würde z. B. die folgende Ausgabe liefern:

```
nwlrbbmqbhcdarzowkky
ykkwozradchbqmbbrlwn
```

Random-Access-Iterator – Zeiger-Funktionalität

Ein Random-Access-Iterator besitzt zum einen die ganze Funktionalität eines Bidirectional-Iterators, verfügt aber auch über die Funktionalität eines Zeigers[1], außer dass es keinen `NULL`-Iterator gibt.
Als Operationen auf Random-Access-Iteratoren sind alle Operationen erlaubt, die auch bei Zeigern erlaubt sind:

- Zuweisungsoperator und Kopierkonstruktor
- Inkrementieren bzw. Dekrementieren mit `++` bzw. `--` (Post- und Präfix)
- Vergleichen mit `==`, `!=`, `<`, `<=`, `>` und `>=`
- Dereferenzieren mit `*` oder `->`
- Index-Operator `[]`
- Addition und Subtraktion mit `+`, `+=`, `-` und `-=`

[1] Ein Zeiger ist ein Random-Access-Iterator

Überblick zu den Iterator-Kategorien

Abbildung 4.2 zeigt die Iterator-Kategorien, wie sie hierarchisch angeordnet sind. Hierbei handelt es sich nicht um eine Vererbungs-, sondern um eine Funktionalitätshierarchie, wobei die Funktionalität von innen nach aussen zunimmt. So kann z. B. ein Random-Access-Iterator auch überall dort verwendet werden, wo ein Bidirectional-Iterator verwendet werden kann, und ein Bidirectional-Iterator, wo ein Forward-Iterator verwendet werden kann.

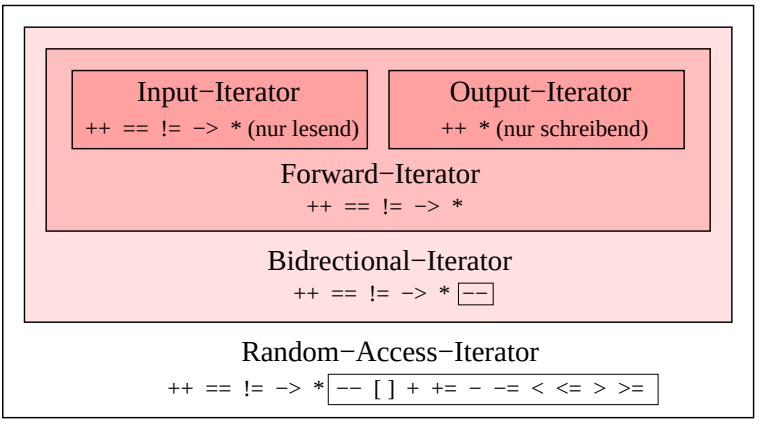


Abbildung 4.2: Funktionalitätshierarchie der Iterator-Kategorien

Die folgende Tabelle zeigt, welche Iterator-Kategorie die Iteratoren in den Containern der Standardbibliothek angehören:

Container	**Iterator-Kategorie**
vector, deque	Random-Access-Iterator (konstant und nicht konstant)
list	Bidirectional-Iterator (konstant und nicht konstant)
set, multiset	Bidirectional-Iterator (nur konstant)
map, multimap	Bidirectional-Iterator (konstant und nicht konstant)

4.4.3 Die Hilfsklasse iterator_traits

In der Hilfsklasse `iterator_traits` werden Typnamen definiert, um abhängig vom entsprechenden Iterator die unterschiedlichen Typen verwenden zu können:

```
template<typename Iterator>
struct iterator_traits {
 typedef typename Iterator::iterator_category iterator_category; // Iter.-Kateg.
 typedef typename Iterator::value_type        value_type;        // Elementtyp
 typedef typename Iterator::difference_type   difference_type;   // Abstandstyp
 typedef typename Iterator::pointer           pointer; // Zeigertyp
                                                       // (Rückgabe für operator->)
 typedef typename Iterator::reference         reference; // Referenztyp
                                                         // (Rückgabe für operator*)
};
```

Iterator-Tags

Um bei eigenen Iteratoren dessen Kategorie zu kënnen, zu er der gehört, wurden so genannte *Iterator-Tags* eingeführt:

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag       : public input_iterator_tag {};
struct bidirectional_iterator_tag : public forward_iterator_tag {};
struct random_access_iterator_tag : public bidirectional_iterator_tag {};
```

Die Klasse `forward_iterator_tag` wird nur von `input_iterator_tag` abgeleitet, und nicht von `output_iterator_tag`. Der Grund dafür ist, dass Algorithmen, die Forward-Iteratoren benutzen, einen Iterator auf das Element hinter dem letzten erwarten. Ein solcher Iterator wird vom Input-Iterator angeboten. Algorithmen dagegen, die Output-Iteratoren verwenden, gehen immer davon aus, dass sie mittels den Dereferenzierungs-Operatoren `*` und `->` auf das jeweilige Element zugreifen können, auf das der Iterator zeigt. Solchen Algorithmen darf also niemals ein ungültiger Iterator hinter das letzte Element zur Verfügung gestellt werden.
Jeder Iteratortyp muss für `iterator_traits<Iterator>::iterator_category` die höchste Iterator-Kategorie angeben, die er unterstützt.
Die Iterator-Tags werden in Kapitel 4.4.4 auf Seite 239 benötigt, wenn wir eigene Iteratoren erstellen.

iterator_traits und Zeiger

Um die Typen der Klasse `iterator_traits` auch bei normalen Zeigern verwenden zu können, bietet die Standardbibliothek entsprechende Spezialisierungen der Templateklasse `iterator_traits` für `T*` und `const T*` an:

```
template<typename T>
struct iterator_traits<T*> {
   typedef random_access_iterator_tag iterator_category;
   typedef T                          value_type;
   typedef ptrdiff_t                  difference_type;
   typedef T*                         pointer;
   typedef T&                         reference;
};
template<typename T>
struct iterator_traits<const T*> {
   typedef random_access_iterator_tag iterator_category;
   typedef T                          value_type;
   typedef ptrdiff_t                  difference_type;
   typedef const T*                   pointer;
   typedef const T&                   reference;
};
```

An diesen Deklarationen ist erkennbar, dass Zeiger zur Kategorie der Random-Access-Iteratoren gehören.
Bei Output-Iteratoren sind `difference_type`, `pointer` und `reference` als `void` definiert.

iterator_traits ermöglicht nur von Iteratoren abhängige Algorithmen

Die Templateklasse `iterator_traits` ermöglicht die Implementierung von Algorithmen, die nur von Iteratoren abhängig sind und somit nicht auf einen bestimmten Containertyp anwendbar sind. Als Beispiel soll der Algorithmus `count()` hier dienen, der zählt, wie oft ein Wert in den Elementen eines bestimmten Bereich vorkommt. Dieser Algorithmus `count()` könnte z. B. wie in Programm 4.65 realisiert sein, in dem dann auch `count()` sowohl auf eine Multiset, eine Liste und einen Vektor angewendet wird..

Programm 4.65 – `iterator9.cpp`:
Mögliche Realisierung des STL-Algorithmus count()

```
#include <set>
#include <list>
#include <vector>
#include <iostream>
using namespace std;
template<typename InputIterator, typename T>
typename iterator_traits<InputIterator>::difference_type
my_count(InputIterator anf, InputIterator ende, const T& wert) {
   typename iterator_traits<InputIterator>::difference_type n = 0;
   for ( ; anf != ende; ++anf)
      if (*anf == wert)
         ++n;
   return n;
}
int main(void) {
  unsigned        i;
  multiset<char>           mset;
  multiset<char>::iterator sit;
  list<char>               liste;
  list<char>::iterator     lit;
  vector<char>    vekt;
   for (i=1; i<=20; i++) {
      char z = 'A' + rand() % 6; // Zufallsschlüssel: A, B, C, D, E, F
      mset.insert( z );
      liste.push_back(z);
      vekt.push_back(z);
   }
   cout << "mset : ";
   for (sit = mset.begin(); sit != mset.end(); sit++)
      cout << *sit;
   cout << " ..B = " << my_count(mset.begin(), mset.end(), 'B') << " mal; "
        << "X = "    << my_count(mset.begin(), mset.end(), 'X') << " mal" <<endl;
   cout << "liste: ";
   for (lit = liste.begin(); lit != liste.end(); lit++)
      cout << *lit;
   cout << " ..B = " << my_count(liste.begin(), liste.end(), 'B') << " mal; "
        << "X = "    << my_count(liste.begin(), liste.end(), 'X') << " mal" <<endl;
   cout << "vekt : ";
   for (i = 0; i < vekt.size(); i++)
       cout << vekt[i];
```

```
   cout << " ..B = " << my_count(vekt.begin(), vekt.end(), 'B') << " mal; "
        << "X = "    << my_count(vekt.begin(), vekt.end(), 'X') << " mal" <<endl;
}
```

Programm 4.65 würde z. B. die folgende Ausgabe liefern:

```
mset : AAABBBBBBCCDDEEEEEFF ..B = 6 mal; X = 0 mal
liste: BEDBFBEADBCBCBFEAAEE ..B = 6 mal; X = 0 mal
vekt : BEDBFBEADBCBCBFEAAEE ..B = 6 mal; X = 0 mal
```

4.4.4 Erstellen eigener Iteratoren

Um die Definition von Iteratoren zu vereinfachen, stellt die STL die Templateklasse `iterator` zur Verfügung, die als Basisklasse dient und die fünf Typen für `iterator_traits` bereit stellt:

```
template<typename Category, typename T, typename Distance = ptrdiff_t,
         typename Pointer = T*, typename Reference = T&>
struct iterator {
   typedef Category  iterator_category;
   typedef T         value_type;
   typedef Distance  difference_type;
   typedef Pointer   pointer;
   typedef Reference reference;
};
```

In Programm 4.66 wird eine eigene Klasse `RingListe` mit eigenen Iteratoren erstellt.

Programm 4.66 – `iterator10.cpp`:
Eigene Klasse RingListe mit selbst erstellten Iteratoren

```
#include <iostream>
#include <list>
#include <string>
using namespace std;
template<class T>
class RingListe {
     friend class iterator;
  public:
     class iterator {
          typedef std::bidirectional_iterator_tag iterator_category;
          typedef T                               value_type;
          typedef std::ptrdiff_t                  difference_type;
          typedef T*                              pointer;
          typedef T&                              reference;
       public:
          iterator(list<T>& l, const typename list<T>::iterator& i)
                   : it(i), r(&l) { }
          bool operator==(const iterator& x) const { return it == x.it;    }
          bool operator!=(const iterator& x) const { return !(*this == x); }
          reference operator*() const { return *it; }
          iterator& operator++() {
```

```
            if (++it == r->end())
               it = r->begin();
            return *this;
         }
         iterator operator++(int) {
            iterator tmp = *this;
            ++*this;
            return tmp;
         }
         iterator& operator--() {
            if (it == r->begin())
               it = r->end();
            --it;
            return *this;
         }
         iterator operator--(int) {
            iterator tmp = *this;
            --*this;
            return tmp;
         }
         iterator insert(const T& x) { return iterator(*r, r->insert(it, x)); }
         iterator erase() {
            if ( (it = r->erase(it)) == r->end())
               it = r->begin();
            return *this;
         }
       private:
         typename list<T>::iterator it;
         list<T>*                   r;
     };
     void     push_back(const T& x) { lst.push_back(x);                  }
     iterator begin()               { return iterator(lst, lst.begin()); }
     int      size()                { return lst.size();                 }
     bool     empty()               { return lst.empty();                }
  private:
     list<T> lst;
};
int main(void)
{
   int             i;
   RingListe<char> r1Liste;
   for (i=0; i<5; i++)
      r1Liste.push_back('A'+i);
   RingListe<char>::iterator it = r1Liste.begin();
   it++; it++;
   it.insert('X');
   it = r1Liste.begin();
   for (i = 0; i < r1Liste.size() * 3; i++) // 3 mal durch die Ringlistse laufen
      cout << *it++;
   cout << endl;
```

```
   RingListe<int> r2Liste;
   for (i=1; i<=20; i++)
      r2Liste.push_back(i);
   RingListe<int>::iterator it2 = r2Liste.begin();
   while (!r2Liste.empty()) {
      for (i=1; i<5; i++)
         ++it2;
      cout << *it2 << ",";
      it2 = it2.erase();
   }
}
```

Programm 4.66 liefert die folgende Ausgabe:

```
ABXCDEABXCDEABXCDE
5,10,15,20,6,12,18,4,13,1,9,19,11,3,17,16,2,8,14,7,
```

4.4.5 Iterator-Funktionen

Während für Random-Access-Iteratoren sowohl Positionierungen über mehrere Element hinweg als auch Abstandsberechnungen zwischen zwei Iteratoren möglich sind, werden solche Operationen bei den anderen Iterator-Kategorien nicht unterstützt. Um nun Algorithmen, die solche Operationen benötigen, allgemein definieren zu können, so dass diese auf alle Iterator-Kategorien anwendbar sind, werden über die Headerdatei `<iterator>` zwei Hilfsfunktionen zur Verfügung gestellt:

- `distance()`

  ```
  template <class InputIterator>
  difference_type distance(InputIterator first, InputIterator last);
  ```

  Die Funktion `distance()` liefert die Anzahl der Elemente, die im Bereich von `first` bis `last` liegen, also wie oft `first` inkrementiert werden muss, bis es gleich `last` ist. Es werden hier abhängig von der jeweiligen Iterator-Kategorie zwei überladene Funktionen `distance()` angeboten. Mögliche Realisierungen dazu sind nachfolgend gezeigt:

  - für *Random-Access-Iteratoren*:

    ```
    template<typename RandomAccessIterator>
    inline typename iterator_traits<RandomAccessIterator>::difference_type
    distance(RandomAccessIterator first, RandomAccessIterator last,
             random_access_iterator_tag) {      return last - first; }
    ```

  - für die anderen Iterator-Kategorien:

    ```
    template<typename InputIterator>
    inline typename iterator_traits<InputIterator>::difference_type
    distance(InputIterator first, InputIterator last, input_iterator_tag) {
       typename iterator_traits<InputIterator>::difference_type n = 0;
       while (first != last) {
          ++first; ++n;
       }
       return n;
    }
    ```

- `advance()`

```
template <class InputIterator, class Distance>
void advance(InputIterator& it, Distance n);
```

`advance(it, n)` inkrementiert den Iterator `it` um den Wert `n`. Bei `n>0` ist dieser Aufruf äquivalent zu `n`-mal `it++` und bei `n<0` ist er äquivalent zu `n`-mal `it--`. Bei `n==0` hat dieser Aufruf keine Auswirkung auf den Iterator. Da nur Random-Access- und Bidirectional-Iteratoren rückwärts laufen können, sind negative Werte für `n` ausschließlich für diese beiden Iterator-Kategorien erlaubt.
Es werden hier abhängig von der jeweiligen Iterator-Kategorie drei überladene Funktionen `advance()` angeboten. Mögliche Realisierungen dazu sind nachfolgend gezeigt:

- für *Input- und Forward-Iteratoren*:

```
template<typename InputIterator, typename Distance> inline
void advance(InputIterator& it, Distance n, input_iterator_tag) {
   while (n--)
      ++it;
}
```

- für *Birectional-Iteratoren*:

```
template<typename BidirectionalIterator, typename Distance> inline
void advance(BidirectionalIterator& it, Distance n,
             bidirectional_iterator_tag) {
   if (n > 0)
      while (n--) ++it;
   else
      while (n++) --it;
}
```

- für *Random-Access-Iteratoren*:

```
template<typename RandomAccessIterator, typename Distance> inline
void advance(RandomAccessIterator& it, Distance n,
             random_access_iterator_tag) {
   it += n;
}
```

Die nach aussen sichtbare Bibliotheksfunktion `advance()` ruft eine dieser drei Hilfsfunktionen auf:

```
template<typename InputIterator, typename Distance> inline
void advance(InputIterator& it, Distance n) {
   advance(it, n, iterator_category(it));
}
```

Programm 4.67 ist ein Demonstrationsprogramm zu den beiden Iterator-Funktionen `advance()` und `distance()`. Es zieht zweimal 6 Lottozahlen aus 49 Zahlen.

Programm 4.67 – `iterator11.cpp`:
Demoprogramm zu den beiden Iterator-Funktionen advance() und distance()

```
#include <cstdlib>
#include <ctime>
#include <vector>
#include <list>
```

```
#include <iterator>
#include <iostream>
using namespace std;
int main(void) {
  unsigned i;
  srand(time(0));
   vector<int> v(50);
   vector<int>::iterator it;
   for (i=0; i < v.size(); i++)
      v[i] = i;
   for (i=1; i<=6; i++) {
      do {
         it = v.begin();
         advance(it, rand()%50);
      } while (*it == 0);
      cout << *it << ", ";
      *it = 0;
   }
   cout << endl;
   cout << "In v sind " << distance(v.begin(), v.end()) << " Zahlen" << endl;
   cout << "Von einschlesslich zuletzt gezogener Zahl bis Ende noch "
        << distance(it, v.end()) << endl;
   list<int> l;
   list<int>::iterator lit;
   for (i=0; i < 50; i++)
      l.push_back(i);
   for (i=1; i<=6; i++) {
      do {
         lit = l.begin();
         advance(lit, rand()%50);
      } while (*lit == 0);
      cout << *lit << ", ";
      *lit = 0;
   }
   cout << endl;
   cout << "In l sind " << distance(l.begin(), l.end()) << " Zahlen" << endl;
   cout << "Von einschlesslich zuletzt gezogener Zahl bis Ende noch "
        << distance(lit, l.end()) << endl;
}
```

Programm 4.67 liefert z. B. folgende Ausgabe:

```
17, 46, 45, 32, 48, 7,
In v sind 50 Zahlen
Von einschlesslich zuletzt gezogener Zahl bis Ende noch 43
40, 3, 11, 8, 25, 17,
In l sind 50 Zahlen
Von einschlesslich zuletzt gezogener Zahl bis Ende noch 33
```

4.4.6 Vordefinierte Iteratoren

Die STL stellt einige sehr nützliche vordefinierten Iteratoren zur Verfügung, die nachfolgend kurz vorgestellt werden.

Reverse-Iteratoren

Reverse-Iteratoren kehren die Bedeutung der Operatoren `++` und `--` um, was bedeutet, dass ein Reverse-Iterator bei `++` rückwärts und bei `--` vorwärts im entsprechenden Container weiterpositioniert wird. Um Reverse-Iteratoren benutzen zu können, muss der entsprechende Container *Bidirectional-Iteratoren* bzw. *Random-Access-Iteratoren* unterstützen.
Reverse-Iteratoren bieten die folgenden Methoden an:

- `rbegin()` zeigt auf das letzte Element des Containers.
- `rend()` zeigt vor das erste Element des Containers.
- `base()` liefert einen normalen Iterator auf das Element, auf das der Reverse-Iterator aktuell zeigt.

Um Reverse-Iteratoren zu definieren, stehen folgende Schlüsselwörter zur Verfügung:

```
<ContainerName>::reverse_iterator       // erlaubt Änderung der Elemente
<ContainerName>::const_reverse_iterator // erlaubt keine Änderung der Elemente
```

Programm 4.68 ist ein Demonstrationsprogramm zu Reverse-Iteratoren. Es liest die Datei `revit.txt` zeilenweise in einen Vektor ein, bevor es diesen Vektor mittels eines Reverse-Iterators wieder rückwärts ausgibt. Zudem benutzt es die Methode `base()`, um sich über den Reverse-Iterator `rueck` einen normalen Iterator auf das viertletzte Element des Vektors geben zu lassen. Über diesen normalen Iterator gibt es dann die letzten vier Elemente (Zeilen) vorwärts aus.

Programm 4.68 – `iterator12.cpp`:
Demoprogramm zu Reverse-Iteratoren

```
#include <fstream>
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
using namespace std;
int main(void) {
   int nr = 0;
   ifstream datei("revit.txt");
   string zeile;
   vector<string> v;
   //.... Datei revit.txt zeilenweise in Vektor v einlesen
   while (getline(datei, zeile)) {
      v.push_back(zeile);
      nr++;
   }
   //.... Inhalt von Vektor v rückwärts ausgeben
   vector<string>::reverse_iterator rueck;
   for (rueck = v.rbegin(); rueck != v.rend(); rueck++)
      cout << setw(2) << nr-- << "  " << *rueck << endl;
```

```
   //.... letzten 4 Zeilen vorwärts ausgeben
   cout << "------------------------- Die letzten 4 Zeilen der Datei" << endl;
   rueck = v.rbegin();
   vector<string>::iterator vor = rueck.base()-4;
   for ( ; vor != v.end(); vor++)
      cout << *vor << endl;
}
```

Hat die Datei `revit.txt` folgenden Inhalt:

```
Das ist Zeile eins
und hier ist Zeile 2
Dritte Zeile und
hier die vierte
Number five and
number six
Hier nun die 7. Zeile
und auch gleich noch die Achte
Zeile 9
und Zeile 10
```

so liefert Programm 4.68 die folgende Ausgabe:

```
10  und Zeile 10
 9  Zeile 9
 8  und auch gleich noch die Achte
 7  Hier nun die 7. Zeile
 6  number six
 5  Number five and
 4  hier die vierte
 3  Dritte Zeile und
 2  und hier ist Zeile 2
 1  Das ist Zeile eins
------------------------- Die letzten 4 Zeilen der Datei
Hier nun die 7. Zeile
und auch gleich noch die Achte
Zeile 9
und Zeile 10
```

Inserter-Iteratoren

Mittels Iteratoren kann man nur bereits in einem Container vorhandene Elemente überschreiben, aber nicht neue Elemente in einem Container einfügen oder aber vorhandene Elemente daraus entfernen. Für das Einfügen von Elementen wurden nun spezielle Iteratoren zur Verfügung gestellt: die so genannten *Inserter*. Dies sind Adapter-Klassen, die die Funktionalität anderer Klassen ändern bzw. erweitern. So bieten sie z. B. einen eigenen überladenen Zuweisungsoperator = an, der nicht das entsprechende Element überschreibt, sondern mittels Aufruf einer geeigneten Methode des entsprechenden Containers an der entsprechenden Stelle einfügt.
Nachfolgend werden nun solche *Inserter* vorgestellt:

Back-Inserter – Einfügen am Ende des Containers mit push_back()

Um einen *Back-Inserter* für einen Container zu erzeugen, steht die Funktion `back_inserter()` zur Verfügung, wie z. B.:

```
vector<char> v;
back_insert_iterator<vector<char> > bc; // Back-Inserter-It. für char-Vektoren
bc = back_inserter(v); // erzeugt Back-Inserter für Vektor v
```

In der zweiten Zeile muss zwischen den beiden spitzen Klammern mindestens ein Leerzeichen angegeben werden, damit der Compiler hier nicht fälschlicherweise den Operator >> annimmt. Es gibt nun zwei Möglichkeiten, um über den *Back-Inserter* `bc` Elemente im Vektor `v` einzufügen:

- Zuweisen der einzufügenden Elemente an den *Back-Inserter* `bc`, wie z. B.:

```
bc = 'x'; // fügt das Zeichen 'x' am Ende des Vektors an
bc = 'y'; // fügt das Zeichen 'y' am Ende des Vektors an
......
```

- Verwenden eines Algorithmus, wie z. B. `copy()`:

```
string vokale("aeiou");
copy(vokale.begin(), vokale.end(), bc); // fügt a,e,i,o und u am Vektorende an
```

  In einem solchen Fall hätte man sich das Anlegen der eigenen *Back-Inserter*-Variablen `bc` auch sparen können:

```
string vokale("aeiou");
copy(vokale.begin(), vokale.end(), back_inserter(v));
```

Es gilt hier: *Ein Container, der einen Back-Inserter verwenden will, muss eine* `public`*-Methode* `push_back()` *besitzen, was z. B. für die sequenziellen Container* `vector`, `list` *und* `deque` *zutrifft.*

Front-Inserter – Einfügen am Anfang des Containers mit push_front()

Um einen *Front-Inserter* für einen Container zu erzeugen, steht die Funktion `front_inserter()` zur Verfügung, wie z. B.:

```
list<char> l; // Verkettete Liste, da vector keine Methode push_front() hat
front_insert_iterator<list<char> > fc; // Front-Inserter-It. für char-Listen
fc = front_inserter(v); // erzeugt Front-Inserter für Liste l
```

In der zweiten Zeile muss wieder zwischen den beiden spitzen Klammern mindestens ein Leerzeichen angegeben werden, damit der Compiler hier nicht fälschlicherweise den Operator >> annimmt. Es gibt nun wieder mehrere Möglichkeiten, um über den *Front-Inserter* `fc` Elemente in der Liste `l` am Anfang einzufügen:

```
//.....................Zuweisen der einzufügenden Elemente an Front-Inserter
fc = 'x'; // fügt das Zeichen 'x' am Anfang der Liste an
fc = 'y'; // fügt das Zeichen 'y' am Anfang der Liste an
......
//.............................Verwenden eines Algorithmus, wie z.B. copy()
string vokale("aeiou");
copy(vokale.begin(), vokale.end(), fc); // fügt a,e,i,o,u am Listenanf. ein
//... oder:
copy(vokale.begin(), vokale.end(), front_inserter(v));
```

Es gilt hier: *Ein Container, der einen Front-Inserter verwenden will, muss eine* `public`*-Methode* `push_front()` *besitzen, was z. B. für die sequenziellen Container* `list` *und* `deque` *zutrifft, aber nicht für* `vector`.

Inserter – Einfügen in der Mitte eines Containers mit insert()

Um einen allgemeinen *Inserter* für einen Container zu erzeugen, steht die Funktion `inserter()` zur Verfügung, wie z. B.:

```
vector<char> v;
insert_iterator<vector<char> > mc; // Inserter-Iterat. für char-Vektoren
mc = inserter(v, v.begin()+3); // erzeugt Inserter an Position 3 für Vektor v
```

In der zweiten Zeile muss wieder zwischen den beiden spitzen Klammern mindestens ein Leerzeichen angegeben werden, damit der Compiler hier nicht fälschlicherweise den Operator >> annimmt. Es gibt nun wieder mehrere Möglichkeiten, um über den *Inserter* `mc` Elemente im Vektor `v` einzufügen:

```
//.........................Zuweisen der einzufügenden Elemente an Inserter
mc = 'x'; // fügt das Zeichen 'x' an Pos. 3 im Vektor ein
mc = 'y'; // fügt das Zeichen 'y' an Pos. 3 im Vektor ein
......
//.............................Verwenden eines Algorithmus, wie z.B. copy()
string vokale("aeiou");
copy(vokale.begin(), vokale.end(), mc); // fügt a,e,i,o,u an Pos. 3 ein
//... oder:
copy(vokale.begin(), vokale.end(), inserter(v));
```

Beim *Inserter* wird also vor der festgelegten Position eingefügt, wodurch sich alle Elemente verschieben, die hinter dem eingefügten Element liegen. Der *Inserter* wird dann anschließend automatisch auf das nächste Element weiter bewegt.
Es gilt hier: *Ein Container, der einen allgemeinen Inserter verwenden will, muss eine* `public`*-Methode* `insert()` *besitzen, was z. B. für die sequenziellen Container* `vector`, `list` *und* `deque` *zutrifft.*

Programm 4.69 – `iterator13.cpp`*:*
Demoprogramm zu Inserter-Iteratoren

```
#include <vector>
#include <list>
#include <iostream>
using namespace std;

template <typename T>
void ausgabe(T container) {
   typename T::iterator it;
   for (it = container.begin(); it != container.end(); it++)
      cout << *it;
   cout << " (" << container.size() << ")" << endl;
}
int main(void) {
   string text("--gnafnA---");
   list<char>  l;
```

```
   back_insert_iterator<list<char> >  bl = back_inserter(l);
   bl = 'x'; //..... Am Ende der Liste einfügen
   bl = 'y';
   bl = 'z';
   //..... Am Anfang der Liste einfügen
   copy(text.begin(), text.end(), front_inserter(l));
   ausgabe(l);
   vector<char> v(10);
   for (unsigned i=0; i < 10; i++)
      v[i] = '.';
   ausgabe(v);
   //..... Ab Position 3 einfügen
   insert_iterator<vector<char> > mc = inserter(v, v.begin()+3);
   mc = 'x';
   mc = 'y';
   ausgabe(v);
   //..... Ab Position 8 vom Ende her einfügen
   copy(text.begin(), text.end(), inserter(v, v.end()-8));
   ausgabe(v);
}
```

Programm 4.69 liefert die folgende Ausgabe:

```
---Anfang--xyz (14)
.......... (10)
...xy....... (12)
...x--gnafnA---y....... (23)
```

Stream-Iteratoren

Stream-Iteratoren ermöglichen das Automatisieren von bestimmten Ein- und Ausgaben.

Ostream-Iteratoren

werden auf Ausgabe-Streamobjekte angewendet und können, wie in Programm 4.70 gezeigt, eingesetzt werden.

Programm 4.70 – `iterator14.cpp`:
Demoprogramm zu Ostream-Iteratoren

```
#include <fstream>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
using namespace std;

int main(void) {
   string str("Mittelstreifen");
   ostream_iterator<char> outChar(cout, "--");
   // String mit zwei Querstrichen zwischen den einzelnen Zeichen ausgeben
   copy(str.begin(), str.end(), outChar);
   cout << endl;
```

```
   vector<int> v;
   for (unsigned i=1; i<=10; i++)
      v.push_back(i*i);
   ostream_iterator<int> outIntVect(cout, ", ");
   // ganzen Vektor mit Kommas zwischen den Werten ausgeben
   copy(v.begin(), v.end(), outIntVect);
   *outIntVect++ = v[2]; // 2. Element von Vektor nochmals ausgeben
   *outIntVect++ = 4711; // 4711 ausgeben
   cout << endl;
}
```

Programm 4.70 liefert die folgende Ausgabe:

```
M--i--t--t--e--l--s--t--r--e--i--f--e--n--
1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 9, 4711,
```

Programm 4.71 ist ein weiteres Beispiel zu Ostream-Iteratoren. Es erwartet auf der Kommandozeile den Namen einer Datei, deren Inhalt es dann Wort für Wort mittels `copy()` und einem Back-Inserter in einem `string`-Vektor abspeichert. Anschließend sortiert dieses Programm diesen `string`-Vektor mit `sort()` und gibt ihn dann mittels `copy()` von Anfang bis Ende und dem Iterator `out` auf der Standardausgabe (hier der Bildschirm) aus.

Programm 4.71 – `iterator15.cpp`:
Wortweise Ausgabe einer Datei mittels Stream-Iteratoren

```
#include <fstream>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
using namespace std;
int main(int argc, char *argv[]) {
   if (argc != 2) {
      cerr << "usage: " << argv[0] << " datei" << endl; exit(1);
   }
   vector<string> v; // string-Vektor
   ifstream datei(argv[1]); // Input-Stream auf Datei argv[1]
   //..................................... Iteratoren
   istream_iterator<string> begin(datei), end; // auf Dateianfang und für Ende
   ostream_iterator<string> out(cout, "\n");   // auf cout
   *out++ = "---Anfang------- " + string(argv[1]) + "------------------";
   copy(begin, end, back_inserter(v)); // Wort für Wort am Ende anfügen
   sort(v.begin(), v.end());      // Vektor sortieren
   copy(v.begin(), v.end(), out); // und dann ausgeben
   *out++ = "---Ende--------- " + string(argv[1]) + "------------------";
}
```

Hat man z. B. die folgende Datei `text.txt`:

```
dieser Text wird
Wort für Wort ausgegeben.
Jedes Wort in einer Zeile
```

und man ruft **./iterator15 text.txt** auf, so liefert es die folgende Ausgabe:

```
---Anfang------- text.txt------------------
Jedes
Text
Wort
Wort
Wort
Zeile
ausgegeben.
dieser
einer
für
in
wird
---Ende--------- text.txt------------------
```

Istream-Iteratoren

sind das Gegenstück zu Ostream-Iteratoren. Mit Istream-Iteratoren kann man Elemente direkt aus Eingabe-Streamobjekte lesen, wie es in Programm 4.72 gezeigt ist.

Programm 4.72 – `iterator16.cpp`:
Demoprogramm zu Istream-Iteratoren

```
#include <algorithm>
#include <fstream>
#include <iostream>
#include <iterator>
using namespace std;
bool groesser100(int z) { return z > 100; }
int main(void) {
   //.... Schreiben von ganzen Zahlen in Datei 'zahlen.txt'
   ofstream ausDatei("zahlen.txt");
   ausDatei << "2 20 150 30 1000 200 60 70 2500";
   ausDatei.close();
   //.... 1. Form der Ausgabe mit remove_copy_if()
   ifstream inDatei("zahlen.txt");
   istream_iterator<int> einAnfang(inDatei); // Istream-Iterator für Dateianf.
   istream_iterator<int> einEnde;            // Istream-Iterator für Dateiende
   ostream_iterator<int> ausg(cout, ", "); // Ostream-Iterator
      // Lesen der Zahlen aus 'zahlen.txt' und die ausgeben, die <= 100 sind
      // Zahlen, die >= 100 werden nicht mittels cout ausgegeben
   remove_copy_if(einAnfang, einEnde, ausg, groesser100);
   inDatei.close(); cout << endl;
   //.... 2. Form der Ausgabe mit expliziten Durchlaufen
   ifstream inDatei2("zahlen.txt");
   istream_iterator<int> it(inDatei2); // Istream-Iterator für Dateianf.
   istream_iterator<int> itEnde;       // Istream-Iterator für Dateiende
   for ( ; it != itEnde; it++)
      if (*it <= 100)
         cout << *it << ", ";
}
```

Programm 4.72 liefert die folgende Ausgabe:

```
2, 20, 30, 60, 70,
2, 20, 30, 60, 70,
```

Für Istream-Iteratoren gibt es zwei Konstruktoren (siehe auch Programm 4.72):

```
istream_iterator<typ> anfang(inStream); // Iterator auf den Anfang des
                                        // entsprechenden Eingabe-Streamobjekts
istream_iterator<typ> ende; // Iterator auf Ende eines Eingabe-Streamobjekts (EOF)
```

Auch wird aus Programm 4.72 ersichtlich, dass zum Weiterpositionieren des Iterators der Operator `operator++()` und zum Zugriff auf das Element, auf das der Iterator gerade zeigt, der Operator `operator*()` überladen ist.

Spezielle Iteratoren ostreambuf_iterator und istreambuf_iterator

Es ist zwar möglich, Iteratoren zu `istream_iterator<char>` and `ostream_iterator<char>` anzulegen. Diese Iteratoren überlesen jedoch *white spaces* (Leer-, Tab- und Neuezeilezeichen). Ist dies nicht gewünscht, muss man die speziellen Iteratoren `ostreambuf_iterator` und `ostreambuf_iterator` verwenden, wie es in Programm 4.73 gezeigt ist.

Programm 4.73 – `iterator17.cpp`:
Demoprogramm zu den speziellen Iteratoren ostreambuf_iterator und ostreambuf_iterator

```
#include <algorithm>
#include <fstream>
#include <iostream>
#include <iterator>
using namespace std;
int main(void) {
   cout << "---white spaces (Leer-, Tab-, Neuezeilezeichen) überlesen" << endl;
   ifstream ifs1("text2.txt");
   istream_iterator<char> is1(ifs1), end;
   ostream_iterator<char> os1(cout);
   while (is1 != end)
      *os1++ = *is1++;
   cout << endl << endl << "---white spaces nicht mehr überlesen" << endl;
   ifstream ifs2("text2.txt");
   istreambuf_iterator<char> is2(ifs2), end2;
   ostreambuf_iterator<char> os2(cout);
   while (is2 != end2)
      *os2++ = *is2++;
}
```

Hat die Datei `text2.txt` folgenden Inhalt:

```
Das ist ein
einfacher Text
mit drei Zeilen.
```

so liefert Programm 4.73 die folgende Ausgabe:

```
---white spaces (Leer-, Tab-, Neuezeilezeichen) überlesen
DasisteineinfacherTextmitdreiZeilen.

---white spaces nicht mehr überlesen
Das ist ein
einfacher Text
mit drei Zeilen.
```

Der spezielle Iterator raw_storage_iterator

In C++ führt der Operator `new` implizit zwei Schritte aus:

1. Zunächst alloziert er Speicher für ein Objekt,
2. und anschließend ruft er dann automatisch den entsprechenden Konstruktor auf, um ein Objekt in dem allozierten Speicherplatz anzulegen.

In manchen Situationen ist es aber notwendig, diese beiden Operationen getrennt voneinander ausführen zu lassen. Ist `it` ein Iterator, der auf einen reservierten, aber noch nicht initialisierten Speicherplatz zeigt, dann kann man sich in diesem Speicherplatz mit `raw_storage_iterator` ein Objekt anlegen lassen, wobei hierzu dann auch der entsprechende Konstruktor zu `T` aufgerufen wird:

```
raw_storage_iterator<ForwardIterator, T> // definiert in Headerdatei <memory>
```

Programm 4.74 – `iterator18.cpp`:
Demoprogramm zum speziellen Iterator raw_storage_iterator

```
#include <iostream>
#include <iterator>
#include <algorithm>
#include <memory>
using namespace std;
class Zahl {
  public:
     Zahl(int x) : wert(x) {}
     int get(void) { return wert; }
  private:
     int wert;
};
int main(void) {
   unsigned i;
   int positiv[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   const unsigned ANZ = sizeof(positiv) / sizeof(*positiv);
   Zahl *negativ = reinterpret_cast<Zahl*> (new char[(ANZ+3) * sizeof(Zahl)]);
   // transform() kopiert das Array 'positiv' an den Anfang des allozierten,
   // aber noch nicht initialisierten Speicherplatz 'negativ',
   // wobei alle Werte mit Aufruf von negate() negiert werden
   transform(positiv, positiv+ANZ,
             raw_storage_iterator<Zahl*, Zahl>(negativ),
             negate<int>());
   for (i=0; i < ANZ; i++)
      cout << negativ[i].get() << ", ";
   cout << endl;
```

```
   raw_storage_iterator<Zahl*, Zahl> it(negativ+ANZ);
   *it++ = -20; // Alternative Zugriffe über den raw_storage_iterator
   *it++ = -200;
   *it++ = -2000;
   for (i=0; i < ANZ+3; i++)
      cout << negativ[i].get() << ", ";
   cout << endl;
   delete [] reinterpret_cast<char*>(negativ); // nicht: delete [] negativ
}
```

Programm 4.74 liefert die folgende Ausgabe:

```
-1, -2, -3, -4, -5, -6, -7, -8, -9, -10,
-1, -2, -3, -4, -5, -6, -7, -8, -9, -10, -20, -200, -2000,
```

4.5 Funktionsobjekte

Für Funktionsobjekte gilt Folgendes:

- Ein Funktionsobjekt ist ein Objekt, für das der Funktionsoperator `operator()` definiert ist.
- Funktionsobjekte sind somit Zeiger auf Funktionen und Objekte von Klassen, bei denen `operator()` überladen ist.
- Es hat sich eingebürgert, dass man Klassen, die den `operator()` überladen, einfach als „Funktionsobjekte“ bezeichnet.

So handelt es sich z. B. bei folgender Struktur (Klasse) um ein Funktionsobjekt:

```
template<class T>
struct kleiner {
   bool operator() (const T& x, const T& y) const { return x < y; }
}
```

Solche elementare Funktionsobjekte stellt die Standardbibliothek in der Headerdatei `<functional>` zur Verfügung, wie z. B.:

```
template <class T> //... binary_function wird nachfolgend vorgestellt
struct less : public binary_function<T,T,bool> {
  bool operator()(const T& x, const T& y) const { return x < y; }
};
```

4.5.1 Basisklassen für Funktionsobjekte

Man unterscheidet zwischen:

- einstelligen Funktionsobjekten (`operator()` hat ein Argument):

```
result_type operator() (argument_type)
```

- zweistelligen Funktionsobjekten (`operator()` hat zwei Argumente):

```
result_type operator() (first_argument_type, second_argument_type)
```

Da einige Funktionen der Standardbibliothek die Typnamen `result_type`, `argument_type`, `first_argument_type` und/oder `second_argument_type` benutzen, bietet die Standardbibliothek zur Vereinfachung die Basisklassen `unary_function` und `binary_function` an:

```
template <class Arg, class Result>
struct unary_function {
  typedef Arg    argument_type;
  typedef Result result_type;
};
template <class Arg1, class Arg2, class Result>
struct binary_function {
  typedef Arg1   first_argument_type;
  typedef Arg2   second_argument_type;
  typedef Result result_type;
};
```

Da `less` ein zweistelliges Funktionsobjekt ist, wird es von `binary_function` abgeleitet, so dass für `less` die Typen `first_argument_type`, `second_argument_type` und `result_type` mit den Werten `T`, `T` und `bool` zur Verfügung stehen:

```
template <class T>
struct less : public binary_function<T,T,bool> {
  bool operator() (const T& x, const T& y) const { return x < y; }
};
```

Man kann natürlich auch selbstdefinierte Funktionsobjekte von `unary_function` bzw. `binary_function` ableiten, wie es in Programm 4.75 gezeigt ist.

Programm 4.75 – `fktsobj1.cpp`:
Ableiten eines eigenen Funktionsobjekts von unary_function

```
#include <vector>
#include <iostream>
using namespace std;
template<typename InputIter, typename Predicate>
inline InputIter my_find_if(InputIter first, InputIter last, Predicate pred) {
   while (first != last && !pred(*first))
      ++first;
   return first;
}
template<typename T>
class zeichBereich : public unary_function<T, bool> {
  public:
     zeichBereich(const T& from, const T& to) : von(from), bis(to) {}
     bool operator() (const T& z) const { return z >= von && z <= bis; }
  private:
     char  von, bis;
};
int main(void) {
  unsigned i;
  vector<char> v;
```

```
   for (i=0; i < 10; i++)
      v.push_back('A'+rand()%26);
   for (i=0; i < v.size(); i++)
      cout << v[i];
   cout << endl;
   vector<char>::iterator it = v.begin();
   while ( (it = my_find_if(it, v.end(), zeichBereich<char>('A', 'M'))) != v.end())
      cout << *it++;
   cout << endl;
}
```

Programm 4.75 liefert z. B. die folgende Ausgabe:

```
NWLRBBMQBH
LBBMBH
```

4.5.2 Vordefinierte arithmetische Funktionsobjekte

Tabelle 4.1 zeigt die in der Standardbibliothek vordefinierten arithmetischen Funktionsobjekte.

Tabelle 4.1: Vordefinierte arithmetische Funktionsobjekte

Funktionsobjekt	Operation	Rückgabetyp	Argumentzahl
`plus`	x + y	T	2
`minus`	x - y	T	2
`multiplies`	x * y	T	2
`divides`	x / y	T	2
`modulus`	x % y	T	2
`negate`	- x	T	1

Programm 4.76 – `fktsobj2.cpp`:
Demoprogramm zu den vordefinierten arithmetischen Funktionsobjekten

```
#include <string>
#include <functional>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;

int main(void) {
   int x = 23, y = 5;
   plus<int>       pl;
   minus<int>      mi;
   multiplies<int> mu;
   divides<int>    di;
   modulus<int>    mo;
   negate<int>     ne;
```

```
   cout << "plus("       << x << "," << y << ") = " << pl(x, y) << endl;
   cout << "minus("      << x << "," << y << ") = " << mi(x, y) << endl;
   cout << "multiplies(" << x << "," << y << ") = " << mu(x, y) << endl;
   cout << "divides("    << x << "," << y << ") = " << di(x, y) << endl;
   cout << "modulus("    << x << "," << y << ") = " << mo(x, y) << endl;
   cout << "negate("                 << y << ") = " << ne(y)    << endl;
   cout << ".... ohne Hilfs-Funktionsobjekte" << endl;
   cout << "plus("       <<x << "," <<y <<") = " << plus<int>()(x, y)       <<endl;
   cout << "minus("      <<x << "," <<y <<") = " << minus<int>()(x, y)      <<endl;
   cout << "multiplies(" <<x << "," <<y <<") = " << multiplies<int>()(x, y) <<endl;
   cout << "divides("    <<x << "," <<y <<") = " << divides<int>()(x, y)    <<endl;
   cout << "modulus("    <<x << "," <<y <<") = " << modulus<int>()(x, y)    <<endl;
   cout << "negate("               <<y <<") = " << negate<int>()(y)        <<endl;
   int a1[] = { 55, 30, 30, 40 },
       a2[] = { 10, 20, 15, 23 };
   transform(a1, a1+4, a2, ostream_iterator<int>(cout, ", "), minus<int>());
   cout << endl;
   transform(a1, a1+4, a1, negate<int>());
   for (int i=0; i < 4; i++)
      cout << a1[i] << ", ";
   cout << endl;
   string s1(".... und "),
          s2("Tschuess");
   plus<string> stringadd;
   cout << stringadd(s1, s2) << endl;
}
```

Programm 4.76 liefert die folgende Ausgabe:

```
plus(23,5) = 28
minus(23,5) = 18
multiplies(23,5) = 115
divides(23,5) = 4
modulus(23,5) = 3
negate(5) = -5
.... ohne Hilfs-Funktionsobjekte
plus(23,5) = 28
minus(23,5) = 18
multiplies(23,5) = 115
divides(23,5) = 4
modulus(23,5) = 3
negate(5) = -5
45, 10, 15, 17,
-55, -30, -30, -40,
.... und Tschuess
```

Programm 4.77 zeigt, dass Funktionsobjekte immer den entsprechenden Operator der jeweiligen Klasse aufrufen, der natürlich dann auch definiert sein muss. In Programm 4.77 wird z. B. bei `plus()` der `operator+()` von der Klasse `Zeit` aufgerufen.

Programm 4.77 – `fktsobj3.cpp`:
Zeitaddierer mit dem Funktionsobjekt plus()

```
#include <vector>
#include <functional>
#include <numeric>
#include <iterator>
#include <iomanip>
#include <iostream>
using namespace std;

class Zeit {
     friend ostream& operator<< (ostream& os, const Zeit &z) {
        return os << setw(2) << z.tage << " Tage, "
                  << setfill('0') << setw(2) << z.std << ":"
                  << setfill('0') << setw(2) << z.min << ":"
                  << setfill('0') << setw(2) << z.sek << setfill(' ') << endl;
     }
  public:
     Zeit(unsigned h, unsigned m, unsigned s) : tage(0), std(h), min(m), sek(s) {}
     Zeit(const Zeit &z) : tage(z.tage), std(z.std), min(z.min), sek(z.sek) {}
     Zeit operator+ (const Zeit& z) const {
        Zeit t(*this);
        unsigned gSek = z.tage * 86400 + z.std * 3600 + z.min * 60 + z.sek +
                        t.tage * 86400 + t.std * 3600 + t.min * 60 + t.sek;
        t.tage = gSek / 86400;  gSek %= 86400;
        t.std  = gSek / 3600;   gSek %= 3600;
        t.min  = gSek / 60;
        t.sek  = gSek % 60;
        return t;
     }
  private:
     unsigned tage, std, min, sek;
};
int main(void) {
   vector<Zeit> z;
   for (int i=0; i < 5; i++) {
      int h = rand()%30;
      int m = rand()%60;
      int s = rand()%60;
      z.push_back( Zeit(h, m, s) );
      cout << Zeit(h, m, s);
   }
   cout << "------------------" << endl
        << accumulate(z.begin(), z.end(), Zeit(0, 0, 0), plus<Zeit>() );
}
```

Programm 4.77 liefert z. B. die folgende Ausgabe:

```
 0 Tage, 13:46:57
 0 Tage, 25:53:55
 0 Tage, 16:12:09
```

```
 0 Tage, 01:02:07
 0 Tage, 20:19:23
------------------
 3 Tage, 05:14:31
```

4.5.3 Vordefinierte Funktionsobjekte für Vergleiche

Tabelle 4.2 zeigt die vordefinierten Funktionsobjekte für Vergleiche.

Tabelle 4.2: Vordefinierte Funktionsobjekte für Vergleiche

Funktionsobjekt	Operation	Rückgabetyp	Argumentzahl
`equal_to`	x == y	bool	2
`not_equal_to`	x != y	bool	2
`greater`	x > y	bool	2
`greater_equal`	x >= y	bool	2
`less`	x < y	bool	2
`less_equal`	x <= y	bool	2

Programm 4.78 – `fktsobj4.cpp`:
Demoprogramm zu den vordefinierten Funktionsobjekten für Vergleiche

```
#include <vector>
#include <string>
#include <functional>
#include <iomanip>
#include <iostream>
using namespace std;

int main(void) {
   int x = 23, y = 5, z = 23, i;
   equal_to<int>      eq;
   not_equal_to<int>  ne;
   greater_equal<int> ge;
   less_equal<int>    le;
   cout << boolalpha;
   cout << "equal_to("     << x << "," << y << ") = " << eq(x, y) << endl;
   cout << "not_equal_to(" << x << "," << y << ") = " << ne(x, y) << endl;
   cout << "greater(" << x << "," << y << ") = " << greater<int>()(x, y) <<endl;
   cout << "greater_equal(" << x << "," << z << ") = " << ge(x, z) <<endl;
   cout << "less(" << y << "," << x << ") = " << less<int>()(y, x) <<endl;
   cout << "less_equal(" << x << "," << y << ") = " << le(x, y) <<endl;
   vector<string> namen;
   namen.push_back("Micha");
   namen.push_back("Zorro");
   namen.push_back("Adam");
   namen.push_back("Berta");
   namen.push_back("Emil");
```

```
   sort(namen.begin(), namen.end(), less<string>());
   for (i = 0; i < namen.size(); i++)
      cout << namen[i] << ", ";
   cout << endl;
   sort(namen.begin(), namen.end(), greater<string>());
   for (i = 0; i < namen.size(); i++)
      cout << namen[i] << ", ";
}
```

Programm 4.78 liefert die folgende Ausgabe:

```
equal_to(23,5) = false
not_equal_to(23,5) = true
greater(23,5) = true
greater_equal(23,23) = true
less(5,23) = true
less_equal(23,5) = false
Adam, Berta, Emil, Micha, Zorro,
Zorro, Micha, Emil, Berta, Adam,
```

4.5.4 Vordefinierte Funktionsobjekte für logische Operationen

Tabelle 4.3 zeigt die vordefinierten Funktionsobjekte für logische Operationen.

Tabelle 4.3: Vordefinierte Funktionsobjekte für logische Operationen

Funktionsobjekt	Operation	Rückgabetyp	Argumentzahl
`logical_and`	x && y	bool	2
`logical_or`	x || y	bool	2
`logical_not`	!x	bool	1

Programm 4.79 – `fktsobj5.cpp`:
Demoprogramm zu den vordefinierten Funktionsobjekten für logische Operationen

```
#include <functional>
#include <iomanip>
#include <iostream>
using namespace std;
int main(void) {
   cout << setw(8) << "x" << " | " << setw(8) << "y" << " || "
        << setw(11) << "logical_and" << " | "
        << setw(11) << "logical_or"  << " | "
        << setw(11) << "logical_not" << " | " << boolalpha << endl;
   for (int i=0; i <= 1; i++)
      for (int j=0; j <= 1; j++) {
          bool x = i, y = j;
          cout << setw(8) << x << " | " << setw(8) << y << " || "
               << setw(11) << logical_and<bool>()(x,y) << " | "
```

```
               << setw(11) << logical_or <bool>()(x,y) << " | "
               << setw(11) << logical_not<bool>()(x) << " | " << endl;
      }
   bool array[] = { false, false, true, true, false, true };
   transform(array, array+6, array, logical_not<bool>());
   for (int b=0; b < 6; b++)
      cout << array[b] << ", ";
}
```

Programm 4.79 liefert die folgende Ausgabe:

```
       x |        y || logical_and |  logical_or | logical_not |
   false |    false ||       false |       false |        true |
   false |     true ||       false |        true |        true |
    true |    false ||       false |        true |       false |
    true |     true ||        true |        true |       false |
true, true, false, false, true, false,
```

4.5.5 Adapter

Funktionsobjekte, die andere Funktionsobjekte als Argument übernehmen und so neue Funktionsobjekte zusammensetzen, werden als *Adapter* bezeichnet. Eine Kombination von bereits definierten Funktionsobjekten mit Hilfe von Adaptern erspart oft die Entwicklung neuer Funktionsobjekte.

Festlegen eines festen Werts für ein Argument mit dem bind-Adapter

bind-Adapter ermöglichen es, zweistellige Funktionsobjekte als einstellige zu nutzen, indem sie für ein Argument eines zweistelligen Funktionsobjekts ein konstantes Funktionsobjekt festlegen. Legt man z. B. für das zweistellige Funktionsobjekt `minus<int>()` als erstes Argument den Wert 100 fest, so bedeutet dies, dass dieses Funktionsobjekt immer das zweite Argument von 100 subtrahiert.

binder1st und bind1st – Festlegen eines festen Werts für das 1. Argument

Die Realisierung des Funktionsobjekts `binder1st` könnte z. B. wie folgt aussehen:

```
template <class Operation>
class binder1st : public unary_function<typename Operation::second_argument_type,
                                        typename _Operation::result_type> {
public:
  binder1st(const Operation& x,
            const typename Operation::first_argument_type& y) : op(x), value(y) {}
  typename Operation::result_type operator()
                 (const typename Operation::second_argument_type& x) const {
     return op(value, x);
  }
protected:
  Operation op;
  typename  Operation::first_argument_type value;
};
```

Hier ist erkennbar, dass der Konstruktor die Operation `op` mit `x` und `value` mit `y` initialisiert und dass der `operator()` als Rückgabe `op(value, x)` liefert.
Um auch Aufrufe für Typen zu ermöglichen, die nicht implizit konvertiert werden können, weil z. B. der dazu benötigte Konstruktor `explicit` deklariert ist, wird noch folgende Hilfsfunktion angeboten:

```
template <class Operation, class T>
inline binder1st<Operation>
bind1st(const Operation& op, const T& x) {
  typedef typename Operation::first_argument_type Arg1_type;
  return binder1st<Operation>(op, Arg1_type(x));
}
```

Programm 4.80 – `fktsobj6.cpp`:
Demoprogramm zu binder1st() und bind1st()

```
#include <functional>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;
int main(void) {
   int array[] =  23, 45, 110, 500, 0 ;
   transform(array, array+5, array, binder1st<minus<int> >(minus<int>(), 100 ) );
   for (int i=0; i < 5; i++)
      cout << array[i] << ", ";
   cout << endl;
   transform(array, array+5,
             ostream_iterator<int>(cout, ", "), bind1st(multiplies<int>(), 2));
}
```

Programm 4.80 liefert die folgende Ausgabe:

```
77, 55, -10, -400, 100,
154, 110, -20, -800, 200,
```

binder2nd und bind2nd – Festlegen eines festen Werts für das 2. Argument

Die Realisierung des Funktionsobjekts `binder2nd` könnte z. B. wie folgt aussehen:

```
template <class Operation>
class binder2nd : public unary_function<typename Operation::first_argument_type,
                                        typename Operation::result_type> {
public:
  binder2nd(const Operation& x,
            const typename Operation::second_argument_type& y) : op(x), value(y) {}
  typename Operation::result_type
  operator()(const typename Operation::first_argument_type& x) const {
     return op(x, value);
  }
protected:
  Operation op;
  typename  Operation::second_argument_type value;
};
```

Hier ist erkennbar, dass der Konstruktor die Operation `op` mit `x` und `value` mit `y` initialisiert und dass der `operator()` als Rückgabe `op(x, value)` liefert. Diese Rückgabe unterscheidet sich von `binder1st()`, dass die Reihenfolge der beiden Argumente `x` und `value` hier vertauscht ist.
Um auch hier Aufrufe für Typen zu ermöglichen, die nicht implizit konvertiert werden können, weil z. B. der dazu benötigte Konstruktor `explicit` deklariert ist, wird folgende Hilfsfunktion angeboten:

```
template <class Operation, class T>
inline binder2nd<Operation>
bind2nd(const Operation& op, const T& x) {
  typedef typename Operation::second_argument_type Arg2_type;
  return binder2nd<Operation>(op, Arg2_type(x));
}
```

Programm 4.81 – `fktsobj7.cpp`:
Demoprogramm zu binder2nd() und bind2nd()

```
#include <vector>
#include <functional>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;
template<typename T>
ostream& operator<< (ostream& os, vector<T> v) {
    typename vector<T>::iterator it;
    for (it = v.begin(); it != v.end(); it++)
       os << *it << ", ";
    return os << endl;
}
int main(void) {
  unsigned    i;
  vector<int> v;
   for (i=0; i < 20; i++)
      v.push_back(rand()%100);
   cout << v;
   transform(v.begin(), v.end(), ostream_iterator<int>(cout, ",  "),
             binder2nd<greater<int> >(greater<int>(), 50 ) );
   cout << endl << "...mit 1 gekennzeichnete sind groesser als 50" << endl;
   cout << "kleiner oder gleich 50 sind: "
        << count_if(v.begin(), v.end(), bind2nd(less_equal<int>(), 50)) << endl;
}
```

Programm 4.81 liefert z. B. die folgende Ausgabe:

```
83, 86, 77, 15, 93, 35, 86, 92, 49, 21, 62, 27, 90, 59, 63, 26, 40, 26, 72, 36,
1,  1,  1,  0,  1,  0,  1,  1,  0,  0,  1,  0,  1,  1,  1,  0,  0,  0,  1,  0,
...mit 1 gekennzeichnete sind groesser als 50
kleiner oder gleich 50 sind: 9
```

Negierer

Negierer sind Adapter, die Wahrheitswerte anderer Funktionsobjekte negieren.

unary_negate und not1 – Negierer für einstellige Funktionsobjekte

Die Realisierung von `unary_negate` könnte z. B. wie folgt aussehen:

```
template <class Predicate>
class unary_negate
     : public unary_function<typename Predicate::argument_type, bool> {
public:
   explicit unary_negate(const Predicate& x) : pred(x) {}
   bool operator()(const typename Predicate::argument_type& x) const {
      return !pred(x);
   }
protected:
   Predicate pred;
};
```

Da die Benutzung von `unary_negate` doch etwas umständlich ist, bietet die Standardbibliothek noch die folgende Hilfsfunktion an:

```
template <class Predicate>
inline unary_negate<Predicate>
not1(const Predicate& pred) {
   return unary_negate<Predicate>(pred);
}
```

binary_negate und not2 – Negierer für zweistellige Funktionsobjekte

Die Realisierung von `binary_negate` könnte z. B. wie folgt aussehen:

```
template <class Predicate>
class binary_negate
     : public binary_function<typename Predicate::first_argument_type,
                              typename Predicate::second_argument_type, bool> {
public:
   explicit binary_negate(const Predicate& x) : pred(x)
   bool operator()(const typename Predicate::first_argument_type& x,
                   const typename Predicate::second_argument_type& y) const {
      return !pred(x, y);
   }
protected:
   Predicate pred;
};
```

Da die Benutzung von `binary_negate` doch etwas umständlich ist, bietet die Standardbibliothek noch die folgende Hilfsfunktion an:

```
template <class Predicate>
inline binary_negate<Predicate>
not2(const _Predicate& pred) {
   return binary_negate<Predicate>(pred);
}
```

Programm 4.82 – `fktsobj8.cpp`:
Demoprogramm zu Negierern

```
#include <vector>
#include <functional>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;

template<typename T>
ostream& operator<< (ostream& os, vector<T> v) {
    typename vector<T>::iterator it;
    for (it = v.begin(); it != v.end(); it++)
       os << *it << ", ";
    return os << endl;
}

int main(void) {
   vector<int> v;

   for (unsigned i=0; i < 20; i++)
      v.push_back(rand()%100);
   cout << v;
   transform(v.begin(), v.end(), ostream_iterator<int>(cout, ",  "),
             unary_negate<binder2nd<less_equal<int> > >
                 (bind2nd(less_equal<int>(), 50 ) ) );
   cout << endl << "...mit 1 gekennzeichnete sind groesser als 50" << endl;
   cout << "kleiner oder gleich 50 sind: "
        << count_if(v.begin(), v.end(), not1(bind2nd(greater<int>(), 50))) <<endl;
   cout << v;
   transform(v.begin(), v.end(), ostream_iterator<int>(cout, ",  "),
             not1(bind2nd(modulus<int>(), 2))); // bei geraden Zahlen true
   cout << endl << "...mit 1 gekennzeichnete sind gerade" << endl;

   vector<string> namen;
   namen.push_back("Micha");
   namen.push_back("Zorro");
   namen.push_back("Adam");
   namen.push_back("Berta");
   namen.push_back("Emil");
   sort(namen.begin(), namen.end(), not2(less<string>()));
   cout << namen;
   sort(namen.begin(), namen.end(), not2(greater<string>()));
   cout << namen;
}
```

Programm 4.82 liefert z. B. die folgende Ausgabe:

```
83, 86, 77, 15, 93, 35, 86, 92, 49, 21, 62, 27, 90, 59, 63, 26, 40, 26, 72, 36,
1,  1,  1,  0,  1,  0,  1,  1,  0,  0,  1,  0,  1,  1,  1,  0,  0,  0,  1,  0,
...mit 1 gekennzeichnete sind groesser als 50
```

```
kleiner oder gleich 50 sind: 9
83, 86, 77, 15, 93, 35, 86, 92, 49, 21, 62, 27, 90, 59, 63, 26, 40, 26, 72, 36,
0,  1,  0,  0,  0,  0,  1,  1,  0,  0,  1,  0,  1,  0,  0,  1,  1,  1,  1,  1,
...mit 1 gekennzeichnete sind gerade
Zorro, Micha, Emil, Berta, Adam,
Adam, Berta, Emil, Micha, Zorro,
```

Notwendigkeit von Funktionsobjekten zu eigenen Funktionen

Programm 4.83 zeigt, wie man in STL-Algorithmen eigene Funktionen aufrufen lassen kann. Dieses Programm lässt sich aus einem `char`-Vektor alle Zeichen ausgeben, die keine Buchstaben sind. Dazu übergibt es an `remove_copy_if()` die Adresse der eigenen Funktion `istBuchstabe()`.

Programm 4.83 – `fktsobj9.cpp`:
Aufruf eigener Funktionen in STL-Algorithmen

```
#include <cctype>
#include <vector>
#include <functional>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;
bool istBuchstabe(unsigned char zeich) {
   return islower(zeich) || isupper(zeich);
}
int main(void) {
   vector<char> v;

   for (unsigned i=0; i < 30; i++) {
      char zeich;
      v.push_back( zeich = char(rand()%90 + 33) );
      cout << zeich;
   }
   cout << endl;
   remove_copy_if(v.begin(), v.end(), ostream_iterator<char>(cout, ""),
                  istBuchstabe);
   cout << endl;
}
```

Programm 4.83 liefert z. B. die folgende Ausgabe:

```
j1<:8:O-f"AdSp8m]EUOh)xf_q_.FX
1<:8:-"8])__.
```

Möchte man sich nun alle Zeichen aus dem `char`-Vektor ausgeben lassen, die Buchstaben sind, wäre der folgende Aufruf naheliegend:

```
   remove_copy_if(v.begin(), v.end(), ostream_iterator<char>(cout, ""),
                  not1(istBuchstabe()));
```

Dieser Aufruf ist jedoch nicht möglich, da `istBuchstabe()` eine Funktion und kein Funktionsobjekt mit überladenen `operator()` ist.
Man könnte das Problem somit beheben, indem man eine Klasse `IstBuchstabe` einführt, wie es in Programm 4.84 gezeigt ist.

Programm 4.84 – `fktsobj10.cpp`:
In Funktionsobjekt eingebetteter Aufruf eigener Funktionen in STL-Algorithmen

```
......
bool istBuchstabe(unsigned char zeich) {
   return islower(zeich) || isupper(zeich);
}
struct IstBuchstabe : unary_function<unsigned char, bool> {
   bool operator()(unsigned char z) const { return istBuchstabe(z); }
};
int main(void) {
   vector<char> v;

   for (unsigned i=0; i < 30; i++) {
      char zeich;
      v.push_back( zeich = char(rand()%90 + 33) );
      cout << zeich;
   }
   cout << endl;
   remove_copy_if(v.begin(), v.end(), ostream_iterator<char>(cout, ""),
                  not1(IstBuchstabe()));
   cout << endl;
}
```

Programm 4.84 liefert z. B. die folgende Ausgabe:

```
j1<:8:O-f"AdSp8m]EUOh)xf_q_.FX
jOfAdSpmEUOhxfqFX
```

Um nun eigene Funktionen in solchen Fällen aufrufen zu lassen, ohne dass man zuerst ein Funktionsobjekt dazu erstellt, bietet die Standardbibliothek Adapter für Funktionszeiger an, die eine Zusammenarbeit von eigenen ein- bzw. zweistelligen Funktionen mit Bibliotheksfunktionen erlauben.

Adapter für Zeiger auf einstellige Funktionen

Die Realisierung des Adapters `pointer_to_unary_function` könnte z. B. wie folgt aussehen:

```
template <class Arg, class Result>
class pointer_to_unary_function : public unary_function<Arg, Result> {
   public:
     pointer_to_unary_function()
     explicit pointer_to_unary_function(Result (*x)(Arg)) : ptr(x)
     Result operator()(Arg x) const  return ptr(x);
  protected:
     Result (*ptr)(Arg);
};
```

Da die Benutzung von `pointer_to_unary_function` doch etwas umständlich ist, bietet die Standardbibliothek noch die folgende Hilfsfunktion an:

```
template <class Arg, class Result>
inline pointer_to_unary_function<Arg, Result>
ptr_fun(Result (*x)(Arg)) {
   return pointer_to_unary_function<Arg, Result>(x);
}
```

Adapter für Zeiger auf zweistellige Funktionen

Die Realisierung des Adapters `pointer_to_binary_function` könnte z. B. wie folgt aussehen:

```
template <class Arg1, class Arg2, class Result>
class pointer_to_binary_function : public binary_function<Arg1,Arg2,Result> {
  public:
     pointer_to_binary_function() {}
     explicit pointer_to_binary_function(Result (*x)(Arg1, Arg2)) : ptr(x) {}
     Result operator()(Arg1 x, Arg2 y) const {
        return ptr(x, y);
     }
  protected:
     Result (*ptr)(Arg1, Arg2);
};
```

Da die Benutzung von `pointer_to_binary_function` doch etwas umständlich ist, bietet die Standardbibliothek noch die folgende Hilfsfunktion an:

```
template <class Arg1, class Arg2, class Result>
inline pointer_to_binary_function<Arg1,Arg2,Result>
ptr_fun(_Result (*x)(Arg1, Arg2)) {
   return pointer_to_binary_function<Arg1,Arg2,Result>(x);
}
```

Programm 4.85 – `fktsobj11.cpp`:
Demoprogramm zu Funktionszeiger-Adaptern

```
#include <cctype>
#include <cmath>
#include <vector>
#include <functional>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;
bool istBuchstabe(unsigned char zeich) { return islower(zeich) || isupper(zeich); }
int main(void) {
  vector<char> v;
   for (unsigned i=0; i < 30; i++) {
      char zeich;
      v.push_back( zeich = char(rand()%90 + 33) );
      cout << zeich;
```

```
   }
   cout << endl;
   remove_copy_if(v.begin(), v.end(), ostream_iterator<char>(cout, ""),
             not1(pointer_to_unary_function<unsigned char, bool>(istBuchstabe)));
   cout << endl;
   remove_copy_if(v.begin(), v.end(), ostream_iterator<char>(cout, ""),
                  not1(ptr_fun(istBuchstabe)));
   cout << endl;
   double z[] = { 2, 3.1, 5 };
   double (*powZgr)(double, double) = pow;
   transform(z, z+3, ostream_iterator<double>(cout, ", "),
             bind2nd(ptr_fun(powZgr), 3));
   cout << endl;
   transform(z, z+3, ostream_iterator<double>(cout, ", "),
             bind1st(ptr_fun(powZgr), 2));
}
```

Programm 4.85 liefert z. B. die folgende Ausgabe:

```
j1<:8:O-f"AdSp8m]EUOh)xf_q_.FX
jOfAdSpmEUOhxfqFX
jOfAdSpmEUOhxfqFX
8, 29.791, 125,
4, 8.57419, 32,
```

Adapter für Zeiger auf Methoden

Als Adapter für Zeiger auf Methoden stellt die Standardbibliothek die folgenden Hilfsfunktionen zur Verfügung:

- für Container, die Instanzen der entsprechenden Klasse enthalten:
  `mem_fun_ref(&methode)` – Methode ohne bzw. mit einem Argument
- für Container, die Zeiger auf Objekte der entsprechenden Klasse enthalten:
  `mem_fun(&methode)` – Methode ohne bzw. mit einem Argument

Programm 4.86 – `fktsobj12.cpp`:
Demoprogramm zu mem_fun_ref() und mem_fun()

```
#include <functional>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;

class K {
  public:
     K(char zeich) : z(zeich) { }
     void ausgabe(void) const { cout << z;        } // ohne Argument
     char next(int i)   const { return char(z+i); } // mit Argument
  private:
     char z;
};
```

```
int main(void) {
   const K vok[] = { 'A', 'E', 'I', 'O', 'U' };
   const int a[] = { 5, 4, 3, 2, 1 };
   //..................................... fuer Methoden ohne Argument (Referenz)
   for_each(vok, vok+5, mem_fun_ref(&K::ausgabe));
   cout << endl;
   //................................ fuer Methoden mit einem Argument (Referenz)
   transform(vok, vok+5, a, ostream_iterator<char>(cout, ""),
             mem_fun_ref(&K::next));
   cout << endl;

   const K *kvok[] = { new K('a'),new K('e'),new K('i'),new K('o'),new K('u')};
   //..................................... fuer Methoden ohne Argument (Zeiger)
   for_each(kvok, kvok+5, mem_fun(&K::ausgabe));
   cout << endl;
   //................................ fuer Methoden mit einem Argument (Zeiger)
   transform(kvok, kvok+5, a, ostream_iterator<char>(cout, ""),
             mem_fun(&K::next));
   cout << endl;
}
```

Programm 4.86 liefert die folgende Ausgabe:

```
AEIOU
FILQV
aeiou
filqv
```

Kombinieren mehrerer Funktionsobjekte

Programm 4.87 zeigt, wie man mehrere Funktionsobjekte kombinieren kann. Es zeigt wie man einen ganzen Vektor mit einen Bruch multiplizierfen lassen kann.

Programm 4.87 – `fktsobj13.cpp`:
Demoprogramm zum Kombinieren mehrerer Funktionsobjekte

```
#include <vector>
#include <functional>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;

template<typename T>
ostream& operator<< (ostream& os, vector<T> v)
{
    typename vector<T>::iterator it;
    for (it = v.begin(); it != v.end(); it++)
       os << *it << ", ";
    return os << endl;
}
```

```
//.............................................................. unary_bruchMult
template<typename Op1, typename Op2>
class unary_bruchMult : public unary_function<typename Op2::argument_type,
                                              typename Op1::result_type> {
public:
  unary_bruchMult(Op1 x, Op2 y) : op1(x), op2(y) {}
  typename Op1::result_type operator()
      (const typename Op2::argument_type& z) const { return op1(op2(z)); }
private:
  Op1 op1;
  Op2 op2;
};
//........................................ Zugehoerige Hilfsfunktion bruchMult
template<typename Op1, typename Op2>
inline unary_bruchMult<Op1, Op2>
bruchMult(Op1 oper1, Op2 oper2) {
   return unary_bruchMult<Op1, Op2>(oper1, oper2);
}
//..................................................................... main
int main(void) {
   vector<double> v;

   for (unsigned i=0; i < 5; i++)
      v.push_back( rand()%100 );
   cout << v;
   //.... Ausgabe von v / 3 * 4
   transform(v.begin(), v.end(), ostream_iterator<double>(cout, ", "),
             bruchMult(bind2nd(multiplies<double>(), 4),
                       bind2nd(divides<double>(), 3) ) );
   cout << endl;
   //.... Alle Elemente von v[i] = v[i] / 3 * 4
   transform(v.begin(), v.end(), v.begin(),
             bruchMult(bind2nd(multiplies<double>(), 4),
                       bind2nd(divides<double>(), 3) ) );
   //.... Ausgabe von v / 4 * 3
   transform(v.begin(), v.end(), ostream_iterator<double>(cout, ", "),
             bruchMult(bind2nd(multiplies<double>(), 3),
                       bind2nd(divides<double>(), 4) ) );
   cout << endl;
}
```

Programm 4.87 liefert z. B. die folgende Ausgabe:

```
83, 86, 77, 15, 93,
110.667, 114.667, 102.667, 20, 124,
83, 86, 77, 15, 93,
```

In Programm 4.87 ist allerdings auch erkennbar, dass solche kombinierten Funktionsobjekte doch sehr kompiliziert und nicht sehr gut lesbar sind. Hier wäre sicherlich die Definition eines neuen Funktionsobjekts die bessere Wahl, wie es auch in Programm 4.88 gezeigt ist.

Programm 4.88 – `fktsobj14.cpp`:
Eigenes neues Funktionsobjekt statt Kombinieren mehrerer Funktionsobjekte

```
#include <vector>
#include <functional>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;

template<typename T>
ostream& operator<< (ostream& os, vector<T> v) {
    typename vector<T>::iterator it;
    for (it = v.begin(); it != v.end(); it++)
       os << *it << ", ";
    return os << endl;
}
//.............................................................. Klasse bruchMult
template<typename T>
class bruchMult : public unary_function<T, double> {
  public:
     bruchMult(T z, T n) : zaehler(z), nenner(n) {}
     double operator() (double zahl) const { return zahl * zaehler / nenner; }
  private:
     const T zaehler, nenner;
};
//..................................................................... main
int main(void) {
   vector<double> v;

   for (unsigned i=0; i < 5; i++)
      v.push_back( rand()%100 );
   cout << v;
   //.... Ausgabe von v / 3 * 4
   transform(v.begin(), v.end(), ostream_iterator<double>(cout, ", "),
             bruchMult<int>(4, 3));
   cout << endl;
   //.... Alle Elemente von v[i] = v[i] / 3 * 4
   transform(v.begin(), v.end(), v.begin(), bruchMult<int>(4, 3));
   //.... Ausgabe von v / 4 * 3
   transform(v.begin(), v.end(), ostream_iterator<double>(cout, ", "),
             bruchMult<int>(3, 4));
   cout << endl;
}
```

Programm 4.88 liefert die gleiche Ausgabe wie Programm 4.87.

4.6 Algorithmen

Die Algorithmen sind wohl mit die wichtigsten Komponenten der C++-Standardbibliothek.

4.6.1 Allgemeines zu den Algorithmen

Die Headerdateien <algorithm> und <numeric>

Bis auf die numerischen Algorithmen, die in der Headerdatei `<numeric>` definert sind, sind alle anderen Algorithmen in der Headerdatei `<algorithm>` definert.

Typnamen für Template-Parameter

In diesem Kapitel werden die in Tabelle 4.4 gezeigten Typnamen für Template-Parameter verwendet.

Tabelle 4.4: Typnamen für Template-Parameter

Typname	Bedeutung
`InputIterator`	Iterator (einmaliges Vorwärtslesen (read-only))
`OutputIterator`	Iterator (einmaliges Vorwärtsschreiben (write-only))
`ForwardIterator`	Iterator (mehrmaliges Vorwärtslesen und -schreiben (read/write))
`BidirectionalIterator`	Iterator (mehrmaliges Vorwärts-/Rückwärtslesen und -schreiben)
`RandomAccessIterator`	Iterator mit Zeiger-Funktionalität
`UnaryOperation`	einstelliges Funktionsobjekt, das einen Wert liefert
`Predicate`	einstelliges Funktionsobjekt, das `bool`-Wert liefert
`BinaryOperation`	zweistelliges Funktionsobjekt, das einen Wert liefert
`BinaryPredicate`	zweistelliges Funktionsobjekt, das `bool`-Wert liefert
`Compare`	zweistelliges Funktionsobjekt, das seine beiden Argument vergleicht und einen `bool`-Wert liefert
`Function`	Funktion oder Funktionsobjekt, das seine Argumente meist nicht modifiziert
`Generator`	Funktionsobjekt, das Werte generiert
`RandomNumberGenerator`	Funktionsobjekt, das zufällige Werte generiert
`Size`	ganzahliger Typparameter für Größenangaben

4.6.2 Überblick zu den Algorithmen

Nachfolgend wird eine kurze Übersicht zu den einzelnen Algorithmen, wobei diese nach ihrem Einsatzgebiet gruppiert sind, und manche hier doppelt aufgezählt sind, wenn diese bei unterschiedlichen Aufgabenstellungen verwendet werden können.

Zähl-Algorithmen

- `count` $^{O(n)}$
  zählen, wie oft ein bestimmtes Element in einem Bereich vorhanden ist.
- `count_if` $^{O(n)}$
  zählen, wie viele Elemente eine vorgegebene Bedingung erfüllen.

Kopier-Algorithmen

- `copy` $O(n)$
  kopiert die Elemente eines Bereichs in einen anderen.
- `copy_backward` $O(n)$
  kopiert die Elemente eines Bereichs rückwärts in einen anderen.
- `transform` $O(n)$
  kopiert die Elemente eines Bereichs in einen anderen, wobei vor dem Kopiervorgang jedes Element verändert werden kann.
- `partial_sort_copy` $O(n \log n)$
  kopiert von einem Bereich mit Sortierung in einen anderen Bereich.
- `remove_copy` $O(n)$
  kopiert aus einem Bereich Elemente, die einen bestimmten Wert nicht besitzen, in einen anderen.
- `remove_copy_if` $O(n)$
  kopiert aus einem Bereich Elemente, die eine bestimmte Bedingung nicht erfüllen, in einen anderen.
- `replace_copy` $O(n)$
  kopiert die Elemente eines Bereichs in einen anderen, wobei Elemente mit einem bestimmten Wert einen neuen Wert erhalten.
- `replace_copy_if` $O(n)$
  kopiert die Elemente eines Bereichs in einen anderen, wobei Elemente, die eine bestimmte Bedingung erfüllen, einen neuen Wert erhalten.
- `reverse_copy` $O(n)$
  kopiert die Elemente eines Bereichs in umgekehrter Reihenfolge in einen anderen.
- `rotate_copy` $O(n)$
  kopiert zwei Teilbereiche in einen anderen Bereich, wobei es den zweiten Teilbereich vor den ersten kopiert.
- `unique_copy` $O(n)$
  kopiert die Elemente eines Bereichs in einen anderen, wobei bei Folgen von gleichen Elementen nur das Erste kopiert wird.

Sortier-Algorithmen

- `partial_sort` $O(n \log n)$
  sortiert von einem ganzen Bereich nur einen Teil am Anfang.
- `partial_sort_copy` $O(n \log n)$
  kopiert von einem Bereich mit Sortierung in einen anderen Bereich.
- `sort` $O(n \log n)$
  sortiert die Elemente eines Bereichs.
- `stable_sort` $O(n \log n)$ *oder* $O(n (\log n)^2)$
  sortiert die Elemente eines Bereichs, wobei die relative Reihenfolge gleicher Elemente erhalten bleibt.
- `nth_element` $O(n)$
  ordnet Elemente eines Bereichs so um, dass alle Elemente vor dem *n*-ten Element kleiner und alle danach größer sind.

Such-Algorithmen

- `find` $O(n)$
  liefert das erste Element, das einen bestimmten Wert besitzt.
- `find_if` $O(n)$
  liefert das erste Element, das eine vorgegebene Bedingung erfüllt.
- `find_first_of` $O(n^2)$
  liefert das erste Vorkommen eines Elements aus einem Bereich in einem anderen.
- `find_end` $O(n^2)$
  liefert das letzte Vorkommen eines Bereichs in einem anderen.
- `max_element` $O(n)$
  liefert Iterator auf das größte Element eines Bereichs.
- `min_element` $O(n)$
  liefert Iterator auf das kleinste Element eines Bereichs.
- `search` $O(n^2)$
  liefert das erste Vorkommen der Elemente aus einem Bereich in einem anderen.
- `search_n` $O(n^2)$
  liefert das erste Vorkommen von `n` gleichen Elementen in einem Bereich.
- `adjacent_find` $O(n)$
  sucht in einem Bereich nach zwei aufeinanderfolgende gleiche Elemente.
- `binary_search` $O(\log n)$
  prüft, ob ein gesuchtes Element in einem Bereich vorhanden ist.
- `equal_range` $O(2 \cdot \log n)$
  liefert Iteratorpaar entsprechend `lower_bound` und `upper_bound`, die nachfolgend beschrieben sind.
- `lower_bound` $O(\log n)$
  liefert erste Position, an der ein Element in einem Bereich eingefügt werden kann, so dass dieser weiterhin sortiert ist.
- `upper_bound` $O(\log n)$
  liefert letzte Position, an der ein Element in einem Bereich eingefügt werden kann, so dass dieser weiterhin sortiert ist.

Lösch-Algorithmen

- `remove` $O(n)$
  entfernt aus einem Bereich Elemente, die einen bestimmten Wert besitzen.
- `remove_if` $O(n)$
  entfernt aus einem Bereich Elemente, die eine bestimmte Bedingung erfüllen.
- `remove_copy` $O(n)$
  kopiert aus einem Bereich Elemente, die einen bestimmten Wert nicht besitzen, in einen anderen.
- `remove_copy_if` $O(n)$
  kopiert aus einem Bereich Elemente, die eine bestimmte Bedingung nicht erfüllen, in einen anderen.

Ersetz-Algorithmen

- `replace` $O(n)$
  ersetzt in einem Bereich die Werte von Elementen, die einen bestimmten Wert besitzen, durch einen neuen Wert.
- `replace_if` $O(n)$
  ersetzt in einem Bereich die Werte von Elementen, die eine bestimmte Bedingung erfüllen, durch einen neuen Wert.
- `replace_copy` $O(n)$
  kopiert die Elemente eines Bereichs in einen anderen, wobei Elemente mit einem bestimmten Wert einen neuen Wert erhalten.
- `replace_copy_if` $O(n)$
  kopiert die Elemente eines Bereichs in einen anderen, wobei Elemente, die eine bestimmte Bedingung erfüllen, einen neuen Wert erhalten.

Vergleichs-Algorithmen

- `equal` $O(n)$
  prüft, ob die Elemente zweier Bereiche paarweise gleich sind.
- `mismatch` $O(n)$
  liefert das erste Wertepaar, bei dem sich zwei Bereiche unterscheiden.
- `max` $O(1)$
  liefert das Größere von zwei Objekten.
- `min` $O(1)$
  liefert das Kleinere von zwei Objekten.
- `lexicographical_compare` $O(n)$
  prüft, ob die Elemente eines Bereichs lexikographisch kleiner sind als die Elemente eines anderen Bereichs.
- `includes` $O(n)$
  prüft, ob alle Elemente eines Bereichs in anderem enthalten sind (Teilmenge).

Tausch-, Umkehr- und Rotier-Algorithmen

- `swap` $O(1)$
  vertauscht zwei Objekte.
- `iter_swap` $O(1)$
  tauscht die Werte, auf die zwei Iteratoren zeigen.
- `swap_ranges` $O(n)$
  vertauscht die Elemente zweier Bereiche.
- `reverse` $O(n)$
  kehrt die Reihenfolge der Elemente eines Bereichs um.
- `reverse_copy` $O(n)$
  kopiert Elemente eines Bereichs in umgekehrter Reihenfolge in einen anderen.
- `rotate` $O(n)$
  vertauscht zwei Teilbereiche (Rotieren gegen Uhrzeigersinn).
- `rotate_copy` $O(n)$
  kopiert zwei Teilbereiche in einen anderen Bereich, wobei es den zweiten Teilbereich vor den ersten kopiert.

Traversierungs-Algorithmen

- `for_each` $O(n)$
  führt eine angegebene Funktion für die Elemente eines Bereichs aus.

Umordnungs-Algorithmen

- `random_shuffle` $O(n)$
  ordnet die Elemente eines Bereichs zufällig um.
- `nth_element` $O(n)$
  ordnet Elemente eines Bereichs so um, dass alle Elemente vor *n*-ten Element kleiner und alle danach größer sind.
- `unique` $O(n)$
  entfernt aus einer Folge von gleichen Elementen alle bis auf das Erste.

Gruppier-Algorithmen

- `partition` $O(n)$
  teilt die Elemente eines Bereichs in zwei Gruppen, wobei nur eine Gruppe einen vorgegebene Bedingung erfüllt.
- `stable_partition` $O(n \log n)$
  teilt die Elemente eines Bereichs in zwei Gruppen, wobei nur eine Gruppe einen vorgegebene Bedingung erfüllt; die relative Reihenfolge der Elemente bleibt dabei erhalten.

Permutations-Algorithmen

- `next_permutation` $O(n)$
  `prev_permutation` $O(n)$
  generieren die lexikographisch nächst größere bzw. nächst kleinere Permutation zu den Elementen eines Bereichs.

Misch-Algorithmen

- `merge` $O(n)$
  mischt zwei sortierte Bereiche in einen neuen Bereich.
- `inplace_merge` $O(n)$ *oder* $O(n \log n)$
  mischt die Elemente zweier aufeinanderfolgender, sortierter Teilbereiche, so das ein sortierter Bereich entsteht.

Initialisierungs-Algorithmen

- `fill` $O(n)$
  füllt einen Bereich mit Elementen eines bestimmten Werts.
- `fill_n` $O(n)$
  füllt `n` Elemente am Anfang eines Bereichs mit einem bestimmten Wert.
- `generate` $O(n)$
  füllt einen Bereich mit Hilfe eines Funktionsobjekts.
- `generate_n` $O(n)$
  füllt `n` Elemente am Anfang eines Bereichs mit Hilfe eines Funktionsobjekts.

Mengen-Algorithmen

- `includes` $O(n)$
  prüft, ob alle Elemente eines Bereichs in einem anderen Bereich enthalten sind (Teilmenge).
- `set_difference` $O(n)$
  bildet Differenzmenge zweier Bereiche, was die Elemente sind, die im ersten, aber nicht im zweiten Bereich vorkommen.
- `set_intersection` $O(n)$
  bildet Schnittmenge zweier Bereiche, was die Elemente sind, die in beiden Bereichen vorkommen.
- `set_symmetric_difference` $O(n)$
  kopiert die Elemente zweier Bereiche, die nur in einem der beiden Bereiche enthalten sind, in einen anderen Bereich.
- `set_union` $O(n)$
  bildet Vereinigungsmenge zweier Bereiche, was die Elemente sind, die in mindestens einem der beiden Bereichen vorkommen.

Numerische Algorithmen

- `accumulate` $O(n)$
  liefert die Summe der Elemente eines Bereichs.
- `adjacent_difference` $O(n)$
  berechnet jeweils die Differenz zweier aufeinanderfolgender Elemente eines Bereichs und kopiert die Ergebnisse in einen anderen Bereich.
- `inner_product` $O(n)$
  liefert die Summe der Produkte der korrespondierenden Elemente zweier Bereiche.
- `partial_sum` $O(n)$
  berechnet die fortlaufenden Summen der Elemente eines Bereichs und kopiert die Ergebnisse in einen anderen Bereich.

Heap-Algorithmen

- `make_heap` $O(n)$
  ordnet die Elemente eines Bereichs so um, dass sie einen Heap bilden.
- `pop_heap` $O(\log n)$
  entfernt ein Element aus einem Heap.
- `push_heap` $O(\log n)$
  fügt ein neues Element zu einem Heap hinzu.
- `sort_heap` $O(n \log n)$
  wandelt einen Heap in einen sortierten Bereich um.

Nachfolgend werden nun die einzelnen Algorithmen in alphabetischer Reihenfolge vorgestellt. Dabei wird immer zuerst die Headerdatei angegeben, die man bei Verwendung des entsprechenden Algorithmus inkludieren muss. Anschließend wird dessen Prototyp angegeben, wobei manchmal auch eine mögliche Realisierung des jeweiligen Algorithmus gezeigt wird, wenn dies dem Verständnis dient.

4.6.3 accumulate – Summe eines Bereichs

```
                                                    #include <numeric>
```

```
//... 1. Variante: verwendet operator+()
template<typename InputIterator, typename T>
T accumulate(InputIterator first, InputIterator last, T init) {
   for ( ; first != last; ++first)
      init = init + *first;
   return init;
}
//... 2. Variante: verwendet Funktionsobjekt binary_op()
template<typename InputIterator, typename T, typename BinaryOperation>
T accumulate(InputIterator first, InputIterator last, T init,
             BinaryOperation binary_op) {
   for ( ; first != last; ++first)
      init = binary_op(init, *first);
   return init;
}
```

`accumulate` liefert die Summe der Elemente aus dem Bereich `[first, last)` plus dem Startwert `init`, der den Typ des Rückgabewerts festlegt.

Programm 4.89 – `accumulate.cpp`:
Demoprogramm zu accumulate

```
#include <vector>
#include <string>
#include <algorithm>
#include <numeric>
#include <iterator>
#include <iostream>
using namespace std;
int main(void) {
  const int   arr[] = { 1, 2, 3, 4, 5 };
  vector<int> v(arr, arr+5);
   cout << "Summe:   " << accumulate(v.begin(), v.end(), 0) << endl;
   cout << "Produkt: " << accumulate(v.begin(), v.end(), 1,
                                     multiplies<int>()) << endl;
   cout << "Produkt: " << accumulate(v.begin(), v.end(), 1.333,
                                     multiplies<double>()) << endl;
   string s[] =  "Hallo, ", "wie ", "geht's ", "denn so" ;
   vector<string> vs(s, s+4);
   cout << accumulate(vs.begin(), vs.end(), string("---")) << endl;
}
```

Programm 4.89 liefert die folgende Ausgabe:

```
Summe:   15
Produkt: 120
Produkt: 159.96
---Hallo, wie geht's denn so
```

4.6.4 adjacent_difference – Differenz benachbarter Elemente

```
#include <numeric>
```

```
//... 1. Variante: verwendet operator-()
template<typename InputIterator, typename OutputIterator>
OutputIterator adjacent_difference(InputIterator first, InputIterator last,
                                   OutputIterator result);
//... 2. Variante: verwendet binären Operator op
template<typename InputIterator, typename OutputIterator, typename BinaryOperation>
OutputIterator adjacent_difference(InputIterator first, InputIterator last,
                                   OutputIterator result, BinaryOperation op);
```

`adjacent_difference` berechnet für den Bereich `[first, last)` die Differenzen zweier aufeinanderfolgender Elemente und schreibt die Ergebnisse in den bei `result` beginnenden Bereich. So werden z. B. für einen Bereich mit den Elementen `a`, `b`, `c`, `d` folgende Werte nach `result` geschrieben: `a`, `b-a`, `c-b`, `d-c`. Bei der zweiten Varianten wird statt `operator-()` die angegebene binäre Operation `op` verwendet. Der Rückgabewert ist `result + (last - first)`.

Programm 4.90 – `adjacentdiff.cpp`:
Demoprogramm zu adjacent_difference

```
#include <vector>
#include <algorithm>
#include <numeric>
#include <iterator>
#include <iostream>
using namespace std;

int main(void) {
  int celsius[] = { 10, 6, 12, 14, 6 }, diff[5];
   vector<int> v(celsius, celsius+5);
   copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
   adjacent_difference(v.begin(), v.end(), diff);
   cout << endl << showpos << "  ";
   copy(diff+1, diff+5, ostream_iterator<int>(cout, " "));
   adjacent_difference(v.begin(), v.end(), diff, minus<int>());
   cout << endl << "  ";
   copy(diff+1, diff+5, ostream_iterator<int>(cout, " "));
   adjacent_difference(v.begin(), v.end(), diff, modulus<int>());
   cout << endl << "   " << noshowpos;
   copy(diff+1, diff+5, ostream_iterator<int>(cout, "  "));
}
```

Programm 4.90 liefert die folgende Ausgabe:

```
10 6 12 14 6
  -4 +6 +2 -8
  -4 +6 +2 -8
   6  0  2  6
```

4.6.5 adjacent_find – Suchen gleicher benachbarter Elemente

```
#include <algorithm>
```

```
//... 1. Variante: verwendet operator==()
template<typename ForwardIterator>
ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last);
//... 2. Variante: verwendet pred zum Vergleichen
template<typename ForwardIterator, typename BinaryPredicate>
ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last,
                              BinaryPredicate pred);
```

Eine mögliche Realisierung von `adjacent_find` wäre z. B.:

```
if (first == last)
   return last;
ForwardIterator next = first;
while(++next != last) {
   if (*first == *next)  // bzw.  if (pred(*first, *next))
      return first;
   first = next;
}
return last;
```

`adjacent_find` sucht im Bereich `[first, last)` nach zwei aufeinanderfolgender Elemente, die gleich sind. Bei der zweiten Varianten wird statt `operator==` das Funktionsobjekt `pred()` zum Vergleichen verwendet. Der Rückgabewert ist das erste Element, dem ein Gleiches folgt, bzw. `last`, wenn keine gleiche aufeinanderfolgenden Elemente gefunden werden.

Programm 4.91 – `adjacentfind.cpp`: Demoprogramm zu adjacent_find

```
#include <vector>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;
class QuerSum {
  public:
     QuerSum() {}
     bool operator() (int first, int second) { return qSum(first)==qSum(second); }
  private:
     int qSum(int z) {
        int sum = 0;
        while (z > 0) {
           sum += z%10;
           z /= 10;
        }
        return sum;
     }
```

```
};
int main(void) {
   int z[] = { 1, 2, 2, 3, 4, 4, 5, 6, 6 };
   int *fzgr = z;
   while ( (fzgr = adjacent_find(fzgr, z+9)) != z+9)
       cout << *fzgr << ", " << *(fzgr++) << endl;
   string s[] = { "das", "ist", "der", "der", "Mann", "Zorro", "Zorro" };
   string *szgr = s;
   while ( (szgr = adjacent_find(szgr, s+7)) != s+7)
       cout << *szgr << ", " << *(szgr++) << endl;
   int q[] = { 99, 1881, 199, 54321, 12345 };
   int *qzgr = q;
   while ( (qzgr = adjacent_find(qzgr, q+5, QuerSum())) != q+5)
       cout << *qzgr << ", " << *(qzgr++) << endl;
}
```

Programm 4.91 liefert die folgende Ausgabe:

```
2, 2
4, 4
6, 6
der, der
Zorro, Zorro
1881, 99
12345, 54321
```

4.6.6 binary_search – Binäres Suchen

```
                                                    #include <algorithm>
```

```
//... 1. Variante: verwendet operator<()
template<typename ForwardIter, typename T>
bool binary_search(ForwardIter first, ForwardIterator last, const T& val) {
   ForwardIterator it = lower_bound(first, last, val);
   return it != last && !(val < *it);
}
//... 2. Variante: verwendet comp zum Vergleichen
template<typename ForwardIterator, typename T, typename Compare>
bool binary_search(ForwardIterator first, ForwardIterator last,
                   const T& val, Compare comp) {
   ForwardIterator it = lower_bound(first, last, val, comp);
   return it != last && !comp(val, *i);
}
```

`binary_search` sucht mittels binärer Suche im Bereich `[first, last)` nach einem Element, das gleich `val` ist. Der Bereich muss bei der ersten Variante mit dem `operator<` und bei der zweiten Variante mit dem Funktionsobjekt `comp` sortiert sein. Wird ein Element gefunden, das gleich `val` ist, liefert `binary_search` als Rückgabewert `true`, ansonsten `false`.

Programm 4.92 – `binsearch.cpp`:
Demoprogramm zu binary_search

```
#include <vector>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;
int main(void) {
   string s[] = { "Adam", "Berta", "Micha", "Robert", "Zorro" };
   if ( binary_search(s, s+5, "Micha") )
      cout << "Micha gefunden" << endl;
   reverse(s, s+5); //... Reihenfolge umkehren
   if ( binary_search(s, s+5, "Micha") )
      cout << "Micha gefunden" << endl;
   else
      cout << "Micha nicht gefunden" << endl;
   if ( binary_search(s, s+5, "Micha", greater<string>()) )
      cout << "Micha gefunden" << endl;
}
```

Programm 4.92 liefert die folgende Ausgabe:

```
Micha gefunden
Micha nicht gefunden
Micha gefunden
```

4.6.7 copy – Kopieren eines Bereichs

```
                                                    #include <algorithm>
```

```
template<typename InputIterator, typename OutputIterator> inline
OutputIterator copy(InputIterator first, InputIterator last,
                    OutputIterator result) {
   for ( ; first != last; ++result, ++first)
      *result = *first;
   return result;
}
```

`copy` kopiert die Elemente aus dem Bereich `[first, last)` unter Verwendung des entsprechenden Zuweisungsoperators in den bei `result` beginnenden Bereich.
Der Rückgabewert ist ein Iterator hinter das letzte kopierte Element. Quell- und Zielbereich dürfen sich nur dann überlappen, wenn `result < first` ist. Somit können Elemente von Containern von hinten nach vorne kopiert werden, was z. B. bei Löschoperationen in der Mitte eines Containers erforderlich ist.

Programm 4.93 – copy.cpp:
Demoprogramm zu copy

```
#include <string>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;

int main(void) {
   string s[] = { "Adam", "Berta", "Micha", "Robert", "Zorro" };
   copy(s+2, s+5, s);
   copy(s, s+5, ostream_iterator<string>(cout, " "));
   cout << endl;
}
```

Programm 4.93 liefert die folgende Ausgabe:

```
Micha Robert Zorro Robert Zorro
```

4.6.8 copy_backward – Rückwärtiges Kopieren eines Bereichs

```
                                                    #include <algorithm>
```

```
template<typename BidirectionalIter1, typename BidirectionalIter2> inline
BidirectionalIterator2 copy_backward(BidirectionalIterator1 first,
                                     BidirectionalIterator1 last,
                                     BidirectionalIterator2 result)
{
   while (first != last)
     *--result = *--last;
   return result;
}
```

`copy_backward` kopiert die Elemente aus dem Bereich `[first, last)` unter Verwendung des entsprechenden Zuweisungsoperators rückwärts in den bei `result` endenden Bereich. Der Iterator `result` verhält sich dabei wie ein Reverse-Iterator.
`result` sollte nicht im Bereich `[first, last)` liegen, damit keine Werte vor dem Kopieren überschrieben werden.
Der Rückgabewert ist ein Iterator auf das letzte kopierte Element: `result - (last - first)`.

Programm 4.94 – copyback.cpp:
Demoprogramm zu copy_backward

```
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;
```

```
int main(void)
{
   int z[] = { 1, 2, 3, 4, 5, 6, 7 };

   copy_backward(z, z+4, z+7);
   copy(z, z+7, ostream_iterator<int>(cout, " "));
   cout << endl;
}
```

Programm 4.94 liefert die folgende Ausgabe:

```
1 2 3 1 2 3 4
```

4.6.9 count – Zählen des Vorkommens eines Werts

```
                                                    #include <algorithm>
```

```
template<typename InputIterator, typename T>
typename iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last, const T& value) {
   typename iterator_traits<InputIterator>::difference_type n = 0;
   for ( ; first != last; ++first)
      if (*first == value)
        ++n;
   return n;
}
```

`count` gibt zurück, wie oft der Wert `value` im Bereich `[first, last)` vorhanden ist. Zum Vergleichen der einzelnen Werte verwendet `count` den `operator==`, der natürlich für den Typ `T` definiert sein muss.
Die assoziativen Container bieten eigene `count()`-Methoden an, die wesentlich effizienter sind, als der `count`-Algorithmus.

Programm 4.95 – `count.cpp`:
Demoprogramm zu count

```
#include <algorithm>
#include <iostream>
using namespace std;

int main(void)
{
   string s[] = { "Zorro", "Hans", "zorro", "Emil", "Emi", "Zorro", "Zorro" };

   cout << "Zorro kommt " << count(s, s+7, "Zorro") << " mal vor" << endl;
}
```

Programm 4.95 liefert die folgende Ausgabe:

```
Zorro kommt 3 mal vor
```

4.6.10 count_if – Zählen von Elementen, die Bedingung erfüllen

```
#include <algorithm>
```

```
template<typename InputIterator, typename Predicate>
typename iterator_traits<InputIterator>::difference_type
count_if(InputIterator first, InputIterator last, Predicate pred) {
   typename iterator_traits<InputIter>::difference_type n = 0;
   for ( ; first != last; ++first)
      if (pred(*first))
         ++n;
   return n;
}
```

`count_if` gibt die Anzahl der Werte aus dem Bereich `[first, last)` zurück, für die `pred` den Wert `true` liefert.

Programm 4.96 – `countif.cpp`:
Demoprogramm zu count_if

```
#include <algorithm>
#include <iostream>
using namespace std;

class Gerade {
  public:
     bool operator() (int zahl) { return zahl % 2 == 0; }
};
int main(void) {
   int z[] = { 1, 40, 53, 102, 34, 3, 58, 10 };
   cout << "Gerade Zahlen im Array: " << count_if(z, z+8, Gerade()) << endl;
}
```

Programm 4.96 liefert die folgende Ausgabe:

```
Gerade Zahlen im Array: 5
```

4.6.11 equal – Prüfen zweier Bereiche auf Gleichheit

```
#include <algorithm>
```

```
//... 1. Variante: verwendet operator<()
template<typename InputIterator1, typename InputIterator2> inline
bool equal(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2) {
   for ( ; first1 != last1; ++first1, ++first2)
      if (!(*first1 == *first2))
         return false;
   return true;
}
```

```
//... 2. Variante: verwendet comp zum Vergleichen
template<typename InputIterator1, typename InputIterator2,
         typename BinaryPredicate> inline
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, BinaryPredicate pred) {
   for ( ; first1 != last1; ++first1, ++first2)
      if (!pred(*first1, *first2))
         return false;
   return true;
}
```

`equal` prüft, ob die Elemente aus dem Bereich `[first1, last1)` mit denen aus dem Bereich, der bei `first2` beginnt, paarweise gleich sind. Sind alle Elemente aus dem ersten Bereich gleich dem aus dem zweiten Bereich, so liefert `equal` als Rückgabe `true`, andernfalls `false`.
Bei der zweiten Varianten wird statt `operator==` das zweistellige Funktionsobjekt `pred` zum Vergleichen verwendet.

Programm 4.97 – `equal.cpp`:
Demoprogramm zu equal

```
#include <algorithm>
#include <iomanip>
#include <iostream>
using namespace std;

int main(void)
{
   int z1[] = { 1, 2, 3, 4, 5, 6, 7 };
   int z2[] = { 1, 2, 4, 4, 5, 6, 7 };

   cout << boolalpha;
   cout << "z1 und z2 sind gleich in den" << endl;
   cout << "...ersten  2 Elementen: " << equal(z1, z1+2, z2) << endl;
   cout << "...ersten  4 Elementen: " << equal(z1, z1+4, z2) << endl;
   cout << "...letzten 3 Elementen: " << equal(z1+4, z1+7, z2+4) << endl;
   cout << "Hinsichtl. Paar-Vergleich gilt: z1 <= z2: ";
   cout << equal(z1, z1+7, z2, less_equal<int>()) << endl;
}
```

Programm 4.97 liefert die folgende Ausgabe:

```
z1 und z2 sind gleich in den
...ersten  2 Elementen: true
...ersten  4 Elementen: false
...letzten 3 Elementen: true
Hinsichtl. Paar-Vergleich gilt: z1 <= z2: true
```

4.6.12 equal_range – Einfügebereich für Wert ohne Umsortierung

```
#include <algorithm>
```

```
//... 1. Variante: verwendet operator<()
template<typename ForwardIterator, typename T>
pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first, ForwardIterator last, const T& val);
//... 2. Variante: verwendet comp zum Vergleichen
template<typename ForwardIterator, typename T, typename Compare>
pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first, ForwardIterator last, const T& val,
            Compare comp);
```

`equal_range` liefert den größten Teilbereich `[it1, it2)`, in dem `val` eingefügt werden kann, ohne dass dadurch die Sortierung verletzt wird. Die Iteratoren `it1` und `it2` liegen dabei beide im Bereich `[first, last)` Bei `it1==it2` kann `val` nur vor `it1` (bzw. `it2`) eingefügt werden. Der Bereich muss bei der ersten Variante mit dem `operator<` und bei der zweiten Variante mit dem Funktionsobjekt `comp` sortiert sein. Da die assoziativen Container Bidirectional-Iteratoren anbieten, sollte für sie nicht der Algorithmus `equal_range`, sondern die von diesen Containern eigens angebotene gleichnamige Methode verwendet werden.

Programm 4.98 – `equalrange.cpp`: Demoprogramm zu equal_range

```
.....
int main(void) {
  int z[] = { 1, 3, 5, 7, 7, 7, 9, 9 };
  copy(z, z+8, ostream_iterator<int>(cout, " ")); cout << endl;
   pair<int *, int*> p;
   p = equal_range(z, z+8, 7);
   cout << "lower_bound fuer 7: ";
   copy(p.first, z+8, ostream_iterator<int>(cout, " ")); cout << endl;
   cout << "upper_bound fuer 7: ";
   copy(p.second, z+8, ostream_iterator<int>(cout, " ")); cout << endl;
   p = equal_range(z, z+8, 4);
   cout << "lower_bound fuer 4: ";
   copy(p.first, z+8, ostream_iterator<int>(cout, " ")); cout << endl;
   cout << "upper_bound fuer 4: ";
   copy(p.second, z+8, ostream_iterator<int>(cout, " ")); cout << endl;
   reverse(z, z+8); //... Reihenfolge umkehren
   copy(z, z+8, ostream_iterator<int>(cout, " ")); cout << endl;
   p = equal_range(z, z+8, 7, greater<int>());
   cout << "lower_bound fuer 7: ";
   copy(p.first, z+8, ostream_iterator<int>(cout, " ")); cout << endl;
   cout << "upper_bound fuer 7: ";
   copy(p.second, z+8, ostream_iterator<int>(cout, " ")); cout << endl;
}
```

Programm 4.98 liefert die folgende Ausgabe:

```
1 3 5 7 7 7 9 9
lower_bound fuer 7: 7 7 7 9 9
upper_bound fuer 7: 9 9
lower_bound fuer 4: 5 7 7 7 9 9
upper_bound fuer 4: 5 7 7 7 9 9
9 9 7 7 7 5 3 1
lower_bound fuer 7: 7 7 7 5 3 1
upper_bound fuer 7: 5 3 1
```

4.6.13 fill und fill_n – Füllen eines Bereichs mit einem Wert

```
#include <algorithm>
```

```
template<typename ForwardIterator, typename T>
void fill(ForwardIterator first, ForwardIterator last, const T& value) {
   for ( ; first != last; ++first)
      *first = value;
}
template<typename OutputIter, typename Size, typename T>
OutputIterator fill_n(OutputIterator first, Size n, const T& value) {
   for ( ; n > 0; --n, ++first)
      *first = value;
   return first;
}
```

`fill` und `fill_n` füllen den Bereich `[first, last)` bzw. `[first, first+n)` mit dem Wert `value` unter Verwendung des entsprechenden Zuweisungsoperators.
Der bei der zweiten Variante zurückgegebene Iterator entspricht `first+n`.

Programm 4.99 – `fill.cpp`:
Demoprogramm zu fill und fill_n

```
#include <vector>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;

int main(void) {
   vector<int> v(7);
   fill(v.begin(), v.end(), 5);
   copy(v.begin(), v.end(), ostream_iterator<int>(cout, " ")); cout << endl;
   fill_n(v.begin()+2, 3, 2);
   copy(v.begin(), v.end(), ostream_iterator<int>(cout, " ")); cout << endl;
}
```

Programm 4.99 liefert die folgende Ausgabe:

```
5 5 5 5 5 5 5
5 5 2 2 2 5 5
```

4.6.14 find – Suchen des ersten Vorkommens eines Werts

```
#include <algorithm>
```

```
template<typename InputIterator, typename T> inline
InputIterator find(InputIterator first, InputIterator last, const T& val) {
   while (first != last && !(*first == val))
      ++first;
   return first;
}
```

`find` gibt einen Iterator auf das erste Elemente im Bereich `[first, last)` zurück, das gleich dem Wert `value`. Sollte kein solches Element gefunden werden, wird `last` zurückgegeben. Zum Vergleichen der einzelnen Werte verwendet `find` den `operator==`, der natürlich für den Typ `T` definiert sein muss.

Programm 4.100 – `find.cpp`:
Demoprogramm zu find

```
#include <vector>
#include <algorithm>
#include <iterator>
#include <iomanip>
#include <iostream>
using namespace std;
int main(void) {
   vector<char> z;
   for (int i=1; i<=20; i++)
      z.push_back(char('A'+rand()%7));
   vector<char>::iterator it = z.begin();
   do {
      cout << setw(it-z.begin()) << "";
      copy(it, z.end(), ostream_iterator<char>(cout, "")); cout << endl;
      it++;
   } while ( (it = find(it, z.end(), 'C')) != z.end());
}
```

Programm 4.100 liefert z. B. die folgende Ausgabe:

```
BECFBDDCBDCFGEGADBCC
  CFBDDCBDCFGEGADBCC
       CBDCFGEGADBCC
          CFGEGADBCC
                  CC
                   C
```

4.6.15 find_end – Suchen des letzten Vorkommens eines Bereichs

```
                                                    #include <algorithm>
```

```
//... 1. Variante: verwendet operator==()
template<typename ForwardIterator1, typename ForwardIterator2>
ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1,
                          ForwardIterator2 first2, ForwardIterator2 last2);
//... 2. Variante: verwendet comp zum Vergleichen
template<typename ForwardIterator1, typename ForwardIterator2,
         typename BinaryPredicate>
ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1,
                          ForwardIterator2 first2, ForwardIterator2 last2,
                          BinaryPredicate comp);
```

`find_end` gibt einen Iterator auf das letzte Vorkommen des Bereichs `[first2, last2)` im Bereich `[first1, last1)` zurück. Sollte kein solcher Teilbereich gefunden werden, wird `last1` zurückgegeben. Bei der zweiten Varianten wird statt `operator==` das zweistellige Funktionsobjekt `comp` zum Vergleichen verwendet.

Programm 4.101 – `findend.cpp`:
Demoprogramm zu find_end

```
#include <algorithm>
#include <iterator>
#include <iomanip>
#include <iostream>
using namespace std;
class Doppelt {
  public:
     bool operator() (int erst, int zweit)  return erst == zweit*2;
};
int main(void) {
   int z1[] = { 2, 4, 6, 8, 9, 5, 6, 7 },
       z2[] = { 8, 9, 5 },
       z3[] = { 2, 3, 4 }, *zgr;
   copy(z1, z1+8, ostream_iterator<int>(cout, " ")); cout << endl;
   zgr = find_end(z1, z1+8, z2, z2+3);
   cout << setw((zgr-z1)*2) << "";
   copy(zgr, zgr+3, ostream_iterator<int>(cout, " ")); cout << endl;
   zgr = find_end(z1, z1+8, z3, z3+3, Doppelt()); // Vergleich auf doppelten Wert
   cout << setw((zgr-z1)*2) << "";
   copy(zgr, zgr+3, ostream_iterator<int>(cout, " ")); cout << endl;
}
```

Programm 4.101 liefert die folgende Ausgabe:

```
2 4 6 8 9 5 6 7
      8 9 5
  4 6 8
```

4.6.16 find_first_of – Suchen eines Elements aus anderem Bereich

```
#include <algorithm>
```

```
//... 1. Variante: verwendet operator==()
template<typename InputIter, typename ForwardIter>
InputIterator find_first_of(InputIterator first1, InputIterator last1,
                            ForwardIterator first2, ForwardIterator last2) {
   for ( ; first1 != last1; ++first1)
      for (ForwardIterator iter = first2; iter != last2; ++iter)
         if (*first1 == *iter)
            return first1;
   return last1;
}
//... 2. Variante: verwendet comp zum Vergleichen
template<typename InputIterator, typename ForwardIterator,
         typename BinaryPredicate>
InputIterator find_first_of(InputIterator first1, InputIterator last1,
                            ForwardIterator first2, ForwardIterator last2,
                            BinaryPredicate comp) {
   for ( ; first1 != last1; ++first1)
     for (ForwardIterator iter = first2; iter != last2; ++iter)
        if (comp(*first1, *iter))
           return first1;
      return last1;
}
```

`find_first_of` gibt einen Iterator auf das erste Vorkommen eines Elements aus dem Bereich `[first2, last2)` im Bereich `[first1, last1)` zurück. Sollte kein solches Element gefunden werden, wird `last1` zurückgegeben.
Bei der zweiten Varianten wird statt `operator==` das zweistellige Funktionsobjekt `comp` zum Vergleichen verwendet.

Programm 4.102 – `findfirstof.cpp`: Demoprogramm zu find_first_of

```
#include <algorithm>
#include <iterator>
#include <iomanip>
#include <iostream>
using namespace std;
class Doppelt {
  public:
     bool operator() (int erst, int zweit) { return erst == zweit*2; }
};
int main(void) {
   int z1[] = { 2, 4, 6, 8, 9, 5, 6, 7 },
       z2[] = { 7, 5, 6 },
       z3[] = { 9, 4 };
   int *zgr;
```

```
   copy(z1, z1+8, ostream_iterator<int>(cout, " ")); cout << endl;

   zgr = find_first_of(z1, z1+8, z2, z2+3);
   cout << setw((zgr-z1)*2) << "";
   copy(zgr, zgr+1, ostream_iterator<int>(cout, " ")); cout << endl;
   zgr = find_first_of(z1, z1+8, z3, z3+2, Doppelt()); // Vergleich auf dopp. Wert
   cout << setw((zgr-z1)*2) << "";
   copy(zgr, zgr+1, ostream_iterator<int>(cout, " ")); cout << endl;
}
```

Programm 4.102 liefert die folgende Ausgabe:

```
2 4 6 8 9 5 6 7
    6
      8
```

4.6.17 find_if – Suchen nach Element, das eine Bedingung erfüllt

```
                                                    #include <algorithm>
```

```
template<typename InputIterator, typename Predicate> inline
InputIterator find_if(InputIterator first, InputIterator last, Predicate pred)
{
   while (first != last && !pred(*first))
      ++first;
   return first;
}
```

`find_if` sucht in dem Bereich `[first, last)` nach dem ersten Element, für das `pred` den Wert `true` liefert. Wird ein solches Element gefunden, wird ein Iterator auf dieses Element zurückgegeben, ansonsten wird `last` zurückgegeben.

Programm 4.103 – `findif.cpp`:
Demoprogramm zu find_if

```
#include <algorithm>
#include <iterator>
#include <iomanip>
#include <iostream>
using namespace std;

template<typename T> // T muss ganzzsahliger Typ sein
class QuerSum : public unary_function<T,bool>
{
  public:
     QuerSum(int z) : summe(z) {}
     bool operator()(T z) { return summe == qSum(z); }

  private:
     const int summe;
```

```
     int qSum(int z) {
        int sum = 0;
        while (z > 0) {
           sum += z%10;
           z /= 10;
        }
        return sum;
     }
};

int main(void) {
   int z1[] = { 88, 20, 41, 63, 91, 64, 123 };
   int *zgr;
   copy(z1, z1+7, ostream_iterator<int>(cout, " ")); cout << endl;

   zgr = find_if(z1, z1+7, bind2nd(greater<double>(), 90));
   cout << setw((zgr-z1)*3) << "";
   copy(zgr, zgr+1, ostream_iterator<int>(cout, " ")); cout << endl;

   zgr = find_if(z1, z1+7, QuerSum<int>(9));
   cout << setw((zgr-z1)*3) << "";
   copy(zgr, zgr+1, ostream_iterator<int>(cout, " ")); cout << endl;
}
```

Programm 4.103 liefert die folgende Ausgabe:

```
88 20 41 63 91 64 123
            91
         63
```

4.6.18 for_each – Aufruf einer Funktion für mehrere Elemente

```
                                                    #include <algorithm>
```

```
template<typename InputIter, typename Function>
Function for_each(InputIterator first, InputIterator last, Function f) {
   for ( ; first != last; ++first)
      f(*first);
   return f;
}
```

Für jedes Element im Bereich `[first, last)` wird die Funktion `f` einmal aufgerufen. Sollte `f` einen Rückgabewert liefern, wird dieser von `for_each` ignoriert.
Die Funktion `f` muss mit einem Argument vom Typ `InputIterator::value_type` aufrufbar sein. Um Schwierigkeiten zu vermeiden, sollte die Funktion `f` die Elemente nicht modifizieren, da Input-Iteratoren nur gelesen werden können. Liefert der Iterator allerdings L-Werte, kann die Funktion `f` ihr Argument auch verändern, wobei hier eventuell der Algorithmus `transform` die bessere Wahl ist.

Programm 4.104 – `foreach.cpp`:
Demoprogramm zu for_each

```
#include <cctype>
#include <string>
#include <functional>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;

void gerade(int z) { cout << ((z % 2 == 0) ? "-- " : "   "); }
template<typename T>
class Add : public unary_function<T,void> {
  public:
     Add(const T& arg = T()) : summe(arg) {}
     void operator() (const T& arg) { summe += arg; }
     T get() const { return summe; }
  private:
     T summe;
};
void klein(char &z) { z = static_cast<char>(tolower(z)); }

void anfangGross(const string& s) {
   string tmp(s);
   for_each(tmp.begin(), tmp.end(), klein);
   tmp[0] = toupper(tmp[0]);
   cout << tmp << " ";
}
int main(void) {
   int z[] = { 11, 20, 40, 61, 90 };
   copy(z, z+5, ostream_iterator<int>(cout, " ")); cout << endl;

   for_each(z, z+5, gerade);

   Add<int> sum = for_each(z, z+5, Add<int>());
   cout << endl << "Summe: " << sum.get() << endl
                << "Summe: " << for_each(z, z+5, Add<int>()).get() << endl;

   string worte[] = { "das", "IST", "dAS", "HauS", "vom", "nIkOLaus" };
   for_each(worte, worte+6, anfangGross)("eNDE"); cout << endl;
}
```

Programm 4.104 liefert die folgende Ausgabe:

```
11 20 40 61 90
   -- --    --
Summe: 222
Summe: 222
Das Ist Das Haus Vom Nikolaus Ende
```

4.6.19 generate und generate_n – Füllen mit Funktionsobjekt

```
#include <algorithm>
```

```
template<typename ForwardIterator, typename Generator>
void generate(ForwardIterator first, ForwardIterator last, Generator gen) {
   for ( ; first != last; ++first)
      *first = gen();
}
template<typename OutputIter, typename Size, typename Generator>
OutputIterator generate_n(OutputIterator first, Size n, Generator gen) {
   for ( ; n > 0; --n, ++first)
      *first = gen();
   return first;
}
```

Während `fill` und `fill_n` den Bereich `[first, last)` bzw. `[first, first+n)` mit einem festen Wert füllen, verwenden `generate` und `generate_n` das übergebene Funktionsobjekt zum Füllen. Zum Füllen wird dabei immer der jeweilige Rückgabewert des Funktionsobjekts beim entsprechenden Element verwendet.

Programm 4.105 – `generate.cpp`:
Demoprogramm zu generate und generate_n

```
#include <cstdlib>
#include <ctime>
#include <vector>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;
class Ungerade {
  public:
     Ungerade(int anfang = 1) : zaehler(anfang) {}
     int operator() () {
        int n = zaehler++ * 2 -1;
        return n;
     }
  private:
     int zaehler;
};
int main(void) {
  vector<int> v(10);
   generate(v.begin(), v.begin()+6, Ungerade());
   generate_n(v.begin()+6, 4, Ungerade(7));
   copy(v.begin(), v.end(), ostream_iterator<int>(cout, " ")); cout << endl;
   srand(time(0));
   vector<int> zuf(6);
   //... rand() % 10 +1 fuer jedes Element
   generate_n(zuf.begin(), zuf.size(), rand);
   copy(zuf.begin(), zuf.end(), ostream_iterator<int>(cout, " ")); cout << endl;
```

```
   transform(zuf.begin(), zuf.end(), zuf.begin(), bind2nd(modulus<int>(), 10) );
   copy(zuf.begin(), zuf.end(), ostream_iterator<int>(cout, " ")); cout << endl;
   transform(zuf.begin(), zuf.end(), zuf.begin(), bind2nd(plus<int>(), 1) );
   copy(zuf.begin(), zuf.end(), ostream_iterator<int>(cout, " ")); cout << endl;
}
```

Programm 4.105 liefert die folgende Ausgabe:

```
1 3 5 7 9 11 13 15 17 19
114599252 1932765305 964640140 1807939143 2069497652 1245119773
2 5 0 3 2 3
3 6 1 4 3 4
```

4.6.20 includes – Prüfen, ob Bereich Teilmenge eines anderen ist

```
                                                          #include <algorithm>
```

```
//... 1. Variante: verwendet operator<()
template<typename InputIterator1, typename InputIterator2>
bool includes(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2) {
   while (first1 != last1 && first2 != last2)
      if (*first2 < *first1)
         return false;
      else if(*first1 < *first2)
         ++first1;
      else
        ++first1, ++first2;
   return first2 == last2;
}
//... 2. Variante: verwendet comp zum Vergleichen
template<typename InputIter1, typename InputIter2, typename Compare>
bool includes(InputIter1 first1, InputIter1 last1,
              InputIter2 first2, InputIter2 last2, Compare comp) {
   while (first1 != last1 && first2 != last2)
      if (comp(*first2, *first1))
         return false;
      else if(comp(*first1, *first2))
         ++first1;
      else
         ++first1, ++first2;
   return first2 == last2;
}
```

`includes` liefert `true`, wenn für jedes Element des Bereichs `[first2, last2)` ein äquivalentes Element im Bereich `[first1, last1)` enthalten ist. Bei der zweiten Variante wird statt dem `operator<` zum Vergleichen das Funktionsobjekt `comp` verwendet. Beide Bereiche müssen dabei sortiert sein.

Programm 4.106 – `includes.cpp`:
Demoprogramm zu includes

```
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;

int main(void)
{
   int z1[] = { 1, 2, 3, 4, 5, 6 },
       z2[] = { 2, 3, 6 },
       z3[] = { 6, 8, 9 };

   cout << "z2 ist in z1 " << (includes(z1, z1+6, z2, z2+3) ? "" : "nicht ")
        << "enthalten" << endl;

   cout << "z3 ist in z1 " << (includes(z1, z1+6, z3, z3+3) ? "" : "nicht ")
        << "enthalten" << endl;

   cout << "z3 ist in z1 "
        << (includes(z1, z1+6, z3, z3+3, equal_to<int>()) ? "" : "nicht ")
        << "enthalten (bei Vergleich mit equal_to)" << endl;

   cout << "z3 ist in z1 "
        << (includes(z1, z1+6, z3, z3+3, greater<int>()) ? "" : "nicht ")
        << "enthalten (bei Vergleich mit greater)" << endl;
}
```

Programm 4.106 liefert die folgende Ausgabe:

```
z2 ist in z1 enthalten
z3 ist in z1 nicht enthalten
z3 ist in z1 enthalten (bei Vergleich mit equal_to)
z3 ist in z1 nicht enthalten (bei Vergleich mit greater)
```

4.6.21 inner_product – Summe der Produkte zweier Bereiche

```
                                                    #include <numeric>
```

```
//... 1. Variante: verwendet operator+() und operator*()
template<typename InputIterator1, typename InputIterator2, typename T>
T inner_product(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, T init)
{
   for ( ; first1 != last1; ++first1, ++first2)
      init = init + (*first1 * *first2);
   return init;
}
```

```
//... 2. Variante: verwendet binary_op1 statt operator+()
                   binary_op2 statt operator*()
template<typename InputIterator1, typename InputIterator2, typename T,
         typename BinaryOperation1, typename BinaryOperation2>
T inner_product(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, T init,
                BinaryOperation1 binary_op1,
                BinaryOperation2 binary_op2) {
   for ( ; first1 != last1; ++first1, ++first2)
      init = binary_op1(init, binary_op2(*first1, *first2));
   return init;
}
```

`inner_product` liefert die Summe der Produkte der korrspondierenden Elemente aus dem Bereich `[first1, last1)` und dem Bereich ab `first2`. Auf diese Summe wird dabei noch der Startwert `init`, der den Typ des Rückgabewerts festlegt, aufaddiert.
Die zweite Variante benutzt statt der Summe das Funktionsobjekt `binary_op1` und statt dem Produkt das Funktionsobjekt `binary_op2`.

Programm 4.107 – `innerproduct.cpp`:
Demoprogramm zu inner_product

```
#include <numeric>
#include <iterator>
#include <iostream>
using namespace std;

int main(void) {
   int z1[] = { 1, 2, 3 },
       z2[] = { 2, 3, 6 };

   cout << "Skalarprod. von z1 + z2: " << inner_product(z1, z1+3, z2, 0) <<endl;
   cout << "....... mit Startw. 100: " << inner_product(z1, z1+3, z2, 100) <<endl;

   //... entspricht ersten Aufruf
   cout << inner_product(z1, z1+3, z2, 0, plus<int>(), multiplies<int>()) <<endl;
   //... z1[i] + z2[i] * ...
   cout << inner_product(z1, z1+3, z2, 1, multiplies<int>(), plus<int>()) <<endl;
   cout << inner_product(z1, z1+3, z2, 10, multiplies<int>(), plus<int>()) <<endl;
   cout << inner_product(z1, z1+3, z2, 0, multiplies<int>(), plus<int>()) <<endl;
}
```

Programm 4.107 liefert die folgende Ausgabe:

```
Skalarprod. von z1 + z2: 26
....... mit Startw. 100: 126
26
135
1350
0
```

4.6.22 inplace_merge – Mischen sortierter Nachbar-Bereiche

```
#include <algorithm>
```

```
//... 1. Variante: verwendet operator<()
template<typename BidirectionalIterator>
void inplace_merge(BidirectionalIterator first, BidirectionalIterator middle,
                   BidirectionalIterator last);
//... 2. Variante: verwendet comp zum Vergleichen
template<typename BidirectionalIterator, typename Compare>
void inplace_merge(BidirectionalIterator first, BidirectionalIterator middle,
                   BidirectionalIterator last, Compare comp);
```

`inplace_merge` mischt die aufeinanderfolgenden, sortierten Bereiche `[first, middle)` und `[middle, last)`, so dass die Elemente in diesem Bereich `[first, last)` danach sortiert sind.
Bei der zweiten Variante wird statt dem `operator<` zum Vergleichen das Funktionsobjekt `comp` verwendet.

Programm 4.108 – `inplacemerge.cpp`:
Demoprogramm zu inplace_merge

```
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;
class nachKomma {
  public:
     bool operator() (double eins, double zwei) const {
        int a = int(eins),
            b = int(zwei);
        return (eins - a) - (zwei - b);
     }
};
int main(void) {
  int  z1[] = { 1, 3, 5, 2, 4, 6, 7 };
   inplace_merge(z1, z1+3, z1+7);
   copy(z1, z1+7, ostream_iterator<int>(cout, " ")); cout << endl;
  int  z2[] = { 7, 6, 4, 2, 5, 3, 1 };
   inplace_merge(z2, z2+4, z2+7, greater<int>());
   copy(z2, z2+7, ostream_iterator<int>(cout, " ")); cout << endl;
   double d[] = { 12.4, 14.5, 17.6, 18.7, 19.1, 20.2, 22.3 };
   inplace_merge(d, d+4, d+7, nachKomma());
   copy(d, d+7, ostream_iterator<double>(cout, " ")); cout << endl;
}
```

Programm 4.108 liefert die folgende Ausgabe:

```
1 2 3 4 5 6 7
7 6 5 4 3 2 1
19.1 20.2 22.3 12.4 14.5 17.6 18.7
```

4.6.23 iter_swap – Tauschen der Werte, auf die Iteratoren zeigen

```
                                                    #include <algorithm>
```

```
template<typename ForwardIterator1, typename ForwardIterator2> inline
void iter_swap(ForwardIterator1 a, ForwardIterator2 b) {
   typedef typename iterator_traits<ForwardIterator1>::value_type ValueType1;
   typedef typename iterator_traits<ForwardIterator2>::value_type ValueType2;

   ValueType1 tmp = *a;
   *a = *b;
   *b = tmp;
}
```

Bei zwei Iteratoren, die auf den gleichen Elementtyp zeigen, entspricht ein `iter_swap`-Aufruf einem Aufruf von `swap`.

Programm 4.109 – `iterswap.cpp`: Demoprogramm zu iter_swap

```
#include <vector>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;
int main(void) {
   string s1[] = { "Eins", "Zwei", "Drei"  },
          s2[] = { "One",  "Two",  "Three" };
   cout << "Am Anfang:" << endl;
   cout << "  s1: ";
   copy(s1, s1+3, ostream_iterator<string>(cout, ", ")); cout << endl;
   cout << "  s2: ";
   copy(s2, s2+3, ostream_iterator<string>(cout, ", ")); cout << endl;
   for (unsigned i=0; i < 3; i++)
      iter_swap(s1+i, s2+i);
   cout << "Nach iter_swap:" << endl;
   cout << "  s1: ";
   copy(s1, s1+3, ostream_iterator<string>(cout, ", ")); cout << endl;
   cout << "  s2: ";
   copy(s2, s2+3, ostream_iterator<string>(cout, ", ")); cout << endl;
}
```

Programm 4.109 liefert die folgende Ausgabe:

```
Am Anfang:
  s1: Eins, Zwei, Drei,
  s2: One, Two, Three,
Nach iter_swap:
  s1: One, Two, Three,
  s2: Eins, Zwei, Drei,
```

4.6.24 lexicographical_compare – Vergleichen zweier Bereiche

```
#include <algorithm>
```

```
//... 1. Variante: verwendet operator<()
template<typename InputIter1, typename InputIter2>
bool lexicographical_compare(InputIter1 first1, InputIter1 last1,
                             InputIter2 first2, InputIter2 last2) {
   for ( ; first1 != last1 && first2 != last2 ; ++first1, ++first2) {
      if (*first1 < *first2)
         return true;
      if (*first2 < *first1)
         return false;
   }
   return first1 == last1 && first2 != last2;
}
//... 2. Variante: verwendet comp zum Vergleichen
template<typename InputIter1, typename InputIter2, typename Compare>
bool lexicographical_compare(InputIter1 first1, InputIter1 last1,
                             InputIter2 first2, InputIter2 last2,
                             Compare comp) {
   for ( ; first1 != last1 && first2 != last2 ; ++first1, ++first2) {
      if (comp(*first1, *first2))
         return true;
      if (comp(*first2, *first1))
         return false;
   }
   return first1 == last1 && first2 != last2;
}
```

`lexicographical_compare` liefert `true`, wenn die Elemente im Bereich `[first1, last1)` lexikographisch kleiner als die Elemente des Bereichs `[first2, last2)` sind. Bei der zweiten Variante wird statt dem `operator<` zum Vergleichen das Funktionsobjekt `comp` verwendet.

Programm 4.110 – `lexicompare.cpp`: Demoprogramm zu lexicographical_compare

```
#include <string>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;
int main(void) {
   string s1 = "Hans",
          s2 = "Helmut",
          s3 = "Hansi";
   cout << s2 << (lexicographical_compare(s2.begin(), s2.end(),
                                          s1.begin(), s1.end()) ? " < " : " > ")
        << s1 << endl;
```

```
   cout << s1 << (lexicographical_compare(s1.begin(), s1.end(),
                                          s3.begin(), s3.end()) ? " < " : " > ")
        << s3 << endl;
   const int    z1[] = { 1,   4,   8   };
   const double z2[] = { 1.2, 6.3, 9.5 };
   cout << "z1"<< (lexicographical_compare(z1, z1+3, z2, z2+3) ? " < " : " > ")
        << "z2" << endl;
   cout << "z2"<< (lexicographical_compare(z2, z2+3, z1, z1+3,
                                           greater<double>()) ? " > " : " < ")
        << "z1" << endl;
}
```

Programm 4.110 liefert die folgende Ausgabe:

```
Helmut > Hans
Hans < Hansi
z1 < z2
z2 > z1
```

4.6.25 lower_bound – Erste Einfügeposition ohne Umsortierung

```
                                                          #include <algorithm>
```

```
//... 1. Variante: verwendet operator<()
template<typename ForwardIterator, typename T>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                            const T& val);
//... 2. Variante: verwendet comp zum Vergleichen
template<typename ForwardIter, typename T, typename Compare>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                            const T& val, Compare comp);
```

`lower_bound` liefert die erste Position aus dem Bereich `[first, last]` (`last` gehört hier dazu), an der `val` eingefügt werden kann, ohne dass dadurch die Sortierung verletzt wird. Wird kein solches Element gefunden, liefert `lower_bound` als Rückgabewert `last`.

Der Bereich muss bei der ersten Variante mit dem `operator<` und bei der zweiten Variante mit dem Funktionsobjekt `comp` sortiert sein.

Da die assoziativen Container Bidirectional-Iteratoren anbieten, sollte für sie nicht der Algorithmus `lower_bound`, sondern die von diesen Containern eigens angebotene gleichnamige Methode verwendet werden.

Programm 4.111 – `lowerbound.cpp`: Demoprogramm zu lower_bound

```
#include <algorithm>
#include <iterator>
#include <iomanip>
#include <iostream>
```

```
using namespace std;

int main(void) {
   int z[] = { 1, 3, 5, 7, 7, 7, 8, 8 };
   int *p;
   cout << " ";
   copy(z, z+8, ostream_iterator<int>(cout, " ")); cout << endl;
   p = lower_bound(z, z+8, 7); cout << setw((p-z)*2+1) << "7" << endl;
   p = lower_bound(z, z+8, 4); cout << setw((p-z)*2+1) << "4" << endl;
   p = lower_bound(z, z+8, 9); cout << setw((p-z)*2+1) << "9" << endl;
   reverse(z, z+8); //... Reihenfolge umkehren
   cout << " ";
   copy(z, z+8, ostream_iterator<int>(cout, " ")); cout << endl;
   p = lower_bound(z, z+8, 7, greater<int>());
   cout << setw((p-z)*2+1) << "7" << endl;
   p = lower_bound(z, z+8, 9, greater<int>());
   cout << setw((p-z)*2+1) << "9" << endl;
}
```

Programm 4.111 liefert die folgende Ausgabe:

```
 1 3 5 7 7 7 8 8
      7
    4
                9
 8 8 7 7 7 5 3 1
    7
9
```

4.6.26 max – Größeres von zwei Objekten

```
                                                    #include <algorithm>
```

```
//... 1. Variante: verwendet operator<()
template<typename T> inline
const T& max(const T& a, const T& b) {
   return (a < b) ? b : a;
}

//... 2. Variante: verwendet comp zum Vergleichen
template<typename T, typename Compare> inline
const T& max(const T& a, const T& b, Compare comp) {
   return comp(a, b) ? b : a;
}
```

`max` liefert das Größere der beiden Argumente als Rückgabewert. Sind beide Argumente gleich, wird das erste zurückgegeben.
Bei der zweiten Variante wird statt `operator<` das Funktionsobjekt `comp` zum Vergleichen verwendet.

Programm 4.112 – `max.cpp`:
Demoprogramm zu max

```
#include <string>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;

int main(void)
{
   string s1 = "Hans",
          s2 = "Helmut",
          s3 = "Hansi";

   string s = max(s1, s2);
   cout << s1 << ((s == s1) ? " > " : " < ") << s2 << endl;
   s = max(s1, s3);
   cout << s1 << ((s == s1) ? " > " : " < ") << s3 << endl;

   s = max(s2, s1, greater<string>());
   cout << s2 << ((s == s2) ? " < " : " > ") << s1 << endl;
   s = max(s3, s1, greater<string>());
   cout << s3 << ((s == s3) ? " < " : " > ") << s1 << endl;
}
```

Programm 4.112 liefert die folgende Ausgabe:

```
Hans < Helmut
Hans < Hansi
Helmut > Hans
Hansi > Hans
```

4.6.27 max_element – Iterator auf größtes Element eines Bereichs

```
                                                    #include <algorithm>
```

```
//... 1. Variante: verwendet operator<()
template<typename ForwardIterator>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last) {
   if (first == last)
      return first;
   ForwardIterator result = first;
   while (++first != last)
      if (*result < *first)
         result = first;
   return result;
}
```

```
//... 2. Variante: verwendet comp zum Vergleichen
template<typename ForwardIterator, typename Compare>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last,
                            Compare comp) {
   if (first == last)
      return first;
   ForwardIterator result = first;
   while (++first != last)
      if (comp(*result, *first))
         result = first;
   return result;
}
```

`max_element` liefert einen Iterator auf den größten Wert im Bereich `[first,last)`. Bei der zweiten Variante wird statt `operator<` das Funktionsobjekt `comp` zum Vergleichen verwendet.

Programm 4.113 – `maxelement.cpp`: Demoprogramm zu max_element

```
#include <algorithm>
#include <iterator>
#include <iomanip>
#include <iostream>
using namespace std;
class Abs {
  public:
     bool operator() (int eins, int zwei) { return abs(eins) < abs(zwei); }
};
int main(void) {
   int z[] = { 2, -5, 7, -3, 7, -9, 7 };
   int *p;

   cout << showpos << " ";
   copy(z, z+7, ostream_iterator<int>(cout, " ")); cout << endl;

   p = max_element(z, z+7);         cout << setw((p-z+1)*3) << "--" << endl;
   p = max_element(z, z+7, Abs());  cout << setw((p-z+1)*3) << "--" << endl;

   p = max_element(z, z+7, greater<int>());
   cout << setw((p-z+1)*3) << "--" << endl;
}
```

Programm 4.113 liefert die folgende Ausgabe:

```
 +2 -5 +7 -3 +7 -9 +7
       --
                --
                --
```

4.6.28 merge – Mischen zweier sortierter Bereiche

```
#include <algorithm>
```

```
//... 1. Variante: verwendet operator<()
template<typename InputIterator1, typename InputIterator2, typename OutputIterator>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result);
//... 2. Variante: verwendet comp zum Vergleichen
template<typename InputIterator1, typename InputIterator2, typename OutputIterator,
         typename Compare>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result, Compare comp);
```

`merge` mischt die beiden sortierten Bereiche `[first1,last1)` und `[first2,last2)` und speichert den so gemischten und sortierten Bereich an die Stelle, deren Beginn durch `result` festgelegt wurde. `result` sollte nicht in einen der beiden zu mischenden Bereich zeigen.
Bei der zweiten Variante wird statt dem `operator<` zum Vergleichen das Funktionsobjekt `comp` verwendet.
Als Rückgabe liefert `merge` einen Iterator auf das Ende des resultierenden Bereichs.

Programm 4.114 – `merge.cpp`:
Demoprogramm zu merge

```
#include <vector>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;
class nachKomma {
  public:
     bool operator() (double eins, double zwei) const {
        int a = int(eins),
            b = int(zwei);
        return (eins - a) - (zwei - b);
     }
};
int main(void) {
   vector<int> v;
   vector<int>::iterator it;
   int z1[] = { 1, 2, 2, 3, 5, 6 },
       z2[] = { 2, 3, 4, 4, 6 };
   v.reserve(11); // Speicherplatz reservieren
   it = merge(z1, z1+6, z2, z2+5, v.begin());
   copy(v.begin(), it, ostream_iterator<int>(cout, " ")); cout << endl;
   sort(z1, z1+6, greater<int>());
   sort(z2, z2+5, greater<int>());
```

```
   it = merge(z1, z1+6, z2, z2+5, v.begin(), greater<int>());
   copy(v.begin(), it, ostream_iterator<int>(cout, " ")); cout << endl;
   vector<double> v2;
   vector<double>::iterator it2;
   v2.reserve(7); // Speicherplatz reservieren
   double d1[] = { 12.1, 14.5, 17.6, 18.7 },
          d2[] = { 19.2, 20.4, 22.6 };
   it2 = merge(d1, d1+4, d2, d2+3, v2.begin(), nachKomma());
   copy(v2.begin(), it2, ostream_iterator<double>(cout, ", ")); cout << endl;
}
```

Programm 4.114 liefert die folgende Ausgabe:

```
1 2 2 2 3 3 4 4 5 6 6
6 6 5 4 4 3 3 2 2 2 1
19.2, 20.4, 22.6, 12.1, 14.5, 17.6, 18.7,
```

4.6.29 min – Kleineres von zwei Objekten

```
                                                    #include <algorithm>
```

```
//... 1. Variante: verwendet operator<()
template<typename T> inline
const T& min(const T& a, const T& b) {
   return (b < a) ? b : a;
}
//... 2. Variante: verwendet comp zum Vergleichen
template<typename T, typename Compare> inline
const T& min(const T& a, const T& b, Compare comp) {
   return comp(b, a) ? b : a;
}
```

`min` liefert das Kleinere der beiden Argumente als Rückgabewert. Sind beide Argumente gleich, wird das erste zurückgegeben.
Bei der zweiten Variante wird statt `operator<` das Funktionsobjekt `comp` zum Vergleichen verwendet.

Programm 4.115 – `min.cpp`:
Demoprogramm zu min

```
#include <string>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;
int main(void) {
   string s1 = "Hans",
          s2 = "Helmut",
          s3 = "Hansi";
   string s = min(s1, s2);
   cout << s1 << ((s == s1) ? " < " : " > ") << s2 << endl;
```

```
   s = min(s1, s3);
   cout << s1 << ((s == s1) ? " < " : " > ") << s3 << endl;
   s = min(s2, s1, greater<string>());
   cout << s2 << ((s == s2) ? " > " : " < ") << s1 << endl;
   s = min(s3, s1, greater<string>());
   cout << s3 << ((s == s3) ? " > " : " < ") << s1 << endl;
}
```

Programm 4.115 liefert die folgende Ausgabe:

```
Hans < Helmut
Hans < Hansi
Helmut > Hans
Hansi > Hans
```

4.6.30 min_element – Iterator auf kleinstes Element eines Bereichs

```
                                                    #include <algorithm>
```

```
//... 1. Variante: verwendet operator<()
template<typename ForwardIterator>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last) {
   if (first == last)
      return first;
   ForwardIterator result = first;
   while (++first != last)
      if (*first < *result)
         result = first;
   return result;
}

//... 2. Variante: verwendet comp zum Vergleichen
template<typename ForwardIterator, typename Compare>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last,
                            Compare comp) {
   if (first == last)
      return first;
   ForwardIterator result = first;
   while (++first != last)
      if (comp(*first, *result))
         result = first;
   return result;
}
```

`min_element` liefert einen Iterator auf den kleinsten Wert in dem Bereich `[first, last)`.
Bei der zweiten Variante wird statt `operator<` das Funktionsobjekt `comp` zum Vergleichen verwendet.

Programm 4.116 – `minelement.cpp`:
Demoprogramm zu min_element

```
#include <algorithm>
#include <iterator>
#include <iomanip>
#include <iostream>
using namespace std;

class Abs {
  public:
     bool operator() (int eins, int zwei) { return abs(eins) < abs(zwei); }
};

int main(void) {
   int z[] = { 2, -5, 7, -3, 7, -9, 7 };
   int *p;

   cout << showpos << " ";
   copy(z, z+7, ostream_iterator<int>(cout, " ")); cout << endl;

   p = min_element(z, z+7);         cout << setw((p-z+1)*3) << "--" << endl;
   p = min_element(z, z+7, Abs());  cout << setw((p-z+1)*3) << "--" << endl;

   p = min_element(z, z+7, greater<int>());
   cout << setw((p-z+1)*3) << "--" << endl;
}
```

Programm 4.116 liefert die folgende Ausgabe:

```
 +2 -5 +7 -3 +7 -9 +7
                --
 --
       --
```

4.6.31 mismatch – Erstes verschiedene Wertepaar zweier Bereiche

```
                                                    #include <algorithm>
```

```
//... 1. Variante: verwendet operator==()
template<typename InputIterator1, typename InputIterator2>
pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2) {
   while (first1 != last1 && *first1 == *first2) {
      ++first1;
      ++first2;
   }
   return pair<InputIterator1, InputIterator2>(first1, first2);
}
```

```
//... 2. Variante: verwendet pred zum Vergleichen
template<typename InputIterator1, typename InputIterator2,
         typename BinaryPredicate>
pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2,
         BinaryPredicate pred) {
  while (first1 != last1 && pred(*first1, *first2)) {
     ++first1;
     ++first2;
  }
  return pair<InputIterator1, InputIterator2>(first1, first2);
}
```

`mismatch` liefert ein Iterator-Paar auf die ersten Elemente, bei denen sich der Bereich `[first1, last1)` und der ab `first2` beginnende Bereich unterscheiden.
Bei der zweiten Variante wird statt `operator==` das Funktionsobjekt `pred` zum Vergleichen verwendet.

Programm 4.117 – `mismatch.cpp`:
Demoprogramm zu mismatch

```
#include <algorithm>
#include <iostream>
#include <iomanip>
using namespace std;
int main(void) {
   int z1[] = { 1, 3, 5, 6, 8, 9 },
       z2[] = { 1, 3, 5, 7, 8, 9 },
       z3[] = { 2, 4, 6, 7, 8, 9 };
   pair<int *, int*> p;
   cout << "Erstes verschiedenes Element" << endl;
   copy(z1, z1+6, ostream_iterator<int>(cout, " ")); cout << endl;
   copy(z2, z2+6, ostream_iterator<int>(cout, " ")); cout << endl;
   p = mismatch(z1, z1+6, z2);
   cout << setw((p.first-z1)*2+1) << "!=" << endl;
   cout << "Erstes gleiches Element" << endl;
   copy(z1, z1+6, ostream_iterator<int>(cout, " ")); cout << endl;
   copy(z3, z3+6, ostream_iterator<int>(cout, " ")); cout << endl;
   p = mismatch(z1, z1+6, z3, not_equal_to<int>());
   cout << setw((p.first-z1)*2+1) << "==" << endl;
}
```

Programm 4.117 liefert die folgende Ausgabe:

```
Erstes verschiedenes Element
1 3 5 6 8 9
1 3 5 7 8 9
     !=
Erstes gleiches Element
1 3 5 6 8 9
2 4 6 7 8 9
       ==
```

4.6.32 next_permutation – Nächste Permutation eines Bereichs

```
#include <algorithm>
```

```
//... 1. Variante: verwendet operator<()
template<typename BidirectionalIterator>
bool next_permutation(BidirectionalIterator first, BidirectionalIterator last);
//... 2. Variante: verwendet comp zum Vergleichen
template<typename BidirectionalIterator, typename Compare>
bool next_permutation(BidirectionalIterator first, BidirectionalIterator last,
                      Compare comp);
```

`next_permutation` generiert für den Bereich `[first, last)` die nächste Permutation der Elemente.
Der Bereich muss bei der ersten Variante mit dem `operator<` und bei der zweiten Variante mit dem Funktionsobjekt `comp` lexikographisch sortiert sein.
Wenn eine weitere größere Permutation möglich ist, wird diese generiert, was durch den Rückgabewert `true` angezeigt wird. Ist keine weitere größere Permutation mehr möglich, wird die kleinste Permutation (Elemente nach `operator<` bzw. `comp` sortiert) generiert, und `false` zurückgegeben.

Programm 4.118 – `nextpermut.cpp`: Demoprogramm zu next_permutation

```
....
int main(void) {
   int z1[] = { 1, 2, 3 }, z2[] = { 2, 1, 3 };
   do {
      copy(z1, z1+3, ostream_iterator<int>(cout, " "));
      cout << ", ";
   } while (next_permutation(z1, z1+3));
   copy(z1, z1+3, ostream_iterator<int>(cout, " ")); cout << endl;
   do {
      copy(z2, z2+3, ostream_iterator<int>(cout, " "));
      cout << ", ";
   } while (next_permutation(z2, z2+3));
   copy(z2, z2+3, ostream_iterator<int>(cout, " ")); cout << endl;
   char s[] = { 'C', 'B', 'A' };
   do {
      copy(s, s+3, ostream_iterator<char>(cout, ""));
      cout << ", ";
   } while (next_permutation(s, s+3, greater<char>()));
   copy(s, s+3, ostream_iterator<char>(cout, "")); cout << endl;
}
```

Programm 4.118 liefert die folgende Ausgabe:

```
1 2 3 , 1 3 2 , 2 1 3 , 2 3 1 , 3 1 2 , 3 2 1 , 1 2 3
2 1 3 , 2 3 1 , 3 1 2 , 3 2 1 , 1 2 3
CBA, CAB, BCA, BAC, ACB, ABC, CBA
```

4.6.33 nth_element – Relatives Sortieren des n-ten Element

```
#include <algorithm>
```

```
//... 1. Variante: verwendet operator<=()
template<typename RandomAccessIterator>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                 RandomAccessIterator last);

//... 2. Variante: verwendet comp zum Vergleichen
template<typename RandomAccessIterator, typename Compare>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                 RandomAccessIterator last, Compare comp);
```

`nth_element` ordnet die Elemente des Bereichs `[first, last)` so um, dass an der Position `n` genau das Element steht, das bei einer vollständigen Sortierung aller Elemente auch dort stehen würde.
`nth_element` garantiert, dass alle Elemente vor dem *n*-ten Element kleiner oder gleich dem *n*-ten Element sind, und alle danach größer oder gleich diesem Element sind. Mit dem Funktionsobjekt `comp` kann eine andere Beziehung zum *n*-ten Element festgelegt werden. Die Reihenfolge in den beiden Bereichen vor und nach dem *n*-ten Element ist dabei nicht definiert.

Programm 4.119 – `nthelement.cpp`:
Demoprogramm zu nth_element

```
#include <algorithm>
#include <iterator>
#include <iostream>
#include <iomanip>
using namespace std;
int main(void) {
  int z1[] = { 3, 1, 6, 9, 5, 6, 4, 2, 7, 8 };
   copy(z1, z1+10, ostream_iterator<int>(cout, " ")); cout << endl;
   nth_element(z1, z1+4, z1+10);
   copy(z1, z1+10, ostream_iterator<int>(cout, " ")); cout << endl;
   cout << setw(4*2+1) << z1[4] << endl;
   nth_element(z1, z1+6, z1+10, greater<int>());
   copy(z1, z1+10, ostream_iterator<int>(cout, " ")); cout << endl;
   cout << setw(6*2+1) << z1[6] << endl;
}
```

Programm 4.119 liefert die folgende Ausgabe:

```
3 1 6 9 5 6 4 2 7 8
2 1 3 4 5 6 9 6 7 8
        5
8 7 6 9 6 5 4 3 1 2
            4
```

4.6.34 partial_sort – Sortieren des Anfangs eines Bereichs

```
                                                    #include <algorithm>
```

```
//... 1. Variante: verwendet operator<()
template<typename RandomAccessIterator>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                  RandomAccessIterator last);
//... 2. Variante: verwendet comp zum Vergleichen
template<typename RandomAccessIterator, typename Compare>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                  RandomAccessIterator last, Compare comp);
```

`partial_sort` ordnet die Elemente des Bereichs `[first, last)` so an, dass der vordere Teilbereich `[first, middle)` sortiert ist, während die Reihenfolge der restlichen Elemente im hinteren Teilbereich undefiniert ist. So lassen sich z. B. die „Top 10“ aus einem Bereich mit einer Vielzahl von Elementen nach vorne bringen.
Der vordere Teilbereich wird bei der ersten Variante mit dem `operator<` und bei der zweiten Variante mit dem Funktionsobjekt `comp` sortiert.

Programm 4.120 – `partialsort.cpp`:
Demoprogramm zu partial_sort

```
#include <algorithm>
#include <iostream>
#include <iomanip>
using namespace std;
int main(void) {
   int z1[] = { 3, 1, 6, 9, 5, 6, 4, 2, 7, 8 };
   copy(z1, z1+10, ostream_iterator<int>(cout, " ")); cout << endl;
   partial_sort(z1, z1+3, z1+10);
   copy(z1, z1+10, ostream_iterator<int>(cout, " ")); cout << endl;
   cout << setfill('-') << setw(3*2-1) << "-" << endl;
   partial_sort(z1, z1+10, z1+10);
   copy(z1, z1+10, ostream_iterator<int>(cout, " "));
   cout << " (vollstaendig sortiert)" << endl;
   cout << setw(10*2-1) << "-" << endl;
   partial_sort(z1, z1+6, z1+10, greater<int>());
   copy(z1, z1+10, ostream_iterator<int>(cout, " ")); cout << endl;
   cout << setw(6*2-1) << "-" << endl;
}
```

Programm 4.120 liefert die folgende Ausgabe:

```
3 1 6 9 5 6 4 2 7 8
1 2 3 9 6 6 5 4 7 8
-----
1 2 3 4 5 6 6 7 8 9  (vollstaendig sortiert)
-------------------
9 8 7 6 6 5 1 2 3 4
-----------
```

4.6.35 partial_sort_copy – Sortiertes Kopieren eines Bereichs

```
                                                    #include <algorithm>
```

```
//... 1. Variante: verwendet operator<()
template<typename InputIterator, typename RandomAccessIterator>
RandomAccessIterator partial_sort_copy(InputIterator first, InputIterator last,
                                       RandomAccessIterator result_first,
                                       RandomAccessIterator result_last);
//... 2. Variante: verwendet comp zum Vergleichen
template<typename InputIterator, typename RandomAccessIterator, typename Compare>
RandomAccessIterator partial_sort_copy(InputIterator first, InputIterator last,
                                       RandomAccessIterator result_first,
                                       RandomAccessIterator result_last,
                                       Compare comp);
```

`partial_sort_copy` kopiert die ersten *n* Elemente des Bereichs `[first, last)` so in den Bereich `[result_first, result_first+n)`, als ob der ganze Bereich `[first, last)` sortiert wäre. *n* ist dabei das Minimum von `last - first` und `result_last - result_first`. Zurückgegeben wird `result_first + n`. So lassen sich z. B. die „Top 10“ aus einem Bereich mit einer Vielzahl von Elementen in einen anderen Bereich kopieren. Der kopierte Teilbereich wird bei der ersten Variante mit dem `operator<` und bei der zweiten Variante mit dem Funktionsobjekt `comp` sortiert.

Programm 4.121 – `partsortcopy.cpp`: Demoprogramm zu partial_sort_copy

```
#include <algorithm>
#include <iostream>
using namespace std;
int main(void) {
  int z1[] = { 3, 1, 6, 9, 5, 6, 4, 2, 7, 8 }, z2[5];
   copy(z1, z1+10, ostream_iterator<int>(cout, " ")); cout << endl;
   partial_sort_copy(z1, z1+10, z2, z2+6);
   copy(z2, z2+6, ostream_iterator<int>(cout, " ")); cout << endl;
   partial_sort_copy(z1, z1+5, z2, z2+4);
   copy(z2, z2+6, ostream_iterator<int>(cout, " ")); cout << endl;
   partial_sort_copy(z1, z1+10, z2, z2+6, greater<int>());
   copy(z2, z2+6, ostream_iterator<int>(cout, " ")); cout << endl;
   copy(z1, z1+10, ostream_iterator<int>(cout, " ")); cout << endl;
}
```

Programm 4.121 liefert die folgende Ausgabe:

```
3 1 6 9 5 6 4 2 7 8
1 2 3 4 5 6
1 3 5 6 5 6
9 8 7 6 6 5
3 1 6 9 5 6 4 2 7 8
```

4.6.36 partial_sum – Kopieren der fortlaufenden Summen

```
                                                    #include <numeric>
```

```
//... 1. Variante: verwendet operator+()
template<typename InputIterator, typename OutputIterator>
OutputIterator partial_sum(InputIterator first, InputIterator last,
                           OutputIterator result) {
   typedef typename iterator_traits<_InputIterator>::value_type ValueType;
   if (first == last) return result;
   *result = *first;
   ValueType value = *first;
   while (++first != last) { value = value + *first; *++result = value; }
   return ++result;
}
//... 2. Variante: verwendet binary_op statt operator+()
template<typename InputIterator, typename OutputIterator, typename BinaryOperation>
OutputIterator partial_sum(InputIterator first, InputIterator last,
                           OutputIterator result, BinaryOperation binary_op) {
   typedef typename iterator_traits<InputIterator>::value_type ValueType;
   if (first == last) return result;
   *result = *first;
   ValueType value = *first;
   while (++first != last) { value = binary_op(value, *first); *++result = value; }
   return ++result;
}
```

`partial_sum` summiert nacheinander die Elemente des `[first, last)` auf und speichert die Ergebnisse in den bei `result` beginnenden Bereich.
Die zweite Variante benutzt statt der Summe das Funktionsobjekt `binary_op`.

Programm 4.122 – `partialsum.cpp`:
Demoprogramm zu partial_sum

```
.......
int main(void) {
  int z[] = { 1, 2, 3, 4, 5 };
   copy(z, z+5, ostream_iterator<int>(cout, " + ")); cout << endl;
   partial_sum(z, z+5, ostream_iterator<int>(cout, ",  ")); cout << endl;
   cout << "---------------------" << endl;
   copy(z, z+5, ostream_iterator<int>(cout, " * ")); cout << endl;
   partial_sum(z, z+5, ostream_iterator<int>(cout, ",  "), multiplies<int>());
}
```

Programm 4.122 liefert die folgende Ausgabe:

```
1 + 2 + 3 + 4 + 5 +
1,  3,  6,  10,  15,
---------------------
1 * 2 * 3 * 4 * 5 *
1,  2,  6,  24,  120,
```

4.6.37 partition – Bilden zweier Gruppen mit einer Bedingung

```
#include <algorithm>
```

```
template<typename ForwardIterator, typename Predicate>
ForwardIterator partition(ForwardIterator first, ForwardIterator last,
                          Predicate pred);
```

`partition` verteilt die Elemente des Bereichs `[first, last)` auf zwei Bereiche, so dass die erste Gruppe nur Elemente enthält, die die über `pred` vorgegebene Bedingung erfüllen, und die zweite Gruppe nur solche Elemente, die diese Bedingung nicht erfüllen. Der Rückgabewert ist ein Iterator auf den Anfang der zweiten Gruppe.

Programm 4.123 – `partition.cpp`:
Demoprogramm zu partition

```
#include <algorithm>
#include <iterator>
#include <iostream>
#include <iomanip>
using namespace std;

class Kleiner {
  public:
     Kleiner(int z) : zahl(z) { }
     bool operator() (int wert) { return wert < zahl; }
  private:
     int zahl;
};

int main(void) {
   int *p, z[] = { 3, 1, 6, 8, 5, 6, 4, 2, 7, 9 };

   copy(z, z+10, ostream_iterator<int>(cout, " ")); cout << endl;

   p = partition(z, z+10, bind2nd(modulus<int>(), 2));
   copy(z, z+10, ostream_iterator<int>(cout, " ")); cout << endl;
   cout << setw((p-z)*2) << "|" << endl;
   p = partition(z, z+10, Kleiner(4));
   copy(z, z+10, ostream_iterator<int>(cout, " ")); cout << endl;
   cout << setw((p-z)*2) << "|" << endl;
}
```

Programm 4.123 liefert die folgende Ausgabe:

```
3 1 6 8 5 6 4 2 7 9
3 1 9 7 5 6 4 2 8 6
        |
3 1 2 7 5 6 4 9 8 6
    |
```

4.6.38 prev_permutation – Vorherige Permutation eines Bereichs

```
#include <algorithm>
```

```
//... 1. Variante: verwendet operator<()
template<typename BidirectionalIterator>
bool prev_permutation(BidirectionalIterator first, BidirectionalIterator last);
//... 2. Variante: verwendet comp zum Vergleichen
template<typename BidirectionalIterator, typename Compare>
bool prev_permutation(BidirectionalIterator first, BidirectionalIterator last,
                      Compare comp);
```

`prev_permutation` generiert für den Bereich `[first, last)` die vorhergehende Permutation der Elemente.
Der Bereich muss bei der ersten Variante mit dem `operator<` und bei der zweiten Variante mit dem Funktionsobjekt `comp` lexikographisch sortiert sein.
Wenn eine weitere kleinere Permutation möglich ist, wird diese generiert, was durch den Rückgabewert `true` angezeigt wird. Ist keine weitere kleinere Permutation mehr möglich, wird die größte Permutation (Elemente nach `operator<` bzw. `comp` sortiert) generiert, und `false` zurückgegeben.

Programm 4.124 – `prevpermut.cpp`:
Demoprogramm zu prev_permutation

```
.....
int main(void) {
   int z1[] = { 3, 2, 1 }, z2[] = { 2, 1, 3 };
   do {
      copy(z1, z1+3, ostream_iterator<int>(cout, " "));
      cout << ", ";
   } while (prev_permutation(z1, z1+3));
   copy(z1, z1+3, ostream_iterator<int>(cout, " ")); cout << endl;
   do {
      copy(z2, z2+3, ostream_iterator<int>(cout, " "));
      cout << ", ";
   } while (prev_permutation(z2, z2+3));
   copy(z2, z2+3, ostream_iterator<int>(cout, " ")); cout << endl;
   char s[] = { 'A', 'B', 'C' };
   do {
      copy(s, s+3, ostream_iterator<char>(cout, ""));
      cout << ", ";
   } while (prev_permutation(s, s+3, greater<char>()));
   copy(s, s+3, ostream_iterator<char>(cout, "")); cout << endl;
}
```

Programm 4.124 liefert die folgende Ausgabe:

```
3 2 1 , 3 1 2 , 2 3 1 , 2 1 3 , 1 3 2 , 1 2 3 , 3 2 1
2 1 3 , 1 3 2 , 1 2 3 , 3 2 1
ABC, ACB, BAC, BCA, CAB, CBA, ABC
```

4.6.39 random_shuffle – Zufälliges Umordnen eines Bereichs

```
#include <algorithm>
```

```
//... 1. Variante: verwendet rand() aus Bibliothek
template<typename RandomAccessIterator> inline
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last) {
   if (first == last) return;
   for (RandomAccessIterator i = first + 1; i != last; ++i)
      iter_swap(i, first + (rand() % ((i - first) + 1)));
}
//... 2. Variante: verwendet Funktionsobjekt rand
template<typename RandomAccessIterator, typename RandomNumberGenerator>
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last,
                    RandomNumberGenerator& rand) {
   if (first == last) return;
   for (RandomAccessIterator i = first + 1; i != last; ++i)
      iter_swap(i, first + rand((i - first) + 1));
}
```

`random_shuffle` ordnet die Elemente dem Bereich `[first, last)` zufällig um. Bei der zweiten Variante wird statt `rand()` aus der Standardbibliothek das Funktionsobjekt `rand` als Zufallszahlengenerator verwendet, das immer mit einem `int`-Argument `n` aufgerufen wird und einen Zufallswert aus dem Intervall `[0, n)` liefern muss.

Programm 4.125 – `randomshuffle.cpp`: Demoprogramm zu random_shuffle

```
.......
class MyRand
{
     unsigned zahl;
  public:
     MyRand(int start = 1) : zahl(start) { }
     unsigned operator() (unsigned n) { return (zahl = zahl*1103515245+12345) % n; }
};
int main(void)
{
   string s[] = { "Adam", "Berta", "Micha", "Robert", "Zorro" };
   copy(s, s+5, ostream_iterator<string>(cout, " ")); cout << endl;
   srand(time(0));
   random_shuffle(s, s+5);
   copy(s, s+5, ostream_iterator<string>(cout, " ")); cout << endl;
   random_shuffle(s, s+5);
   copy(s, s+5, ostream_iterator<string>(cout, " ")); cout << endl;
   MyRand meinZuf(time(0)+1);
   random_shuffle(s, s+5, meinZuf);
   copy(s, s+5, ostream_iterator<string>(cout, " ")); cout << endl;
   random_shuffle(s, s+5, meinZuf);
   copy(s, s+5, ostream_iterator<string>(cout, " ")); cout << endl;
}
```

Programm 4.125 liefert z. B. die folgende Ausgabe:

```
Adam Berta Micha Robert Zorro
Zorro Micha Adam Berta Robert
Adam Berta Zorro Micha Robert
Micha Zorro Adam Robert Berta
Berta Micha Zorro Adam Robert
```

4.6.40 remove – Entfernen von bestimmten Wert

```
#include <algorithm>
```

```
template<typename ForwardIterator, typename T>
ForwardIterator remove(ForwardIterator first, ForwardIterator last, const T& val)
```

`remove` „entfernt“ in dem Bereich `[first, last)` alle Elemente, die einen Wert `val` haben. Die Elemente werden dabei nicht wirklich entfernt, sondern durch nach vorne gezogene Elemente, die nicht den Wert `val` besitzen, überschrieben. Die Größe des Bereichs wird durch `remove` nicht verändert.
Als Rückgabewert liefert `remove` einen Iterator auf `last - n`, wenn `n` Elemente entfernt wurden.
Um die Elemente wirklich aus dem entsprechenden Bereich zu entfernen, ist Folgendes von Wichtigkeit:

- Um bei `vector`, `deque` und `string` die Elemente wirklich zu entfernen, kann man die Methode `erase` verwenden.
- `list` stellt eine eigene Methode `remove` zur Verfügung, die anders als der Algorithmus `remove` die entsprechenden Elemente wirklich aus der Liste entfernt und nicht nur überschreibt.
- Die assoziativen Containern bieten zum Löschen von Elementen die Methode `erase` an.

Programm 4.126 – `remove.cpp`: Demoprogramm zu remove

```
.....
int main(void) {
   int  z[] = { 3, 2, 0, 1, 3, 1, 2, 0 };
   vector<int> v(z, z+8);
   vector<int>::iterator it;
   copy(v.begin(), v.end(), ostream_iterator<int>(cout, " ")); cout << endl;
   for (int i = 0; i < 4; i++) {
      it = remove(v.begin(), v.end(), i);
      copy(v.begin(), it, ostream_iterator<int>(cout, " ")); cout << " |  ";
      copy(it, v.end(), ostream_iterator<int>(cout, " "));
      cout << "... " << i << " geloescht" << endl;
      v.erase(it, v.end()); // Elemente wirlich entfernen
   }
}
```

Programm 4.126 liefert die folgende Ausgabe:

```
3 2 0 1 3 1 2 0
3 2 1 3 1 2  |  2 0 ... 0 geloescht
3 2 3 2  |  1 2 ... 1 geloescht
3 3  |  3 2 ... 2 geloescht
 |  3 3 ... 3 geloescht
```

4.6.41 remove_copy – Kopieren aller Elemente, die keinen bestimmten Wert besitzen

```
                                                    #include <algorithm>
```

```
template<typename InputIterator, typename OutputIterator, typename T>
OutputIterator remove_copy(InputIterator first, InputIterator last,
                           OutputIterator result, const T& value) {
   for ( ; first != last; ++first)
      if (!(*first == value)) { *result = *first; ++result; }
   return result;
}
```

`remove_copy` kopiert alle Elemente aus dem Bereich `[first, last)`, die nicht den Wert `value` besitzen, in den bei `result` beginnenden Bereich. `result` sollte nicht in dem Bereich `[first, last)` liegen, da sonst eventuell Elemente schon vor dem Kopieren überschrieben werden. Der Rückgabewert ist ein Iterator hinter das letzte Element des Bereichs, in den kopiert wurde.
Dieser Algorithmus ist stabil, was bedeutet, dass die relative Reihenfolge der Elemente zueinander aus dem Originalbereich auch im Zielbereich erhalten bleibt.

Programm 4.127 – `removecopy.cpp`:
Demoprogramm zu remove_copy

```
.....
int main(void) {
   int  z[] = { 1, 2, 2, 0, 0, 1, 2, 1 }, z2[8], *p;
   copy(z, z+8, ostream_iterator<int>(cout, " ")); cout << endl;
   for (int i = 0; i < 3; i++) {
      p = remove_copy(z, z+8, z2, i);
      copy(z2, p, ostream_iterator<int>(cout, " "));
      cout << "... " << i << " nicht kopiert" << endl;
   }
}
```

Programm 4.127 liefert die folgende Ausgabe:

```
1 2 2 0 0 1 2 1
1 2 2 1 2 1 ... 0 nicht kopiert
2 2 0 0 2 ... 1 nicht kopiert
1 0 0 1 1 ... 2 nicht kopiert
```

4.6.42 remove_if – Entfernen von Elementen mit Bedingung

```
#include <algorithm>
```

```
template<typename ForwardIterator, typename Predicate>
ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                          Predicate pred);
```

`remove_if` „entfernt“ in dem Bereich `[first, last)` alle Elemente, die die über `pred` vorgegebene Bedingung erfüllen. Die Elemente werden dabei nicht wirklich entfernt, sondern durch nach vorne gezogene Elemente, welche die über `pred` vorgegebene Bedingung nicht erfüllen, überschrieben. Die Größe des Bereichs wird durch `remove_if` nicht verändert.
Als Rückgabewert liefert `remove_if` einen Iterator auf `last - n`, wenn `n` Elemente entfernt wurden.
Um die Elemente wirklich aus dem entsprechenden Bereich zu entfernen, ist Folgendes von Wichtigkeit:

- Um bei `vector`, `deque` und `string` die Elemente wirklich zu entfernen, kann man die Methode `erase` verwenden.
- `list` stellt eine eigene Methode `remove_if` zur Verfügung, die anders als der Algorithmus `remove_if` die entsprechenden Elemente wirklich aus der Liste entfernt und nicht nur überschreibt.
- Die assoziativen Containern bieten zum Löschen von Elementen die Methode `erase` an.

Programm 4.128 – `removeif.cpp`:
Demoprogramm zu remove_if

```
.....
int main(void) {
   int  z[] = { 9, 2, 0, 5, 1, 3, 6, 1, 2, 0 };
   vector<int> v(z, z+10);
   vector<int>::iterator it;
   copy(v.begin(), v.end(), ostream_iterator<int>(cout, " ")); cout << endl;
   for (int i = 3; i >= 0; i--) {
      it = remove_if(v.begin(), v.end(), bind2nd(greater_equal<int>(), i));
      copy(v.begin(), it, ostream_iterator<int>(cout, " ")); cout << " |  ";
      copy(it, v.end(), ostream_iterator<int>(cout, " "));
      cout << "... >=" << i << " geloescht" << endl;
      v.erase(it, v.end()); // Elemente wirlich entfernen
   }
}
```

Programm 4.128 liefert die folgende Ausgabe:

```
9 2 0 5 1 3 6 1 2 0
2 0 1 1 2 0  |  6 1 2 0 ... >=3 geloescht
0 1 1 0  |  2 0 ... >=2 geloescht
0 0  |  1 0 ... >=1 geloescht
 |  0 0 ... >=0 geloescht
```

4.6.43 remove_copy_if – Kopieren aller Elemente, die eine Bedingung nicht erfüllen

```
                                                    #include <algorithm>
```

```
template<typename InputIterator, typename OutputIterator, typename Predicate>
OutputIterator remove_copy_if(InputIterator first, InputIterator last,
                              OutputIterator result, Predicate pred) {
   for ( ; first != last; ++first)
      if (!pred(*first)) {
         *result = *first;
         ++result;
      }
   return result;
}
```

`remove_copy_if` kopiert alle Elemente aus dem Bereich `[first, last)`, die die über `pred` vorgegebene Bedingung nicht erfüllen, in den bei `result` beginnenden Bereich.
`result` sollte nicht in dem Bereich `[first, last)` liegen, da sonst eventuell Elemente schon vor dem Kopieren überschrieben werden.
Der Rückgabewert ist ein Iterator hinter das letzte Element des Bereichs, in den kopiert wurde.
Dieser Algorithmus ist stabil, was bedeutet, dass die relative Reihenfolge der Elemente zueinander aus dem Originalbereich auch im Zielbereich erhalten bleibt.

Programm 4.129 – `removecopyif.cpp`:
Demoprogramm zu remove_copy_if

```
......
int main(void) {
   int  z[] = { 9, 2, 0, 5, 1, 3, 6, 1, 2, 0 },
        z2[10], *p;
   copy(z, z+10, ostream_iterator<int>(cout, " ")); cout << endl;

   for (int i = 0; i < 4; i++) {
      p = remove_copy_if(z, z+10, z2, bind2nd(less_equal<int>(), i));
      copy(z2, p, ostream_iterator<int>(cout, " "));
      cout << "... <=" << i << " nicht kopiert" << endl;
   }
}
```

Programm 4.129 liefert die folgende Ausgabe:

```
9 2 0 5 1 3 6 1 2 0
9 2 5 1 3 6 1 2 ... <=0 nicht kopiert
9 2 5 3 6 2 ... <=1 nicht kopiert
9 5 3 6 ... <=2 nicht kopiert
9 5 6 ... <=3 nicht kopiert
```

4.6.44 replace – Ersetzen von Elementen mit bestimmten Wert

```
#include <algorithm>
```

```
template<typename ForwardIterator, typename T>
void replace(ForwardIterator first, ForwardIterator last,
             const T& old_value, const T& new_value) {
   for ( ; first != last; ++first)
      if (*first == old_value)
         *first = new_value;
}
```

`replace` ersetzt in dem Bereich `[first, last)` bei allen Elementen, die einen Wert gleich `old_value` haben, diesen Wert durch `new_value`.
Für den Typ `T` muss dabei sowohl der Zuweisungsoperator als auch der Vergleichsoperator `==` definiert sein.

Programm 4.130 – `replace.cpp`: Demoprogramm zu replace

```
#include <string>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;

int main(void) {
   string satz("Das ist das Haus vom Nikolaus");
   string vokale("aeiou");
   copy(satz.begin(), satz.end(), ostream_iterator<char>(cout, "")); cout << endl;

   for (unsigned i=0; i < vokale.size(); i++) {
      replace(satz.begin(), satz.end(), vokale[i], '-');
      copy(satz.begin(), satz.end(), ostream_iterator<char>(cout, ""));
      cout << endl;
   }
}
```

Programm 4.130 liefert die folgende Ausgabe:

```
Das ist das Haus vom Nikolaus
D-s ist d-s H-us vom Nikol-us
D-s ist d-s H-us vom Nikol-us
D-s -st d-s H-us vom N-kol-us
D-s -st d-s H-us v-m N-k-l-us
D-s -st d-s H--s v-m N-k-l--s
```

4.6.45 replace_copy – Kopieren mit Ersetzen eines Werts

```
#include <algorithm>
```

```
template<typename InputIter, typename OutputIter, typename T>
OutputIterator replace_copy(InputIterator first, InputIterator last,
                            OutputIterator result,
                            const T& old_value, const T& new_value) {
   for ( ; first != last; ++first, ++result)
      *result = *first == old_value ? new_value : *first;
   return result;
}
```

`replace_copy` kopiert alle Elemente aus dem Bereich `[first, last)`, in den bei `result` beginnenden Bereich, wobei alle Elemente, die den Wert `old_value` besitzen, als neuen Wert `new_value` erhalten. `result` sollte nicht in dem Bereich `[first, last)` liegen, da sonst eventuell Elemente schon vor dem Kopieren überschrieben werden. Der Rückgabewert ist ein Iterator hinter das letzte Element des Bereichs, in den kopiert wurde.
Für den Typ `T` muss dabei sowohl der Zuweisungsoperator als auch der Vergleichsoperator `==` definiert sein.

Programm 4.131 – `replacecopy.cpp`:
Demoprogramm zu replace_copy

```
#include <string>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;
int main(void) {
   string satz("Das ist das Haus vom Nikolaus");
   string vokale("aeiou");
   string neu_satz;
   string::iterator it;
   copy(satz.begin(), satz.end(), ostream_iterator<char>(cout, "")); cout << endl;
   for (unsigned i=0; i < vokale.size(); i++) {
      it = replace_copy(satz.begin(), satz.end(), neu_satz.begin(), vokale[i], '-');
      copy(neu_satz.begin(), it, ostream_iterator<char>(cout, ""));
      cout << endl;
   }
}
```

Programm 4.131 liefert die folgende Ausgabe:

```
Das ist das Haus vom Nikolaus
D-s ist d-s H-us vom Nikol-us
Das ist das Haus vom Nikolaus
Das -st das Haus vom N-kolaus
Das ist das Haus v-m Nik-laus
Das ist das Ha-s vom Nikola-s
```

4.6.46 replace_if – Ersetzen von Elementen mittels Bedingung

```
                                                    #include <algorithm>
```

```
template<typename ForwardIterator, typename Predicate, typename T>
void replace_if(ForwardIterator first, ForwardIterator last,
                Predicate pred, const T& new_value) {
   for ( ; first != last; ++first)
      if (pred(*first))
         *first = new_value;
}
```

`replace_if` ersetzt in dem Bereich `[first, last)` alle Elemente, die die über `pred` vorgegebene Bedingung erfüllen.
Für den Typ `T` muss der Zuweisungsoperator definiert sein.

Programm 4.132 – `replaceif.cpp`: Demoprogramm zu replace_if

```
#include <string>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;

int main(void) {
   string satz("Das ist das Haus vom Nikolaus");
   string vokale("aeiou");
   copy(satz.begin(), satz.end(), ostream_iterator<char>(cout, "")); cout << endl;

   for (unsigned i=0; i < vokale.size(); i++) {
      replace_if(satz.begin(), satz.end(),
                 bind2nd(less_equal<char>(), vokale[i]), '-');
      copy(satz.begin(), satz.end(), ostream_iterator<char>(cout, ""));
      cout << endl;
   }
}
```

Programm 4.132 liefert die folgende Ausgabe:

```
Das ist das Haus vom Nikolaus
--s-ist-d-s---us-vom--ikol-us
--s-ist---s---us-vom--ikol-us
--s--st---s---us-vom---kol-us
--s--st---s---us-v---------us
-----------------v-----------
```

4.6.47 replace_copy_if – Kopieren mit Ersetzen über eine Bedingung

```
                                                    #include <algorithm>
```

```
template<typename InputIter, typename OutputIter, typename Predicate,
          typename T>
OutputIterator replace_copy_if(InputIterator first, InputIterator last,
                               OutputIterator result,
                               Predicate pred, const T& new_value) {
   for ( ; first != last; ++first, ++result)
      *result = pred(*first) ? new_value : *first;
   return result;
}
```

`replace_copy_if` kopiert alle Elemente aus dem Bereich `[first, last)` in den bei `result` beginnenden Bereich, wobei alle Elemente, die die über `pred` vorgegebene Bedingung erfüllen, als neuen Wert `new_value` erhalten.
`result` sollte nicht in dem Bereich `[first, last)` liegen, da sonst eventuell Elemente schon vor dem Kopieren überschrieben werden.
Der Rückgabewert ist ein Iterator hinter das letzte Element des Bereichs, in den kopiert wurde.
Für den Typ `T` muss dabei der Zuweisungsoperator definiert sein.

Programm 4.133 – `replacecopyif.cpp`:
Demoprogramm zu replace_copy_if

```
.....
int main(void) {
   string satz("Das ist das Haus vom Nikolaus");
   string vokale("aeiou");
   string neu_satz;
   string::iterator it;
   copy(satz.begin(), satz.end(), ostream_iterator<char>(cout, "")); cout << endl;

   for (unsigned i=0; i < vokale.size(); i++) {
      it = replace_copy_if(satz.begin(), satz.end(), neu_satz.begin(),
                           bind2nd(greater_equal<char>(), vokale[i]), '-');
      copy(neu_satz.begin(), it, ostream_iterator<char>(cout, ""));
      cout << "  ....>= " << vokale[i] << " ersetzt" << endl;
   }
}
```

Programm 4.133 liefert die folgende Ausgabe:

```
Das ist das Haus vom Nikolaus
D-- --- --- H--- --- N-------  ....>= a ersetzt
Da- --- da- Ha-- --- N----a--  ....>= e ersetzt
Da- --- da- Ha-- --- N----a--  ....>= i ersetzt
Da- i-- da- Ha-- --m Nik-la--  ....>= o ersetzt
Das ist das Ha-s -om Nikola-s  ....>= u ersetzt
```

4.6.48 reverse – Umkehren der Reihenfolge in einem Bereich

```
#include <algorithm>
```

```
template<typename BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);
```

`reverse` kehrt die Reihenfolge der Elemente aus dem Bereich `[first, last)` um.

Programm 4.134 – `reverse.cpp`:
Demoprogramm zu reverse

```
#include <string>
#include <algorithm>
#include <iostream>
using namespace std;
int main(void) {
   string zeile;
   while (getline(cin, zeile)) {
      reverse(zeile.begin(), zeile.end());
      cout << zeile << endl;
   }
}
```

Ruft man Programm 4.134 z. B. wie folgt auf:
./reverse < reverse.cpp
so gibt es die Zeilen der Datei `reverse.cpp` rückwärts aus.

4.6.49 reverse_copy – Kopieren mit Umkehren der Reihenfolge

```
#include <algorithm>
```

```
template<typename BidirectionalIterator, typename OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first, BidirectionalIterator last,
                            OutputIterator result) {
   while (first != last) {
      --last;
      *result = *last;
      ++result;
   }
   return result;
}
```

`reverse_copy` kopiert die Elemente aus dem Bereich `[first, last)` in umgekehrter Reihenfolge in den bei `result` beginnenden Bereich. Der Rückgabewert ist ein Iterator hinter das letzte Element des Bereichs, in den kopiert wurde.
`result` sollte nicht in dem Bereich `[first, last)` liegen, da sonst eventuell Elemente schon vor dem Kopieren überschrieben werden.

Programm 4.135 – `reversecopy.cpp`:
Demoprogramm zu reverse_copy

```
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;
int main(void) {
   int  z[] = { 1, 2, 3, 2, 4, 5, 3 },
        z2[7], *p;
   copy(z, z+7, ostream_iterator<int>(cout, " ")); cout << endl;
   p = reverse_copy(z, z+7, z2);
   copy(z2, p, ostream_iterator<int>(cout, " ")); cout << endl;
}
```

Programm 4.135 liefert die folgende Ausgabe:

```
1 2 3 2 4 5 3
3 5 4 2 3 2 1
```

4.6.50 rotate – Vertauschen zweier Teilbereiche

```
                                                    #include <algorithm>
```

```
template<typename ForwardIterator>
void rotate(ForwardIterator first, ForwardIterator middle, ForwardIterator last);
```

`rotate` verschiebt die Elemente des Bereichs `[middle, last)` vor die Elemente des Bereichs `[first, middle)`. Dies entspricht einer Rotation gegen den Uhrzeigersinn.

Programm 4.136 – `rotate.cpp`:
Demoprogramm zu rotate

```
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;

int main(void) {
   int  z[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

   copy(z, z+9, ostream_iterator<int>(cout, " ")); cout << endl;
   rotate(z, z+3, z+7);
   copy(z, z+9, ostream_iterator<int>(cout, " ")); cout << endl;
}
```

Programm 4.136 liefert die folgende Ausgabe:

```
1 2 3 4 5 6 7 8 9
4 5 6 7 1 2 3 8 9
```

4.6.51 rotate_copy – Kopieren zweier Teilbereiche mit Vertauschen

```
                                                    #include <algorithm>
```

```
template<typename ForwardIter, typename OutputIter>
OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle,
                           ForwardIterator last, OutputIterator result) {
   return copy(first, middle, copy(middle, last, result));
}
```

`rotate_copy` kopiert zuerst die Elemente des Bereichs `[middle, last)` und dann die Elemente des Bereichs `[first, middle)` in den bei `result` beginnenden Bereich. Dies entspricht einer Rotation gegen den Uhrzeigersinn.
Der Rückgabewert ist ein Iterator hinter das letzte Element des Bereichs, in den kopiert wurde.

Programm 4.137 – `rotatecopy.cpp`:
Demoprogramm zu rotate_copy

```
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;

int main(void)
{
   int  z[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 },
        z2[9], *p;

   copy(z, z+9, ostream_iterator<int>(cout, " ")); cout << endl;
   p = rotate_copy(z, z+3, z+7, z2);
   copy(z2, p, ostream_iterator<int>(cout, " ")); cout << endl;
}
```

Programm 4.137 liefert die folgende Ausgabe:

```
1 2 3 4 5 6 7 8 9
4 5 6 7 1 2 3
```

4.6.52 search – Suchen des ersten Vorkommens eines Bereichs

```
                                                    #include <algorithm>
```

```
//... 1. Variante: verwendet operator==()
template<typename ForwardIterator1, typename ForwardIterator2>
ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
                        ForwardIterator2 first2, ForwardIterator2 last2);
```

```
//... 2. Variante: verwendet pred zum Vergleichen
template<typename ForwardIterator1, typename ForwardIterator2, typename BinaryPred>
ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
                        ForwardIterator2 first2, ForwardIterator2 last2,
                        BinaryPred pred);
```

`search` sucht im Bereich `[first1, last1)` nach dem ersten Vorkommen des Bereichs `[first2, last2)`. Wird ein solcher gefunden, wird ein Iterator auf das erste Element dieses Teilbereichs zurückgegeben. Wird kein solcher Teilbereich gefunden, wird `last1` zurückgegeben.
Bei der zweiten Variante wird statt `operator==` das Funktionsobjekt `pred` zum Vergleichen verwendet.

Programm 4.138 – `search.cpp`:
Demoprogramm zu search

```
#include <algorithm>
#include <iterator>
#include <iomanip>
#include <iostream>
using namespace std;

class Abs
{
  public:
     bool operator() (int eins, int zwei) { return abs(eins) == abs(zwei); }
};

int main(void)
{
   int ber1[] = { 2, -3, 7, 3, 7, -9, 7 },
       ber2[] = { 3, 7 };
   int *p;

   cout << showpos;
   copy(ber1, ber1+7, ostream_iterator<int>(cout, " ")); cout << endl;

   p = search(ber1, ber1+7, ber2, ber2+2);
   cout << setw((p-ber1)*3) << "";
   copy(p, p+2, ostream_iterator<int>(cout, " ")); cout << endl;

   p = search(ber1, ber1+7, ber2, ber2+2, Abs());
   cout << setw((p-ber1)*3) << "";
   copy(p, p+2, ostream_iterator<int>(cout, " ")); cout << endl;
}
```

Programm 4.138 liefert die folgende Ausgabe:

```
+2 -3 +7 +3 +7 -9 +7
         +3 +7
   -3 +7
```

4.6.53 search_n – Suchen n gleicher Elemente

```
#include <algorithm>
```

```
//... 1. Variante: verwendet operator==()
template<typename ForwardIterator, typename Size, typename T>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                         Size count, const T& val);

//... 2. Variante: verwendet pred zum Vergleichen
template<typename ForwardIter, typename Size, typename T, typename BinaryPred>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                         Size count, const T& val, BinaryPred pred);
```

`search_n` sucht im Bereich `[first, last)` nach einem Teilbereich, der aus `count` gleichen Werten besteht, die gleich `val` sind. Wird ein solcher gefunden, wird ein Iterator auf das erste Element dieses Teilbereichs zurückgegeben. Wird kein solcher Teilbereich gefunden, wird `last` zurückgegeben.
Bei der zweiten Variante wird statt `operator==` das Funktionsobjekt `pred` zum Vergleichen verwendet.

Programm 4.139 – `searchn.cpp`: Demoprogramm zu search_n

```
#include <algorithm>
#include <iterator>
#include <iomanip>
#include <iostream>
using namespace std;

class Abs {
  public:
     bool operator() (int eins, int zwei) { return abs(eins) == abs(zwei); }
};

int main(void) {
   int z[] = { 2, -4, 4, -4, 7, 4, 4, 4, 3, 7 };
   int *p;

   cout << showpos;
   copy(z, z+10, ostream_iterator<int>(cout, " ")); cout << endl;

   p = search_n(z, z+10, 3, 4); // nach 4 dreimal hintereinander suchen
   cout << setw((p-z)*3) << "";
   copy(p, p+3, ostream_iterator<int>(cout, " ")); cout << endl;

   p = search_n(z, z+10, 3, 4, Abs());
   cout << setw((p-z)*3) << "";
   copy(p, p+3, ostream_iterator<int>(cout, " ")); cout << endl;
}
```

Programm 4.139 liefert die folgende Ausgabe:

```
+2 -4 +4 -4 +7 +4 +4 +4 +3 +7
               +4 +4 +4
   -4 +4 -4
```

4.6.54 set_difference – Differenzmenge aus zwei Bereichen

```
                                                    #include <algorithm>
```

```
//... 1. Variante: verwendet operator<()
template<typename InputIterator1, typename InputIterator2, typename OutputIterator>
OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1,
                              InputIterator2 first2, InputIterator2 last2,
                              OutputIter result) {
   while (first1 != last1 && first2 != last2)
      if (*first1 < *first2) {
         *result = *first1;
         ++first1;
         ++result;
      } else if (*first2 < *first1)
         ++first2;
      else {
         ++first1;
         ++first2;
      }
    return copy(first1, last1, result);
}

//... 2. Variante: verwendet comp zum Vergleichen
template<typename InputIterator1, typename InputIterator2, typename OutputIterator,
         typename Compare>
OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1,
                              InputIterator2 first2, InputIterator2 last2,
                              OutputIterator result, Compare comp) {
   while (first1 != last1 && first2 != last2)
      if (comp(*first1, *first2)) {
         *result = *first1;
         ++first1;
         ++result;
      } else if (comp(*first2, *first1))
         ++first2;
      else {
         ++first1;
         ++first2;
      }
   return copy(first1, last1, result);
}
```

`set_difference` kopiert die Elemente aus dem Bereich `[first1, last1)`, die nicht im Bereich `[first2, last2)` enthalten sind, in den bei `result` beginnenden Bereich. Der Bereich muss bei der ersten Variante mit dem `operator<` und bei der zweiten Variante mit dem Funktionsobjekt `comp` sortiert sein. `result` sollte weder im Bereich `[first1, last1)` noch im Bereich `[first2, last2)` liegen.
Der Rückgabewert ist ein Iterator hinter das letzte Element des Bereichs, in den kopiert wurde.

Programm 4.140 – `setdiff.cpp`:
Demoprogramm zu set_difference

```
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;

class Abs
{
  public:
     bool operator() (int eins, int zwei) { return abs(eins) != abs(zwei); }
};

int main(void)
{
   int ber1[] = { 2, 3, 4, 7, 7, 9 },
       ber2[] = { 3, 7 },
       ber3[] = { 2, -3, 4, 7, 8, 9 },
       diff[6];
   int *p;

   cout << " ";
   copy(ber1, ber1+6, ostream_iterator<int>(cout, " ")); cout << "- ";
   copy(ber2, ber2+2, ostream_iterator<int>(cout, " ")); cout << " = ";
   p = set_difference(ber1, ber1+6, ber2, ber2+2, diff);
   copy(diff, p, ostream_iterator<int>(cout, " ")); cout << endl;

   copy(ber3, ber3+6, ostream_iterator<int>(cout, " ")); cout << "- ";
   copy(ber2, ber2+2, ostream_iterator<int>(cout, " ")); cout << " = ";
   p = set_difference(ber3, ber3+6, ber2, ber2+2, diff, Abs());
   copy(diff, p, ostream_iterator<int>(cout, " "));
   cout << " (bei Absolutwert)" << endl;
}
```

Programm 4.140 liefert die folgende Ausgabe:

```
 2 3 4 7 7 9 - 3 7  = 2 4 7 9
2 -3 4 7 8 9 - 3 7  = 2 4 8 9  (bei Absolutwert)
```

4.6.55 set_intersection – Schnittmenge aus zwei Bereichen

```
#include <algorithm>
```

```
//... 1. Variante: verwendet operator<()
template<typename InputIter1, typename InputIter2, typename OutputIter>
OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1,
                                InputIterator2 first2, InputIterator2 last2,
                                OutputIterator result)
{
   while (first1 != last1 && first2 != last2)
      if (*first1 < *first2)
          ++first1;
      else if (*first2 < *first1)
          ++first2;
      else {
          *result = *first1;
          ++first1;
          ++first2;
          ++result;
      }
   return result;
}

//... 2. Variante: verwendet comp zum Vergleichen
template<typename InputIterator1, typename InputIterator2, typename OutputIterator,
         typename Compare>
OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1,
                                InputIterator2 first2, InputIterator2 last2,
                                OutputIterator result, Compare comp)
{
   while (first1 != last1 && first2 != last2)
      if (comp(*first1, *first2))
         ++first1;
      else if (comp(*first2, *first1))
         ++first2;
      else {
         *result = *first1;
         ++first1;
         ++first2;
         ++result;
      }
   return result;
}
```

`set_intersection` kopiert die Elemente aus dem Bereich `[first1, last1)`, die auch im Bereich `[first2, last2)` enthalten sind, in den bei `result` beginnenden Bereich.
Der Bereich muss bei der ersten Variante mit dem `operator<` und bei der zweiten Variante mit dem Funktionsobjekt `comp` sortiert sein.

`result` sollte weder innerhalb des Bereichs `[first1, last1)` noch innerhalb des Bereichs `[first2, last2)` liegen.
Der Rückgabewert ist ein Iterator hinter das letzte Element des Bereichs, in den kopiert wurde.

Programm 4.141 – `setinter.cpp`:
Demoprogramm zu set_intersection

```
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;

class Abs
{
  public:
     bool operator() (int eins, int zwei) { return abs(eins) != abs(zwei); }
};

int main(void)
{
   int ber1[] = { 2, 3, 4, 7, 7, 9 },
       ber2[] = { 3, 7, 8, 9 },
       ber3[] = { 2, -3, 4, 7, 8, -9 },
       diff[6];
   int *p;

   cout << " ";
   copy(ber1, ber1+6, ostream_iterator<int>(cout, " ")); cout << " --- ";
   copy(ber2, ber2+4, ostream_iterator<int>(cout, " ")); cout << " = ";
   p = set_intersection(ber1, ber1+6, ber2, ber2+4, diff);
   copy(diff, p, ostream_iterator<int>(cout, " ")); cout << endl;

   copy(ber3, ber3+6, ostream_iterator<int>(cout, " ")); cout << "--- ";
   copy(ber2, ber2+4, ostream_iterator<int>(cout, " ")); cout << " = ";
   p = set_intersection(ber3, ber3+6, ber2, ber2+4, diff, Abs());
   copy(diff, p, ostream_iterator<int>(cout, " "));
   cout << " (bei Absolutwert)" << endl;
}
```

Programm 4.141 liefert die folgende Ausgabe:

```
 2 3 4 7 7 9  --- 3 7 8 9  = 3 7 9
2 -3 4 7 8 -9 --- 3 7 8 9  = -3 7 8 -9  (bei Absolutwert)
```

4.6.56 set_symmetric_difference – Symmetrische Differenz

```
#include <algorithm>
```

```
//... 1. Variante: verwendet operator<()
template<typename InputIterator1, typename InputIterator2, typename OutputIterator>
OutputIterator
  set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                           InputIterator2 first2, InputIterator2 last2,
                           OutputIterator result) {
   while (first1 != last1 && first2 != last2)
      if (*first1 < *first2)
         *result++ = *first1++;
      else if (*first2 < *first1)
         *result++ = *first2++;
      else {
         ++first1;
         ++first2;
      }
   return copy(first2, last2, copy(first1, last1, result));
}

//... 2. Variante: verwendet comp zum Vergleichen
template<typename InputIterator1, typename InputIterator2, typename OutputIterator,
         typename Compare>
OutputIterator
   set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2,
                            OutputIterator result, Compare comp) {
   while (first1 != last1 && first2 != last2)
      if (comp(*first1, *first2))
         *result++ = *first1++;
      else if (comp(*first2, *first1))
         *result++ = *first2++;
      else {
         ++first1;
         ++first2;
      }
   return copy(first2, last2, copy(first1, last1, result));
}
```

`set_symmetric_difference` kopiert die Elemente des ersten Bereichs `[first1, last1)`, die nicht im zweiten Bereich `[first2, last2)` enthalten sind, und die, die im zweiten, aber nicht im ersten Bereich enthalten sind, in den bei `result` beginnenden Bereich.
Der Bereich muss bei der ersten Variante mit dem `operator<` und bei der zweiten Variante mit dem Funktionsobjekt `comp` sortiert sein.

`result` sollte weder innerhalb des Bereichs `[first1, last1)` noch innerhalb des Bereichs `[first2, last2)` liegen.
Der Rückgabewert ist ein Iterator hinter das letzte Element des Bereichs, in den kopiert wurde.

Programm 4.142 – `setsymdiff.cpp`:
Demoprogramm zu set_symmetric_difference

```
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;

int main(void)
{
   int ber1[] = { 2, 4, 5, 6, 8, 9 },
       ber2[] = { 2, 3, 5, 6, 9 },
       symm[6], schnitt[6], verein[6];
   int *p1, *p2, *p3;

   copy(ber1, ber1+6, ostream_iterator<int>(cout, " ")); cout << "--- ";
   copy(ber2, ber2+5, ostream_iterator<int>(cout, " ")); cout << "= ";

   p1 = set_symmetric_difference(ber1, ber1+6, ber2, ber2+5, symm);
   copy(symm, p1, ostream_iterator<int>(cout, " ")); cout << endl;

   //... Nachbilden von set_symmetric_difference mit
   //...  Vereinigung(ber1, ber2) - Schnittmenge(p1, p2)
   copy(ber1, ber1+6, ostream_iterator<int>(cout, " ")); cout << "--- ";
   copy(ber2, ber2+5, ostream_iterator<int>(cout, " ")); cout << "= ";
   p2 = set_intersection(ber1, ber1+6, ber2, ber2+5, schnitt);
   p3 = set_union(ber1, ber1+6, ber2, ber2+5, verein);
   p1 = set_difference(verein, verein+6, schnitt, schnitt+5, symm);
   copy(symm, p1, ostream_iterator<int>(cout, " ")); cout << endl;
}
```

Programm 4.142 liefert die folgende Ausgabe:

```
2 4 5 6 8 9 --- 2 3 5 6 9 = 3 4 8
2 4 5 6 8 9 --- 2 3 5 6 9 = 3 4 8
```

4.6.57 set_union – Vereinigungsmenge aus zwei Bereichen

```
#include <algorithm>
```

```
//... 1. Variante: verwendet operator<()
template<typename InputIterator1, typename InputIterator2, typename OutputIterator>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result) {
   while (first1 != last1 && first2 != last2)
      if (*first1 < *first2)
         *result++ = *first1++;
      else if (*first2 < *first1)
         *result++ = *first2++;
      else {
         *result++ = *first1++;
         ++first2;
      }
   return copy(first2, last2, copy(first1, last1, result));
}

//... 2. Variante: verwendet comp zum Vergleichen
template<typename InputIterator1, typename InputIterator2, typename OutputIterator,
         typename Compare>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result, Compare comp) {
   while (first1 != last1 && first2 != last2)
      if (comp(*first1, *first2))
         *result++ = *first1++;
      else if (comp(*first2, *first1))
         *result++ = *first2++;
      else {
         *result++ = *first1++;
         ++first2;
      }
   return copy(first2, last2, copy(first1, last1, result));
}
```

`set_union` kopiert alle Elemente aus den beiden Bereichen `[first1, last1)` und `[first2, last2)`, die in mindestens einem dieser beiden Bereiche enthalten sind, in den bei `result` beginnenden Bereich. Ist ein Element in beiden Bereichen enthalten, wird das Element des ersten Bereichs kopiert.
Der Bereich muss bei der ersten Variante mit dem `operator<` und bei der zweiten Variante mit dem Funktionsobjekt `comp` sortiert sein.
`result` sollte weder innerhalb des Bereichs `[first1, last1)` noch innerhalb des Bereichs `[first2, last2)` liegen.
Der Rückgabewert ist ein Iterator hinter das letzte Element des Bereichs, in den kopiert wurde.

Programm 4.143 – `setunion.cpp`:
Demoprogramm zu set_union

```
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;

class Abs {
  public:
     bool operator() (int eins, int zwei) { return abs(eins) != abs(zwei); }
};

int main(void) {
   int ber1[] = { 2, 3, 4, 7, 7, 9 },
       ber2[] = { 3, 7, 8, 9 },
       ber3[] = { 2, -3, 4, 7, 8, -9 },
       diff[6];
   int *p;

   cout << " ";
   copy(ber1, ber1+6, ostream_iterator<int>(cout, " ")); cout << " --- ";
   copy(ber2, ber2+4, ostream_iterator<int>(cout, " ")); cout << " = ";
   p = set_union(ber1, ber1+6, ber2, ber2+4, diff);
   copy(diff, p, ostream_iterator<int>(cout, " ")); cout << endl;

   copy(ber3, ber3+6, ostream_iterator<int>(cout, " ")); cout << "--- ";
   copy(ber2, ber2+4, ostream_iterator<int>(cout, " ")); cout << " = ";
   p = set_union(ber3, ber3+6, ber2, ber2+4, diff, Abs());
   copy(diff, p, ostream_iterator<int>(cout, " "));
   cout << " (bei Absolutwert)" << endl;
}
```

Programm 4.143 liefert die folgende Ausgabe:

```
 2 3 4 7 7 9  --- 3 7 8 9  = 2 3 4 7 7 8 9
2 -3 4 7 8 -9 --- 3 7 8 9  = 2 -3 4 7 8 -9  (bei Absolutwert)
```

4.6.58 sort – Sortieren eines Bereichs

```
                                                    #include <algorithm>
```

```
//... 1. Variante: verwendet operator<()
template<typename RandomAccessIterator> inline
void sort(RandomAccessIterator first, RandomAccessIterator last);

//... 2. Variante: verwendet comp zum Vergleichen
template<typename RandomAccessIterator, typename Compare> inline
void sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

`sort` sortiert unter Verwendung von `operator<` bzw. des Funktionsobjekts `comp` die Elemente im Bereich `[first, last)`.
Da `sort` den *Quicksort* verwendet, kann in sehr seltenen Fällen der schlechteste Fall auftreten, dass ein Laufzeitverhalten von $O(n^2)$ auftritt. In solchen Fällen sollte man dann `stable_sort` bzw. `partial_sort` verwenden.
Im Normalfall ist jedoch `sort` schneller als diese beiden Sortieralgorithmen.

Programm 4.144 – `sort.cpp`:
Demoprogramm zu sort

```
#include <string>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;

int main(void)
{
   string s[] = { "Robert", "Zorro", "Berta", "Adam", "Micha" };

   sort(s, s+5);
   copy(s, s+5, ostream_iterator<string>(cout, " ")); cout << endl;
   sort(s, s+5, greater<string>());
   copy(s, s+5, ostream_iterator<string>(cout, " ")); cout << endl;
}
```

Programm 4.144 liefert die folgende Ausgabe:

```
Adam Berta Micha Robert Zorro
Zorro Robert Micha Berta Adam
```

4.6.59 stable_partition – Stabiles Gruppieren mit einer Bedingung

```
                                                    #include <algorithm>
```

```
template<typename ForwardIterator, typename Predicate>
ForwardIterator stable_partition(ForwardIterator first, ForwardIterator last,
                                 Predicate pred);
```

`stable_partition` verteilt die Elemente des Bereichs `[first, last)` auf zwei Bereiche, so dass die erste Gruppe nur Elemente enthält, die die über `pred` vorgegebene Bedingung erfüllen, und die zweite Gruppe nur solche Elemente, die diese Bedingung nicht erfüllen.
Im Gegensatz zu `partition` bleibt hier die relative Reihenfolge der Elemente erhalten.
Der Rückgabewert ist ein Iterator auf den Anfang der zweiten Gruppe.

Programm 4.145 – `stablepartition.cpp`:
Demoprogramm zu stable_partition

```
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;

class Kleiner {
  public:
     Kleiner(int z) : zahl(z) { }
     bool operator() (int wert) { return wert < zahl; }
  private:
     int zahl;
};

int main(void) {
   int *p, z[] = { 3, 1, 6, 8, 5, 6, 4, 2, 7, 9 };
   int    z2[] = { 3, 1, 6, 8, 5, 6, 4, 2, 7, 9 };

   copy(z, z+10, ostream_iterator<int>(cout, " ")); cout << endl;

   p = partition(z, z+10, bind2nd(modulus<int>(), 2));
   copy(z, p, ostream_iterator<int>(cout, " ")); cout << "| ";
   copy(p, z+10, ostream_iterator<int>(cout, " ")); cout << " (partition)" <<endl;

   p = stable_partition(z2, z2+10, bind2nd(modulus<int>(), 2));
   copy(z2, p, ostream_iterator<int>(cout, " ")); cout << "| ";
   copy(p, z2+10, ostream_iterator<int>(cout, " "));
   cout << " (stable_partition)" << endl;

   p = partition(z, z+10, Kleiner(4));
   copy(z, p, ostream_iterator<int>(cout, " ")); cout << "| ";
   copy(p, z+10, ostream_iterator<int>(cout, " ")); cout << " (partition)" <<endl;

   p = stable_partition(z2, z2+10, Kleiner(4));
   copy(z2, p, ostream_iterator<int>(cout, " ")); cout << "| ";
   copy(p, z2+10, ostream_iterator<int>(cout, " "));
   cout << " (stable_partition)" << endl;
}
```

Programm 4.145 liefert die folgende Ausgabe:

```
3 1 6 8 5 6 4 2 7 9
3 1 9 7 5 | 6 4 2 8 6  (partition)
3 1 5 7 9 | 6 8 6 4 2  (stable_partition)
3 1 2 | 7 5 6 4 9 8 6  (partition)
3 1 2 | 5 7 9 6 8 6 4  (stable_partition)
```

4.6.60 stable_sort – Stabiles Sortieren eines Bereichs

```
#include <algorithm>
```

```
//... 1. Variante: verwendet operator<()
template<typename RandomAccessIterator> inline
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);
//... 2. Variante: verwendet comp zum Vergleichen
template<typename RandomAccessIterator, typename Compare> inline
void stable_sort(RandomAccessIterator first,RandomAccessIterator last,Compare comp);
```

`sort` sortiert unter Verwendung von `operator<` bzw. des Funktionsobjekts `comp` die Elemente im Bereich `[first, last)`. Im Gegensatz zu `sort` bleibt hier die relative Reihenfolge gleicher Elemente erhalten.
`stable_sort` verwendet den *Heapsort*-Agorithmus.

Programm 4.146 – `stablesort.cpp`:
Demoprogramm zu stable_sort

```
.....
class Elem {
     friend ostream& operator<< (ostream& os, const Elem& e) {
        return os << e.zeich << e.stelle << " ";
     }
  public:
     Elem() {}
     Elem(char z) : zeich(z), stelle(nr++) { }
     bool operator< (const Elem& e) const { return zeich < e.zeich; }
     static int nr;
  private:
     char zeich;
     int  stelle;
};
int Elem::nr = 0;
int main(void) {
  unsigned i;
  Elem     a[20];
   srand(1);
   for (i=0; i<20; i++)
      a[i] = 'A' + rand()%2;
   copy(a, a+20, ostream_iterator<Elem>(cout, "")); cout << endl;
   sort(a, a+20);
   copy(a, a+20, ostream_iterator<Elem>(cout, "")); cout << endl;
   Elem::nr = 0;
   srand(1);
   for (i=0; i<20; i++)
      a[i] = 'A' + rand()%2;
//   copy(a, a+20, ostream_iterator<Elem>(cout, "")); cout << endl;
   stable_sort(a, a+20);
   copy(a, a+20, ostream_iterator<Elem>(cout, "")); cout << endl;
}
```

Programm 4.146 liefert die folgende Ausgabe:

```
B0 A1 B2 B3 B4 B5 A6 A7 B8 B9 A10 B11 A12 B13 B14 A15 A16 A17 A18 A19
A19 A18 A17 A16 A15 A12 A10 A7 A6 A1 B8 B9 B11 B5 B13 B14 B4 B3 B2 B0
A1 A6 A7 A10 A12 A15 A16 A17 A18 A19 B0 B2 B3 B4 B5 B8 B9 B11 B13 B14
```

Programm 4.147 zeigt eine Technik, wie man unter Einsatz des `stable_sort` auf einen Vektor von `string`-Paaren zwei Sortierkriterien nacheinander anwenden kann.

Programm 4.147 – `stablesort2.cpp`:
Sortieren nach zwei Sortierkriterien mit stable_sort

```
#include <string>
#include <vector>
#include <algorithm>
#include <functional>
#include <iterator>
#include <iostream>
using namespace std;
typedef pair<string, string> paar;
class MySort {
  public:
     MySort(string paar::*komp) : komponente(komp) { }
     bool operator() (const paar& eins, const paar& zwei) const {
        return eins.*komponente < zwei.*komponente;
     }
  private:
     string paar::*komponente;
};
ostream& operator<< (ostream& os, const vector<paar>& v) {
   for (unsigned i=0; i < v.size(); i++)
      os << "   " << v[i].first << ", " << v[i].second << endl;
   return os;
}
int main(void) {
   vector<paar> namen;
   namen.push_back(paar("Meier",   "Hans"));
   namen.push_back(paar("Schmitt", "Toni"));
   namen.push_back(paar("Schmitt", "Hans"));
   namen.push_back(paar("Meier",   "Fritz"));
   namen.push_back(paar("Schmitt", "Egon"));
   namen.push_back(paar("Meier",   "Berta"));
   namen.push_back(paar("Schmitt", "Anna"));
   cout << ".... Nach Vornamen sortiert" << endl;
   sort(namen.begin(), namen.end(), MySort(&paar::second));
   cout << namen;
   cout << ".... und nun zusaetzl. noch nach Nachnamen sortiert" << endl;
   stable_sort(namen.begin(), namen.end(), MySort(&paar::first));
   cout << namen;
}
```

Programm 4.147 liefert die folgende Ausgabe:

```
.... Nach Vornamen sortiert
   Schmitt, Anna
   Meier, Berta
   Schmitt, Egon
   Meier, Fritz
   Meier, Hans
   Schmitt, Hans
   Schmitt, Toni
.... und nun zusaetzl. noch nach Nachnamen sortiert
   Meier, Berta
   Meier, Fritz
   Meier, Hans
   Schmitt, Anna
   Schmitt, Egon
   Schmitt, Hans
   Schmitt, Toni
```

4.6.61 swap – Vertauschen zweier Objekte

```
                                                    #include <algorithm>
```

```
template<typename T> inline
void swap(T& a, T& b) {
   T tmp = a;
   a = b;
   b = tmp;
}
```

`swap` vertauscht zwei Objekte. Für den Typ `T` muss der Zuweisungsoperator und der Kopierkonstruktor definiert sein. Container-Klassen verfügen über eigene Methoden `swap`, die effizienter als dieser Algorithmus `swap` sind.

Programm 4.148 – `swap.cpp`:
Demoprogramm zu swap

```
.....
int main(void)
{
   string d[] = { "eins", "zwei", "drei",  "vier" },
          e[] = { "one",  "two",  "three", "four" };

   copy(d, d+4, ostream_iterator<string>(cout, ", ")); cout << endl;
   copy(e, e+4, ostream_iterator<string>(cout, ", ")); cout << endl;
   cout << "----------------------" << endl;
   for (unsigned i=0; i < 4; i++)
      swap(d[i], e[i]);
   copy(d, d+4, ostream_iterator<string>(cout, ", ")); cout << endl;
   copy(e, e+4, ostream_iterator<string>(cout, ", ")); cout << endl;
}
```

Programm 4.148 liefert die folgende Ausgabe:

```
eins, zwei, drei, vier,
one, two, three, four,
----------------------
one, two, three, four,
eins, zwei, drei, vier,
```

4.6.62 swap_ranges – Vertauschen der Elemente zweier Bereiche

```
                                                    #include <algorithm>
```

```
template<typename ForwardIterator1, typename ForwardIterator2>
ForwardIterator2 swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1,
                             ForwardIterator2 first2) {
   for ( ; first1 != last1; ++first1, ++first2)
      iter_swap(first1, first2);
   return first2;
}
```

`swap_ranges` vertauscht die Elemente aus dem Bereich `[first1, last1)` mit den Elementen aus dem Bereich, der bei `first2` beginnt. Diese beiden Bereiche sollten sich nicht überlappen
`swap_ranges` liefert Iterator hinter das letzte getauschte Element in dem bei `first2` beginnenden Bereich.

Programm 4.149 – `swapranges.cpp`:
Demoprogramm zu swap_ranges

```
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;

int main(void)
{
   int  z1[] = { 1, 2, 3, 4, 5, 6, 7 },
        z2[] = { 10, 20, 30, 40, 50, 60 };

   swap_ranges(z1+2, z1+6, z2+1);
   copy(z1, z1+7, ostream_iterator<int>(cout, " ")); cout << endl;
   copy(z2, z2+6, ostream_iterator<int>(cout, " ")); cout << endl;
}
```

Programm 4.149 liefert die folgende Ausgabe:

```
1 2 20 30 40 50 7
10 3 4 5 6 60
```

4.6.63 transform – Kopieren mit Modifizieren

```
                                                    #include <algorithm>
```

```
template<typename InputIterator, typename OutputIterator, typename UnaryOperation>
OutputIterator transform(InputIterator first, InputIterator last,
                         OutputIterator result, UnaryOperation unary_op) {
   for ( ; first != last; ++first, ++result)
      *result = unary_op(*first);
   return result;
}
template<typename InputIterator1, typename InputIterator2, typename OutputIterator,
         typename BinaryOperation>
OutputIterator transform(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, OutputIterator result,
                         BinaryOperation binary_op) {
   for ( ; first1 != last1; ++first1, ++first2, ++result)
      *result = binary_op(*first1, *first2);
   return result;
}
```

Die erste Variante von `transform` kopiert die Elemente aus dem Bereich `[first, last)` in den bei `result` beginnenden Bereich, wobei es aber jedes Element mittels des Funktionsobjekts `unary_op` modifiziert. Die zweite Variante von `transform` kopiert die Elemente aus dem Bereich `[first1, last1)` und dem bei `first2` beginnenden Bereich in den bei `result` beginnenden Bereich, wobei es aber für jedes Elementpaar aus den beiden Bereichen das Funktionsobjekt `binary_op` aufruft. `transform` und `for_each` unterscheiden sich in den folgenden Punkten:

- Bei `transform` wird der Rückgabewert des Funktionsobjekts `operator()` verwendet und das an `operator()` übergebene Argument selbst wird nicht modifiziert.
- Bei `for_each` wird das an das Funktionsobjekt `operator()` übergebene Argument als Referenz übergeben, das innerhalb des Funktionsobjekts `operator()` geändert werden kann.

`transform` liefert einen Iterator hinter das letzte kopierte Element in dem bei `result` beginnenden Bereich.

Programm 4.150 – `transform.cpp`: Demoprogramm zu transform

```
.....
class Verschluessel
{
  public:
     string operator() (const string s) {
        string tmp = s;
        transform(tmp.begin(), tmp.end(), tmp.begin(), bind2nd(minus<int>(), 1));
        return tmp;
     }
};
```

```
int main(void) {
  int  z1[] = { 1, 2, 3, 4, 5, 6, 7 };
  vector<int> quadrat;

   copy(z1, z1+7, ostream_iterator<int>(cout, " ")); cout << endl;
   transform(z1, z1+7, z1, back_inserter(quadrat), multiplies<int>());
   copy(quadrat.begin(), quadrat.end(), ostream_iterator<int>(cout, " "));
   cout << endl;
   transform(z1, z1+7, quadrat.begin(), ostream_iterator<int>(cout, " "),
             plus<int>());
   cout << endl;
   string s[] = { "eins", "zwei", "drei", "vier" };
   copy(s, transform(s, s+4, s, Verschluessel()),
        ostream_iterator<string>(cout, " "));  cout << endl;
}
```

Programm 4.150 liefert die folgende Ausgabe:

```
1 2 3 4 5 6 7
1 4 9 16 25 36 49
2 6 12 20 30 42 56
dhmr yvdh cqdh uhdq
```

4.6.64 unique – Komprimieren gleicher Elementfolgen

```
                                                    #include <algorithm>
```

```
//... 1. Variante: verwendet operator==()
template<typename ForwardIterator>
ForwardIterator unique(ForwardIterator first, ForwardIterator last) {
   first = adjacent_find(first, last);
   return unique_copy(first, last, first);
}
//... 2. Variante: verwendet pred zum Vergleichen
template<typename ForwardIterator, typename BinaryPredicate>
ForwardIterator unique(ForwardIterator first, ForwardIterator last,
                       BinaryPredicate pred) {
   first = adjacent_find(first, last, pred);
   return unique_copy(first, last, first, pred);
}
```

`unique` „entfernt“ in dem Bereich `[first, last)` bei Folgen gleicher Elemente alle bis auf das Erste. Die Elemente werden dabei nicht wirklich entfernt, sondern durch nach vorne gezogene Elemente überschrieben. Die Größe des Bereichs wird durch `unique` nicht verändert.
Bei der zweiten Variante wird statt `operator==` das Funktionsobjekt `pred` zum Vergleichen verwendet.
Als Rückgabewert liefert `unique` einen Iterator auf `last - n`, wenn `n` Elemente entfernt wurden.

Programm 4.151 – `unique.cpp`:
Demoprogramm zu unique

```
#include <vector>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;
int main(void) {
   int  z[] = { 3, 2, 2, 1, 2, 1, 1, 0 };
   vector<int> v1(z, z+8),
               v2(z, z+8),
               v3(z, z+8);
   vector<int>::iterator it;

   copy(v1.begin(), v1.end(), ostream_iterator<int>(cout, " ")); cout << endl;
   it = unique(v1.begin(), v1.end());
   copy(v1.begin(), it, ostream_iterator<int>(cout, " ")); cout << " |  ";
   copy(it, v1.end(), ostream_iterator<int>(cout, " ")); cout << endl;
   v1.erase(it, v1.end()); // Elemente wirlich entfernen
   copy(v1.begin(), v1.end(), ostream_iterator<int>(cout, " ")); cout << endl;
   cout << "------------------" << endl;
   sort(v2.begin(), v2.end());
   copy(v2.begin(), v2.end(), ostream_iterator<int>(cout, " ")); cout << endl;
   it = unique(v2.begin(), v2.end());
   copy(v2.begin(), it, ostream_iterator<int>(cout, " ")); cout << " |  ";
   copy(it, v2.end(), ostream_iterator<int>(cout, " ")); cout << endl;
   v2.erase(it, v2.end()); // Elemente wirlich entfernen
   copy(v2.begin(), v2.end(), ostream_iterator<int>(cout, " ")); cout << endl;
   cout << "------------------" << endl;
   copy(v3.begin(), v3.end(), ostream_iterator<int>(cout, " ")); cout << endl;
   it = unique(v3.begin(), v3.end(), greater<int>());
   copy(v3.begin(), it, ostream_iterator<int>(cout, " ")); cout << " |  ";
   copy(it, v3.end(), ostream_iterator<int>(cout, " ")); cout << endl;
   v3.erase(it, v3.end()); // Elemente wirlich entfernen
   copy(v3.begin(), v3.end(), ostream_iterator<int>(cout, " ")); cout << endl;
}
```

Programm 4.151 liefert die folgende Ausgabe:

```
3 2 2 1 2 1 1 0
3 2 1 2 1 0  |  1 0
3 2 1 2 1 0
------------------
0 1 1 1 2 2 2 3
0 1 2 3  |  2 2 2 3
0 1 2 3
------------------
3 2 2 1 2 1 1 0
3  |  2 2 1 2 1 1 0
3
```

4.6.65 unique_copy – Kopieren und Komprimieren gleicher Elementfolgen

```
#include <algorithm>
```

```
//... 1. Variante: verwendet operator==()
template<typename InputIterator, typename OutputIterator>
OutputIterator unique_copy(InputIterator first, InputIterator last,
                           OutputIterator result);
//... 2. Variante: verwendet pred zum Vergleichen
template<typename InputIterator, typename OutputIterator, typename BinaryPredicate>
OutputIterator unique_copy(InputIterator first, InputIterator last,
                           OutputIterator result, BinaryPredicate pred);
```

`unique_copy` kopiert die Elemente aus dem Bereich `[first, last)` in den bei `result` beginnenden Bereich, wobei bei Folgen gleicher Elemente nur das Erste kopiert wird. Bei der zweiten Variante wird statt `operator==` das Funktionsobjekt `pred` zum Vergleichen verwendet. `unique_copy` liefert einen Iterator hinter das letzte kopierte Element in dem bei `result` beginnenden Bereich.

Programm 4.152 – `uniquecopy.cpp`: Demoprogramm zu unique_copy

```
.....
int main(void) {
   int  z[] = { 3, 2, 2, 1, 2, 1, 1, 0 };
   vector<int> v1(8), v2(8), v3(z, z+8);
   vector<int>::iterator it;
   copy(z, z+8, ostream_iterator<int>(cout, " ")); cout << endl;
   it = unique_copy(z, z+8, v1.begin());
   copy(v1.begin(), it, ostream_iterator<int>(cout, " "));
   cout << endl << "-------------------" << endl;
   sort(z, z+8); copy(z, z+8, ostream_iterator<int>(cout, " ")); cout << endl;
   it = unique_copy(z, z+8, v2.begin());
   copy(v2.begin(), it, ostream_iterator<int>(cout, " "));
   cout << endl << "-------------------" << endl;
   copy(v3.begin(), v3.end(), ostream_iterator<int>(cout, " ")); cout << endl;
   it = unique_copy(v3.begin(), v3.end(), v3.begin(), less_equal<int>());
   copy(v3.begin(), it, ostream_iterator<int>(cout, " "));
}
```

Programm 4.152 liefert die folgende Ausgabe:

```
3 2 2 1 2 1 1 0
3 2 1 2 1 0
-------------------
0 1 1 1 2 2 2 3
0 1 2 3
-------------------
3 2 2 1 2 1 1 0
3 2 1 0
```

4.6.66 upper_bound – Letzte Einfügeposition ohne Umsortierung

```
                                                    #include <algorithm>
```

```
//... 1. Variante: verwendet operator<()
template<typename ForwardIterator, typename T>
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
                            const T& val);
//... 2. Variante: verwendet comp zum Vergleichen
template<typename ForwardIterator, typename T, typename Compare>
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
                            const T& val, Compare comp);
```

`upper_bound` liefert die letzte Position aus dem Bereich `[first, last]` (`last` gehört hier dazu), an der `val` eingefügt werden kann, ohne dass dadurch die Sortierung verletzt wird. Wird kein solches Element gefunden, liefert `upper_bound` als Rückgabewert `first`.
Der Bereich muss bei der ersten Variante mit dem `operator<` und bei der zweiten Variante mit dem Funktionsobjekt `comp` sortiert sein.
Da die assoziativen Container Bidirectional-Iteratoren anbieten, sollte für sie nicht der Algorithmus `upper_bound`, sondern die von diesen Containern eigens angebotene gleichnamige Methode verwendet werden.

Programm 4.153 – `upperbound.cpp`:
Demoprogramm zu upper_bound

```
.....
int main(void) {
  int z[] = { 1, 3, 5, 7, 7, 7, 8, 8 }, *p;
   cout << " "; copy(z, z+8, ostream_iterator<int>(cout, " ")); cout << endl;
   p = upper_bound(z, z+8, 7); cout << setw((p-z)*2+1) << "7" << endl;
   p = upper_bound(z, z+8, 4); cout << setw((p-z)*2+1) << "4" << endl;
   p = upper_bound(z, z+8, 9); cout << setw((p-z)*2+1) << "9" << endl;
   reverse(z, z+8); //... Reihenfolge umkehren
   cout << " "; copy(z, z+8, ostream_iterator<int>(cout, " ")); cout << endl;
   p = upper_bound(z, z+8, 7, greater<int>());
   cout << setw((p-z)*2+1) << "7" << endl;
   p = upper_bound(z, z+8, 9, greater<int>());
   cout << setw((p-z)*2+1) << "9" << endl;
}
```

Programm 4.153 liefert die folgende Ausgabe:

```
 1 3 5 7 7 7 8 8
           7
    4
               9
 8 8 7 7 7 5 3 1
         7
9
```

4.6.67 Heap-Algorithmen

```
#include <algorithm>
```

Ein *Heap* ist eine Form eines Binärbaums, der als Array realisiert ist. In einem Standard-Heap ist der Schlüssel eines Element-Knotens immer grösser oder gleich den Schlüsseln seiner Kindknoten. Einen solchen Heap bezeichnet man auch als *Max-Heap*. Abbildung 4.3 zeigt einen Heap, der in Form eines Binärbaums realisiert ist.

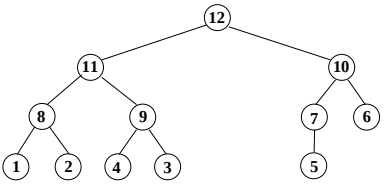


Abbildung 4.3: Realisierung eines Heaps als Binärbaum

In Abbildung 4.3 ist Folgendes erkennbar:

- 12 ist der Wurzel-Knoten, der zwei Kindknoten 11 und 10 besitzt, die beide kleiner als 12 sind.
- 11 hat seinerseits zwei Kindknoten 8 und 9, die beide kleiner als 11 sind.
- 10 hat auch zwei Kindknoten 7 und 6, die beide kleiner als 10 sind.
- 8 hat wieder zwei Kindknoten 1 und 2, die beide kleiner als 8 sind.
- 9 hat auch zwei Kindknoten 4 und 3, die beide kleiner als 9 sind.
- 7 hat nur einen Kindknoten 5, der kleiner als 7 ist.
- 6 hat keinen Kindknoten.

Auch ist in Abbildung 4.3 erkennbar, dass die Elemente auf einer Ebene nicht zwangsweise sortiert vorliegen. So gilt zwar 8 < 9, aber nicht 7 < 6 auf der dritten Ebene.
Heaps können für Container verwendet werden, die Random-Access-Iteratoren unterstützen. Folglich kann man z. B. für ein `list`-Objekt keinen Heap verwenden, da die Klasse `list` keine Random-Access-Iteratoren unterstützt.
Heaps eignen sich insbesondere für `priority_queue`-Objekte, die auch unter zu Hilfenahme der Heap-Algorithmen realisiert sind.

make_heap – Anordnen von Elementen als Heap

```
//... 1. Variante: verwendet operator<()
template<typename RandomAccessIterator>
void make_heap(RandomAccessIterator first, RandomAccessIterator last);
//... 2. Variante: verwendet comp zum Vergleichen
template<typename RandomAccessIterator, typename Compare> inline
void make_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
```

`make_heap` konstruiert durch Umordnen der Elemente aus dem Bereich `[first,last)` einen Heap, wobei die Elemente entweder mit dem `operator<` oder mit Funktionsobjekt `comp` angeordnet werden.

pop_heap – Entfernen eines Elements aus dem Heap

```
//... 1. Variante: verwendet operator<()
template<typename RandomAccessIterator> inline
void pop_heap(RandomAccessIterator first, RandomAccessIterator last);
//... 2. Variante: verwendet comp zum Vergleichen
template<typename RandomAccessIterator, typename Compare> inline
void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
              Compare comp);
```

`pop_heap` entfernt ein Element aus einem Heap. Ist der Bereich `[first, last)` ein gültiger Heap, vertauscht `pop_heap` das Element an der Position `first` mit dem Element an der Position `last` und stellt dann für den Bereich `[first, last-1)` wieder einen gültigen Heap her, der entweder mit dem `operator<` oder mit dem Funktionsobjekt `comp` umgeordnet ist.

push_heap – Hinzufügen eines neuen Elements zum Heap

```
//... 1. Variante: verwendet operator<()
template<typename RandomAccessIterator> inline
void push_heap(RandomAccessIterator first, RandomAccessIterator last);
//... 2. Variante: verwendet comp zum Vergleichen
template<typename RandomAccessIterator, typename Compare> inline
void push_heap(RandomAccessIterator first, RandomAccessIterator last,
i               Compare comp);
```

`push_heap` platziert ein neues Element auf einem Heap. Ist der Bereich `[first, last-1)` ein gültiger Heap, fügt `push_heap` das Element so in dem Heap ein, dass anschließend der Bereich `[first, last)` einen gültigen Heap darstellt, der entweder mit dem `operator<` oder mit dem Funktionsobjekt `comp` umgeordnet ist.

sort_heap – Sortieren der Elemente eines Heaps

```
//... 1. Variante: verwendet operator<()
template<typename RandomAccessIterator>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last) {
   while (last - first > 1)
      pop_heap(first, last--);
}
//... 2. Variante: verwendet comp zum Vergleichen
template<typename RandomAccessIterator, typename Compare>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp) {
   while (last - first > 1)
      pop_heap(first, last--, comp);
}
```

`sort_heap` sortiert die Elemente des Heaps `[first, last)` unter Verwendung des `operator<` oder des Funktionsobjekts `comp`. Infolge dieser Sortierung geht die Heap-Eigenschaft des Bereichs `[first, last)` verloren. Bei `sort_heap` findet kein stabiles

Sortieren statt, was bedeutet, dass gleiche Elemente eventuell in einer anderen Reihenfolge zueinander stehen als vor der Sortierung.

Beispiele zu den Heap-Algorithmen

Programm 4.154 – `heap.cpp`:
Demoprogramm zu den Heap-Algorithmen

```
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;

int main(void) {
   int  z[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };

   cout << "Am Anfang:" << endl << "  ";
   copy(z, z+12, ostream_iterator<int>(cout, " "));

   make_heap(z, z+12);
   cout << endl << "nach make_heap:" << endl << "  ";
   copy(z, z+12, ostream_iterator<int>(cout, " "));

   pop_heap(z, z+12);
   cout << endl << "nach pop_heap:" << endl << "  ";
   copy(z, z+11, ostream_iterator<int>(cout, " ")); cout << " |  " << z[11];

   pop_heap(z, z+11);
   cout << endl << "nach nochmal pop_heap:" << endl << "  ";
   copy(z, z+10, ostream_iterator<int>(cout, " ")); cout << " |  ";
   copy(z+10, z+12, ostream_iterator<int>(cout, " "));

   push_heap(z, z+11);
   cout << endl << "nach push_heap:" << endl << "  ";
   copy(z, z+11, ostream_iterator<int>(cout, " ")); cout << " |  " << z[11];

   push_heap(z, z+12);
   cout << endl << "nach push_heap:" << endl << "  ";
   copy(z, z+12, ostream_iterator<int>(cout, " ")); cout << endl;

   sort_heap(z, z+12); // nun kein Heap mehr
   copy(z, z+12, ostream_iterator<int>(cout, " ")); cout << endl;

   make_heap(z, z+12); // nun wieder Heap
   copy(z, z+12, ostream_iterator<int>(cout, " ")); cout << endl;
   for (int i=0; i < 12; i++) {
      pop_heap(z, z+12-i);
      cout << z[11-i] << " ";
   }
   cout << endl;
}
```

Programm 4.154 liefert die folgende Ausgabe:

```
Am Anfang:
  1 2 3 4 5 6 7 8 9 10 11 12
nach make_heap:
  12 11 7 9 10 6 1 8 4 2 5 3
nach pop_heap:
  11 10 7 9 5 6 1 8 4 2 3  |  12
nach nochmal pop_heap:
  10 9 7 8 5 6 1 3 4 2  |  11 12
nach push_heap:
  11 10 7 8 9 6 1 3 4 2 5  |  12
nach push_heap:
  12 10 11 8 9 7 1 3 4 2 5 6
1 2 3 4 5 6 7 8 9 10 11 12
12 11 7 9 10 6 1 8 4 2 5 3
12 11 10 9 8 7 6 5 4 3 2 1
```

Programm 4.155 – `heap2.cpp`:
Weiteres Demoprogramm zu den Heap-Algorithmen

```
#include <string>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;

int main(void) {
   string s[] = { "Zorro", "Berta", "Anna", "Toni", "Micha" };

   cout << "Am Anfang:" << endl << "  ";
   copy(s, s+5, ostream_iterator<string>(cout, " "));

   make_heap(s, s+5, greater<string>());
   cout << endl << "nach make_heap:" << endl << "  ";
   copy(s, s+5, ostream_iterator<string>(cout, " "));

   pop_heap(s, s+5, greater<string>());
   cout << endl << "nach pop_heap:" << endl << "  ";
   copy(s, s+4, ostream_iterator<string>(cout, " ")); cout << " |  " << s[4];

   pop_heap(s, s+4, greater<string>());
   cout << endl << "nach nochmal pop_heap:" << endl << "  ";
   copy(s, s+3, ostream_iterator<string>(cout, " ")); cout << " |  ";
   copy(s+3, s+5, ostream_iterator<string>(cout, " "));

   push_heap(s, s+4, greater<string>());
   cout << endl << "nach push_heap:" << endl << "  ";
   copy(s, s+4, ostream_iterator<string>(cout, " ")); cout << " |  " << s[4];

   push_heap(s, s+5, greater<string>());
```

```
   cout << endl << "nach push_heap:" << endl << "  ";
   copy(s, s+5, ostream_iterator<string>(cout, " ")); cout << endl;

   sort_heap(s, s+5, greater<string>()); // nun kein Heap mehr
   copy(s, s+5, ostream_iterator<string>(cout, " ")); cout << endl;

   make_heap(s, s+5, greater<string>()); // nun wieder Heap
   copy(s, s+5, ostream_iterator<string>(cout, " ")); cout << endl;
   for (int i=0; i < 5; i++) {
      pop_heap(s, s+5-i, greater<string>());
      cout << s[4-i] << " ";
   }
   cout << endl;
}
```

Programm 4.155 liefert die folgende Ausgabe:

```
Am Anfang:
  Zorro Berta Anna Toni Micha
nach make_heap:
  Anna Berta Zorro Toni Micha
nach pop_heap:
  Berta Micha Zorro Toni  |  Anna
nach nochmal pop_heap:
  Micha Toni Zorro  |  Berta Anna
nach push_heap:
  Berta Micha Zorro Toni  |  Anna
nach push_heap:
  Anna Berta Zorro Toni Micha
Zorro Toni Micha Berta Anna
Anna Berta Micha Zorro Toni
Anna Berta Micha Toni Zorro
```

Kapitel 5

Weitere Komponenten der C++-Standardbibliothek

5.1 Die Klasse auto_ptr

Ein Problem bei Zeigern ist, dass man immer Informationen über den Speicher unterhalten muss, auf den diese Zeiger. Verliert man eine solche Information, weil eine entsprechende Zeigervariable mit der zugehörigen Adresse nicht mehr zugreifbar ist, entstehen Speicherlücken, die nicht mehr benötigten Speicher belegen, den man nicht mehr freigeben kann. Üblicherweise gibt man lokal allozierten und nicht mehr benötigten Speicher wieder vor dem Verlassen des Blocks, in dem der Speicher alloziert wurde, wieder frei, wie z. B.:

```
void speicher(void) {
   char *zgr = new char [10000]; // Dynamisches Allozieren von 10000 Bytes
   //.... Operationen
   delete zgr; // Freigabe des dynamisch allozierten Speichers (alles OK)
}
```

Tritt aber nun vor der Ausführung von `delete` eine Exception auf, wird die Funktion `speicher()` vorzeitig verlassen und die `delete`-Anweisung nicht mehr ausgeführt, woraus dann zwangsläufig eine nicht mehr zugreifbare und damit auch nicht mehr freigebbare „Speicherleiche“ resultiert.
Um dies zu vermeiden, könnte man Exceptions abfangen, um so in jedem Fall den lokal allozierten Speicher freizugeben, wie z. B.:

```
void speicher(void) {
   char *zgr = new char [10000]; // Dynamisches Allozieren von 10000 Bytes
   try {
      //.... Operationen (evtl. mit Auftreten von Exceptions)
   } catch(...) {
      delete zgr; // Freigabe im Falle einer Exception
      throw;
   }
   delete zgr; // Freigabe im Normalfall
}
```

Dieses explizite Abfangen von Exceptions ist aber doch sehr umständlich. Ein besserer Ansatz ist sicher, dass es nicht in der Verantwortung des Programmierers liegt, „Speicherleichen“ zu vermeiden, sondern dass solche automatisch vermieden werden. Hierzu bietet die C++-Standardbibliothek die Klasse `auto_ptr` an, welche den entsprechenden Speicherplatz automatisch wieder freigibt, wenn keine Referenz mehr auf diesen vorliegt.

5.1.1 Definition von auto_ptr-Variablen

Verwendet man die Klasse `auto_ptr`, muss man die folgende Headerdatei inkludieren:

```
#include <memory>
```

Ein `auto_ptr`-Objekt zeigt üblicherweise auf einen dynamisch allozierten Speicher. Das Besondere an `auto_ptr`-Objekten ist, dass bei ihrer Zerstörung automatisch auch der Speicherplatz freigegeben wird, auf den diese zeigen, so dass sich der Programmierer nicht mehr selbst um die Freigabe von dynamisch reservierten Speicher kümmern muss, wenn auf diesen nicht mehr zugreifbar ist.
Somit wird dem Programmierer die Verantwortung zur Vermeidung von „Speicherleichen“ abgenommen.

Definieren von auto_ptr-Variablen mit einer new-Initialisierung

```
auto_ptr<T> aptr(new T(...));
auto_ptr<T> aptr = auto_ptr<int>(new T(...));
```

Programm 5.1 – `autoptr1.cpp`:
Definieren von auto_ptr-Variablen mit einer new-Initialisierung

```
#include <string>
#include <memory>
#include <iostream>
using namespace std;
int main(void) {
   auto_ptr<int> iPtr1 = auto_ptr<int>(new int(123));
   auto_ptr<int> iPtr2 = auto_ptr<int>(new int);
   cout << *iPtr1 << endl;
   *iPtr2 = 321;
   cout << *iPtr2 << endl;
   auto_ptr<string> sPtr1(new string("Ein haus"));
   auto_ptr<string> sPtr2 = auto_ptr<string>(new string);
   cout << *sPtr1 << endl;
   sPtr1->insert(4, "Reihen");
   cout << *sPtr1 << endl;
   *sPtr2 = "Ein mann";
   cout << *sPtr2 << endl;
   sPtr2->insert(4, "Schnee");
   cout << *sPtr2 << endl;
}
```

Programm 5.1 liefert die folgende Ausgabe:

```
123
321
Ein haus
Ein Reihenhaus
Ein mann
Ein Schneemann
```

Definieren von auto_ptr-Variablen und Initialisieren mit auto_ptr-Objekt

```
auto_ptr<T> aptr1;
......
auto_ptr<T> aptr2(aptr1);
auto_ptr<T> aptr2 = aptr1;
```

Hierbei ist jedoch zu beachten, dass beim Kopieren eines `auto_ptr`-Objekts der enthaltene Zeiger kopiert wird, und nun nicht mehr dem Original `aptr1`, sondern der Kopie `aptr2` gehört. Das Original `aptr1` ist damit nicht mehr Eigentümer des Objekts, weshalb dessen Zeiger auf `0` gesetzt wird. Programm 5.2 verdeutlicht dies, indem es sich mit der von der Klasse `auto_ptr` angebotenen Methode `get()` die in den jeweiligen `auto_ptr`-Objekten enthaltenen Zeiger ausgeben lässt.

Programm 5.2 – `autoptr2.cpp`:
Beim Kopieren wird im Original der Zeiger auf 0 gesetzt

```
#include <string>
#include <memory>
#include <iostream>
using namespace std;

int main(void) {
   auto_ptr<int> iPtr1(new int(123));
   auto_ptr<int> iPtr2(iPtr1);

   cout << "iPtr2 = " << *iPtr2 << " (" << iPtr2.get() << ")" << endl;
   cout << "iPtr1 = (" << iPtr1.get() << ")" << endl;

   auto_ptr<string> sPtr1(new string("Hallo"));
   auto_ptr<string> sPtr2 = sPtr1;

   cout << "sPtr2 = " << *sPtr2 << " (" << sPtr2.get() << ")" << endl;
   cout << "sPtr1 = (" << sPtr1.get() << ")" << endl;
}
```

Programm 5.2 liefert z. B. die folgende Ausgabe:

```
iPtr2 = 123 (0x804b008)
iPtr1 = (0)
sPtr2 = Hallo (0x804b018)
sPtr1 = (0)
```

Definieren von auto_ptr-Variablen ohne Initialisierung

```
auto_ptr<T> aptr;
```

Programm 5.3 zeigt, dass man `auto_ptr`-Variablen auch nachträglich dynamisch allozierten Speicher zuweisen kann. Dieses Programm zeigt allerdings auch, dass mit der Zerstörung des `auto_ptr`-Objekts auch der zugehörige Speicher automatisch freigegeben wird, so dass kein Zugriff mehr über den Zeiger möglich ist, der diesen Speicher ursprünglich alloziert hat.

Programm 5.3 – `autoptr3.cpp`:
Nachträgliche Zuweisung eines zuvor dynamisch allozierten Speicher

```
#include <string>
#include <memory>
#include <iostream>
using namespace std;

int main(void)
{
   auto_ptr<int> iPtr;
   int *zgr = new int(123);

   iPtr = auto_ptr<int>(zgr);
   cout << "iPtr = " << *iPtr << " (" << iPtr.get() << ")" << endl;
   cout << "zgr  = " << *zgr  << " (" << zgr        << ")" << endl;
   *zgr = 321;
   cout << "iPtr = " << *iPtr << " (" << iPtr.get() << ")" << endl;
   cout << "zgr  = " << *zgr  << " (" << zgr        << ")" << endl;

   string *str = new string("Hallo");
   {
      auto_ptr<string> sPtr; //... Vorsicht: lokaler auto_ptr
      sPtr = auto_ptr<string>(str);
      cout << "sPtr = " << *sPtr << " (" << sPtr.get() << ")" << endl;
      cout << "str  = " << *str  << " (" << str        << ")" << endl;
   } //....... Speicher von str wird hier freigegeben

   *str = "Neuer Text"; // Fehler: Zugriff auf nicht mehr vorhandenen Speicher
}
```

Programm 5.3 liefert z. B. die folgende Ausgabe:

```
iPtr = 123 (0x804b008)
zgr  = 123 (0x804b008)
iPtr = 321 (0x804b008)
zgr  = 321 (0x804b008)
sPtr = Hallo (0x804b018)
str  = Hallo (0x804b018)
Speicherzugriffsfehler
```

5.1.2 Methoden und Operatoren der Klasse auto_ptr

Mögliche Realisierung der Klasse auto_ptr

```
template<typename T> struct auto_ptr_ref { // wird später behandelt
   explicit auto_ptr_ref(T* p): ptr(p) { }
   T* ptr;
};
template<typename T> class auto_ptr {
 public:
   //.......................................................... Konstruktoren
   explicit auto_ptr(T *p = 0) throw() : ptr(p) { }
   auto_ptr(auto_ptr& a)       throw() : ptr(a.release()) { }
   template<typename X>
   auto_ptr(auto_ptr<X>& a)    throw() : ptr(a.release()) { }
   //............................................................ Destruktor
   ~auto_ptr() { delete ptr; }
   //..................................................... Zuweisungsoperatoren
   auto_ptr& operator=(auto_ptr& a)    throw() { reset(a.release()); return *this; }
   template<typename X>
   auto_ptr& operator=(auto_ptr<X>& a) throw() { reset(a.release()); return *this; }
   //............................................ Dereferenzierungs-Operatoren
   T& operator*()  const throw() { return *ptr; }
   T* operator->() const throw() { return ptr;  }
   //....................................................... get, release, reset
   T*   get() const     throw() { return ptr;                              }
   T*   release()       throw() { T* tmp = ptr; ptr = 0; return tmp;       }
   void reset(T *p = 0) throw() { if (p != ptr) { delete ptr; ptr = p; } }
   //.. auto_ptr_ref-Operationen (werden später in  diesem Kapitel behandelt)
   auto_ptr(auto_ptr_ref<T> ref) throw() : ptr(ref.ptr) { }
   auto_ptr& operator=(auto_ptr_ref<T> ref) throw() {
      if (ref.ptr != this->get()) { delete ptr; ptr = ref.ptr; }
      return *this;
   }
   template<typename X>
   operator auto_ptr_ref<X>() throw() { return auto_ptr_ref<X>(this->release()); }
   template<typename X>
   operator auto_ptr<X>() throw()     { return auto_ptr<X>(this->release());     }
 private:
   T* ptr;
};
```

Die Methoden get(), release() und reset()

- `get()` liefert den im jeweiligen `auto_ptr`-Objekt gespeicherten Zeigerwert.
- `release()` liefert wie `get()` den im jeweiligen `auto_ptr`-Objekt gespeicherten Zeigerwert zurück, setzt allerdings zuvor den Zeigerwert des `auto_ptr`-Objekts auf 0.
- `reset()` gibt den alten Speicher, auf den der im `auto_ptr`-Objekt gespeicherte Zeiger frei und setzt dafür diesen Zeiger auf den neuen als Argument übergebenen Zeigerwert.

Programm 5.4 – `autoptr4.cpp`:
Demoprogramm zu den Methoden get(), release() und reset()

```
#include <memory>
#include <iostream>
using namespace std;

class K {
     friend ostream& operator<<(ostream& os, const K& k) { return os << k.wert; }
  public:
     K(int w) : wert(w) { cout << " K("      << wert << "): " << this << endl; }
     ~K()               { cout << "~K("      << wert << "): " << this << endl; }
     void fkt(void)     { cout << " K::fkt(" << wert << "): " << this << endl; }
  private:
     int wert;
};

int main(void) {
   auto_ptr<K> eins(new K(1)),
               zwei(new K(2));
   cout << "------------------------ Nach Deklarationen" << endl;
   eins->fkt();
   zwei = eins;
   cout << "   (" << zwei.get() << "): " << *zwei << endl;
   cout << "   (" << eins.get() << ")" << endl;
   cout << "------------------------ Nach eins = zwei" << endl;
   eins.reset(new K(111)); // nicht erlaubt: eins = new K(111);
   K *zgr1 = eins.get();
   zgr1->fkt();
   cout << "------------------------ Nach eins.reset(new K(111))" << endl;
   K *zgr2 = eins.release();
   delete zgr2;
   cout << "------------------------ Programmende" << endl;
}
```

Programm 5.4 liefert z. B. die folgende Ausgabe:

```
 K(1): 0x804b008
 K(2): 0x804b018
------------------------ Nach Deklarationen
 K::fkt(1): 0x804b008
~K(2): 0x804b018
   (0x804b008): 1
   (0)
------------------------ Nach eins = zwei
 K(111): 0x804b018
 K::fkt(111): 0x804b018
------------------------ Nach eins.reset(new K(111))
~K(111): 0x804b018
------------------------ Programmende
~K(1): 0x804b008
```

Die Klasse auto_ptr_ref

Da der Kopierkonstruktor und Zuweisungsoperator der Klasse `auto_ptr` ihr Argument modifizieren, kann hier nicht wie sonst üblich eine `const`-Referenz übergeben werden. Folglisch können auch keine `const auto_ptr`-Objekte kopiert oder zugewiesen werden. Damit man aber zumindest `auto_ptr`-Objekte als Parameter oder als Rückgabe bei Funktionen verwenden kann, wurde die Hilfsklasse `auto_ptr_ref` zur impliziten Konvertierung eingeführt.
Programm 5.5 zeigt z. B., dass man sich `auto_ptr`-Objekte auch als Rückgabewerte von Funktionen liefern lassen kann.

Programm 5.5 – `autoptr5.cpp`:
auto_ptr_ref-Objekt als Rückgabewert einer Funktion

```
#include <memory>
#include <iostream>
using namespace std;
auto_ptr<int> res1()  return auto_ptr<int>(new int(123));
int main(void) {
   auto_ptr<int> ap1(res1()); // verwendet res1() zum Initialisieren
   cout << *ap1 << endl;
}
```

Programm 5.5 liefert die folgende Ausgabe:

```
123
```

auto_ptr-Objekte auf Konstanten

Programm 5.6 zeigt den Unterschied zwischen einem konstanten `auto_ptr`-Objekt und einem `auto_ptr`-Objekt, das auf einen konstanten Wert zeigt.

Programm 5.6 – `autoptr6.cpp`:
auto_ptr_ref-Objekt als Rückgabewert einer Funktion

```
#include <memory>
#include <iostream>
using namespace std;
int main(void) {
   int *zgr1 = new int(111);
   const auto_ptr<int> ap1(zgr1); // konstanter Zeiger (Speicher modifizierbar)
   *ap1 = 222;
   cout << "*ap1 = " << *ap1 << endl;
   int *zgr2 = new int(111);
   auto_ptr<const int> ap2(zgr2); // Zeiger auf konstanten Wert
   // *ap2 = 222; // nicht erlaubt
   cout << "*ap2 = " << *ap2 << endl;
}
```

Programm 5.6 liefert die folgende Ausgabe:

```
*ap1 = 222
*ap2 = 111
```

5.1.3 Einschränkungen bei der auto_ptr-Klasse

auto_ptr-Objekte sollten nur auf dynamisch allozierten Speicher zeigen

`auto_ptr`-Objekte sollten nur auf dynamisch allozierten und nicht auf statischen Speicher zeigen, da bei der Zerstörung eines `auto_ptr`-Objekts ja immer der Speicher mit `free` (bzw. `delete`) freigegeben wird, was bei statisch angelegten Speicher zwangsläufig zu einem undefinierten Programmverhalten führt.

Programm 5.7 – `autoptr7.cpp`:
auto_ptr-Objekte sollten nicht statischen Speicher zeigen

```
#include <memory>
#include <iostream>
using namespace std;

int main(void)
{
   int var = 123;
   auto_ptr<int> iPtr(&var); // auto_ptr auf statischen Speicherplatz
   cout << "iPtr = " << *iPtr << " (" << iPtr.get() << ")" << endl;
}
```

Programm 5.7 liefert z. B. die folgende Ausgabe:

```
iPtr = 123 (0xbfffe56c)
*** glibc detected *** free(): invalid pointer: 0xbfffe56c ***
```

auto_ptr-Objekte sollten nicht auf gleiche Speicherbereiche zeigen

Es sollten niemals mehrere `auto_ptr`-Objekte auf einen gleichen, dynamisch allozierten Speicherbereich zeigen, da bei der Zerstörung eines `auto_ptr`-Objekts ja auch immer automatisch der Speicher freigegeben wird, auf den dieses zeigt, so dass die anderen `auto_ptr`-Objekte nun auf eine ungültigen Speicherbereich zeigen, der bereits freigegeben wurde.

Programm 5.8 – `autoptr8.cpp`:
auto_ptr-Objekte sollten nicht auf gleiche Speicherbereiche zeigen

```
#include <memory>
#include <iostream>
using namespace std;

int main(void)
{
   int *zgr = new int(123);
   auto_ptr<int> iPtr1(zgr);
   {
      auto_ptr<int> iPtr2(zgr); // iPtr2 zeigt auf gleichen Speicher wie iPtr1
      cout << "iPtr1 = " << *iPtr1 << " (" << iPtr1.get() << ")" << endl;
      cout << "iPtr2 = " << *iPtr2 << " (" << iPtr2.get() << ")" << endl;
   } //... gemeinsamer Speicher wird hier durch Zerstoerung von iPtr2 freigegeben
   cout << "iPtr1 = " << *iPtr1 << " (" << iPtr1.get() << ")" << endl;
}
```

Programm 5.8 liefert z. B. die folgende Ausgabe:

```
iPtr1 = 123 (0x804a008)
iPtr2 = 123 (0x804a008)
iPtr1 = 0 (0x804a008)
*** glibc detected *** double free or corruption: 0x0804a008 ***
```

auto_ptr-Klasse ist nicht für Arrays ausgelegt

Da die `auto_ptr`-Klasse nur für einfache Objekte und nicht für Arrays ausgelegt ist, wird bei der Freigabe des zugehörigen Speichers immer `delete` und nicht `delete[]` aufgerufen.

Dies kann zu erheblichen Problemen führen, wenn man `auto_ptr` einsetzt, um über sie Arrays zu allozieren. So wird Programm 5.9 entweder zu einem Programmabsturz oder zu einer endlosen Ausgabe der Anzahl aktuell existierenden `X`-Objekte führen.

Programm 5.9 – `autoptr9.cpp`:
Falsche Verwendung von auto_ptr mit Arrays

```
#include <memory>
#include <string>
#include <iostream>
using namespace std;

int z = 0;

class X
{
  public:
      X() : s("Hallo") { ++z; }
      ~X()             { --z; }
      string s;
};

template <class T>
void f( size_t n )
{
   cout << z << ", "; // gib Anzahl von aktuell existierenden X Objekten aus
   auto_ptr<T> p1( new T ); // OK
   auto_ptr<T> p2( new T[n] ); // Fehler (auto_ptr mit new[] statt new initial.)
   cout << z << ", "; // gib Anzahl von aktuell existierenden X Objekten aus
}

int main(void)
{
   while (1)
      f<X>(100);
}
```

auto_ptr-Objekte eignen sich nicht für Containerklassen

`auto_ptr`-Objekte lassen sich nicht normal kopieren oder zuweisen, da in diesem Fall das Original verändert wird (sein Zeiger wird auf 0) gesetzt, so dass danach das Original sich von der Kopie unterscheidet.
Deshalb eignen sich `auto_ptr`-Objekte nicht als Elemente für Containerklassen. Zudem ist auch bei der Deklaration von Funktionen Vorsicht geboten, wie es in Programm 5.10 gezeigt ist.

Programm 5.10 – `autoptr10.cpp`:
auto_ptr-Objekte eignen sich nicht für Containerklassen

```
#include <string>
#include <vector>
#include <memory>
#include <iostream>
using namespace std;
void fkt(auto_ptr<int> p) {
   cout << " fkt: " << *p << "( " << p.get() << ")" << endl;
}
int main(void) {
   auto_ptr<int> iZgr = auto_ptr<int>(new int(123));
   cout << "main: " << *iZgr << "( " << iZgr.get() << ")" << endl;
   fkt(iZgr); // danach ist iZgr == 0
   cout << "main: (" << iZgr.get() << ")" << endl;
   auto_ptr<string> str(new string("Eins"));
   vector<auto_ptr<string> > v;
   // v.push_back(str); // nicht erlaubt, da push_back() const-Parameter fordert
                        //  void push_back(const value_type& x)
}
```

Programm 5.10 liefert z. B. die folgende Ausgabe:

```
main: 123( 0x804b008)
 fkt: 123( 0x804b008)
main: (0)
```

5.2 Die Klasse bitset

Die Klasse `bitset` ermöglicht die Verwaltung von Statusinformationen über eine Bitfolge, wobei anders als bei `unsigned long` die Anzahl der Bits nicht vom System festgelegt ist, sondern mittels eines Template-Arguments frei wählbar ist:

```
template<size_t N>
class bitset {
   ....
};
```

Verwendet man die Klasse `bitset`, muss man folgende Headerdatei inkludieren:

```
#include <bitset>
```

5.2.1 Konstruktoren der Klasse bitset

`bitset()`
: Default-Konstruktor (initialisiert alle Bits mit 0).

`bitset(unsigned long wert)`
: setzt die Bits entsprechend dem `wert`. Hat `wert` mehr Bits als über `N` festgelegt wurden, werden vorne überhängende Bits von `wert` nicht aufgenommen.

`explicit bitset(const string& s, size_t pos=0, size_t n=npos)`
: setzt die Bits entsprechend dem String `s`. Mit `pos` kann der Index des ersten Zeichens und mit `n` die Anzahl der Zeichen angegeben werden, die aus dem String `s` zu übernehmen sind. Der so festgelegte Teilstring darf nur Nullen und Einsen enthalten. Sind andere Zeichen in diesem Teilstring vorhanden, wird die Exception `invalid_argument` ausgelöst. Sollte `pos > s.size()`, wird die Exception `out_of_range` geschickt. Hat der Teilstring mehr Bits als über `N` festgelegt wurden, werden rechts überhängende Zeichen dieses Teilstrings nicht aufgenommen. Hat der Teilstring weniger Bits als über `N` festgelegt wurden, wird links mit Nullen aufgefüllt.

Daneben stellt die Klasse `bitset` noch einen Kopierkonstruktor und Zuweisungsoperator zur Verfügung.

Programm 5.11 – `bitset1.cpp`:
Demoprogramm zu den bitset-Konstruktoren

```
#include <string>
#include <bitset>
#include <exception>
#include <iostream>
using namespace std;

int main(void)
{
   bitset<32> b1;         cout << "bitset<32> b1:         " << b1 << endl;
   cout << "------------------------------------------------------------" << endl;
   bitset<5> b2(6);      cout << "bitset<5> b2(6):      " << b2 << endl;
   bitset<8> b3(0xaffe); cout << "bitset<8> b3(0xaffe): " << b3 << endl;
   cout << "------------------------------------------------------------" << endl;
   string s1("11001001");
   cout << "s1 = " << s1 << endl;
   bitset <5> b4(s1);        cout << "bitset <5> b4(s1):      " << b4 << endl;
   bitset<10> b5(s1);        cout << "bitset<10> b5(s1):      " << b5 << endl;
   bitset <5> b6(s1, 2);     cout << "bitset <5> b6(s1, 2):   " << b6 << endl;
   bitset <5> b7(s1, 3, 4);  cout << "bitset <5> b7(s1, 3, 4):" << b7 << endl;
   cout << "------------------------------------------------------------" << endl;
   string s2("110ab100");
   cout << "s2 = " << s2 << endl;
   try {
      cout << "bitset<5> b8(s2): ";
      bitset<5> b8(s2);
   } catch (const exception& e) {
      cout << e.what() << endl;
   }
```

```
   try {
      cout << "bitset<10> b9(s1, 100): ";
      bitset<10> b9(s1, 100);
   } catch (const exception& e) {
      cout << e.what() << endl;
   }
}
```

Programm 5.11 liefert die folgende Ausgabe:

```
bitset<32> b1:         00000000000000000000000000000000
----------------------------------------------------------
bitset<5> b2(6):      00110
bitset<8> b3(0xaffe): 11111110
----------------------------------------------------------
s1 = 11001001
bitset <5> b4(s1):      11001
bitset<10> b5(s1):      0011001001
bitset <5> b6(s1, 2):   00100
bitset <5> b7(s1, 3, 4):00100
----------------------------------------------------------
s2 = 110ab100
bitset<5> b8(s2): bitset -- string contains characters which are neither 0 nor 1
bitset<10> b9(s1, 100): bitset -- initial position is larger than the string itself
```

5.2.2 Setzen bzw. Modifizieren von Bits

Bei den folgenden Methoden ist zu beachten, dass die Indizes von rechts her gezählt werden:

`bitset& set()`
: setzt alle Bits auf 1 und liefert `*this` als Rückgabe.

`bitset& set(size_t pos, bool val = true)`
: setzt das Bit an der Position `pos` auf 1 bei `val=true` und auf 0 bei `val=false` und liefert `*this` als Rückgabe. Ist `pos` außerhalb des Bereichs `[0,N)`, wird die Exception `out_of_range` geschickt.

`bitset& reset()`
: setzt alle Bits auf 0 und liefert `*this` als Rückgabe.

`bitset& reset(size_t pos)`
: setzt das Bit an der Position `pos` auf 0 und liefert `*this` als Rückgabe. Ist `pos` außerhalb des Bereichs `[0,N)`, wird die Exception `out_of_range` geschickt.

`bitset& flip()`
: invertiert alle Bits und liefert `*this` als Rückgabe.

`bitset& flip(size_t pos)`
: invertiert das Bit an der Position `pos` und liefert `*this` als Rückgabe. Ist `pos` außerhalb des Bereichs `[0,N)`, wird die Exception `out_of_range` geschickt.

Programm 5.12 – `bitset2.cpp`:
Setzen bzw. Modifizieren von Bits

```
#include <string>
#include <bitset>
#include <exception>
#include <iostream>
using namespace std;
int main(void) {
   bitset<8> b;  cout << b << endl;  // 00000000
   b.set();      cout << b << endl;  // 11111111
   b.flip(2);    cout << b << endl;  // 11111011
   for (unsigned int i=0; i < b.size(); i++)
      b.set(i, (i%4==0));            // 00010001
   cout << b << endl << "--------" << endl;
   b.flip();     cout << b << endl;  // 11101110
   for (unsigned int i=0; i < b.size(); i++)
      if (i%2 == 0)                  // 10111011
         b.flip(i);
   cout << b << endl << "--------" << endl;
   for (unsigned int i=0; i < b.size(); i++)
      if (i%2 == 0)                  // 10101010
         b.reset(i);
   cout << b << endl;
   b.reset();  cout << b << endl;    // 00000000
}
```

Programm 5.12 liefert die folgende Ausgabe:

```
00000000
11111111
11111011
00010001
--------
11101110
10111011
--------
10101010
00000000
```

5.2.3 Erfragen von Informationen zu bitset-Objekten

`size_t count() const`
: liefert die Anzahl der gesetzten Bits.

`size_t size() const`
: liefert die Anzahl aller Bits, also den Wert des Template-Arguments `N`.

`bool test(size_t pos) const`
: liefert `true`, wenn das Bit an der Position `pos` gesetzt ist, und ansonsten `false`. Ist `pos` außerhalb des Bereichs `[0,N)`, wird die Exception `out_of_range` geschickt.

`bool any() const`
liefert `true`, wenn mindestens ein Bit gesetzt ist, und ansonsten `false`.

`bool none() const`
liefert `true`, wenn kein Bit gesetzt ist, und ansonsten `false`.

Programm 5.13 – `bitset3.cpp`:
Erfragen von Informationen zu bitset-Objekten

```
#include <bitset>
#include <iomanip>
#include <iostream>
using namespace std;
int main(void) {
   bitset<8> b;  cout << b << " (" << b.count() << ")" << endl;  // 00000000
   for (unsigned int i=0; i < b.size(); i++)
      b.set(i, (i%2==0));                                       // 01010101
   cout << b << " (" << b.count() << ")" << endl;
   cout << b << " (2=" << b.test(2) << ", 3=" << b.test(3) << ")" << endl;
   b.reset(); // 00000000
   cout << boolalpha << b << " (any=" << b.any()
                     << ", none=" << b.none() << ")" <<endl;
   b.set(2);  // 00000100
   cout << b << " (any=" << b.any() << ", none=" << b.none() << ")" <<endl;
}
```

Programm 5.13 liefert die folgende Ausgabe:

```
00000000 (0)
01010101 (4)
01010101 (2=1, 3=0)
00000000 (any=false, none=true)
00000100 (any=true, none=false)
```

5.2.4 Operatoren der Klasse bitset

`reference operator[](size_t pos)`
liefert eine Referenz auf das Bit an der Position `pos`. Ist `pos` außerhalb des Bereichs `[0,N)`, wird die Exception `out_of_range` geschickt. Die Hilfsklasse `reference` wird in Kapitel 5.2.7 vorgestellt.

`bool operator[](size_t pos) const`
liefert das Bit an der Position `pos` Ist `pos` außerhalb des Bereichs `[0,N)`, wird die Exception `out_of_range` geschickt.

`bool operator==(const bitset& rechts) const`
`bool operator!=(const bitset& rechts) const`
liefern `true`, wenn die beiden zu vergleichenden `bitset`-Objekte gleich bzw. ungleich sind, und ansonsten `false`.

`bitset operator~() const { return bitset(*this).flip(); }`
liefert das `bitset`-Objekt, in dem alle Bits invertiert sind.

`bitset& operator<<=(size_t pos)`
: schiebt die Bits um `pos` Positionen nach links, wobei von rechts Nullen nachgezogen werden. Das Ergebnis wird dem `bitset`-Objekt auf der linken Seite zugewiesen und als Rückgabe wird `*this` geliefert.

`bitset& operator>>=(size_t pos)`
: schiebt die Bits um `pos` Positionen nach rechts, wobei von links Nullen nachgezogen werden. Das Ergebnis wird dem `bitset`-Objekt auf der linken Seite zugewiesen und als Rückgabe wird `*this` geliefert.

`bitset& operator&=(const bitset& rechts)`
`bitset& operator|=(const bitset& rechts)`
`bitset& operator^=(const bitset& rechts)`
: führen eine bitweise `AND`-, `OR`- bzw. `XOR`-Verknüpfung zwischen zwei `bitset`-Objekten durch. Das Ergebnis wird dem `bitset`-Objekt auf der linken Seite zugewiesen und als Rückgabe wird `*this` geliefert.

Basierend auf die fünf zuvor vorgestellten zusammengesetzten Zuweisungsoperatoren werden noch die folgenden binären Operatoren angeboten:

```
bitset<N> operator>>(size_t pos) const { return bitset<N>(*this) >>= pos; }
bitset<N> operator<<(size_t pos) const { return bitset<N>(*this) <<= pos; }


template<size_t N> inline
bitset<N> operator&(const bitset<N>& x, const bitset<N>& y)
template<size_t N> inline
bitset<N> operator|(const bitset<N>& x, const bitset<N>& y)
template<size_t N> inline
bitset<N> operator^(const bitset<N>& x, const bitset<N>& y)
```

Programm 5.14 – `bitset4.cpp`:
Operatoren der Klasse bitset

```
#include <string>
#include <bitset>
#include <iomanip>
#include <iostream>
#define COUT(b) cout << #b << "=" << b << endl;
using namespace std;
int main(void) {
   bitset<8> b1;                    COUT(b1)           // b1=00000000
   bitset<8> b2(string("11100100")); cout << b2;       // b2=11100100
   cout << "; b2[0]=" << b2[0] << ", b2[2]=" << b2[2] << endl;
   cout << "-------------------------" << endl;
   b2[0] = b2[1] = 1;               COUT(b2)           // b2=11100111
   cout << "-------------------------" << endl;
   cout << b1 << "==" << b2 << " = " << boolalpha << (b1==b2) << endl;
   cout << b2 << "==" << b2 << " = " << boolalpha << (b2==b2) << endl;
   cout << "-------------------------" << endl;
   b2 <<= 4;                        COUT(b2)           // b2=01110000
   b2 >>= 5;                        COUT(b2)           // b2=00000011
   cout << "-------------------------" << endl;
```

```
   b1 = ~b2;                          COUT(b1)          // b1=11111100
   b1 >>= 1;                          COUT(b1)          // b1=01111110
   cout << "--------------------------" << endl;
   b2 &= b1;                          COUT(b2)          // b2=00000010
   COUT((b1|b2))                                      // b1|b2=01111110
   COUT((b1^b2))                                      // b1^b2=01111100
}
```

Programm 5.14 liefert die folgende Ausgabe:

```
b1=00000000
11100100; b2[0]=0, b2[2]=1
-------------------------
b2=11100111
-------------------------
00000000==11100111 = false
11100111==11100111 = true
-------------------------
b2=01110000
b2=00000011
-------------------------
b1=11111100
b1=01111110
-------------------------
b2=00000010
(b1|b2)=01111110
(b1^b2)=01111100
```

5.2.5 Umwandeln in eine Dezimalzahl bzw. einen String

`unsigned long to_ulong() const`
: liefert zum `bitset`-Objekt die entsprechende Dezimalzahl als `unsigned long`. Falls das Bitmuster nicht als `unsigned long`-Wert darstellbar ist, wird die Exception `overflow_error` geschickt.

`string to_string() const`
: liefert das Bitmuster des `bitset`-Objekts als String; siehe auch Programm 5.14.

Programm 5.15 – `bitset5.cpp`:
Umwandeln in eine Dezimalzahl bzw. einen String

```
#include <bitset>
#include <iomanip>
#include <iostream>
using namespace std;
int main(void) {
   bitset<16> b1(0xaffe);;
   cout << b1 << " = " << b1.to_ulong()
        << " = 0x" << hex << b1.to_ulong()
        << " = 0"  << oct << b1.to_ulong() << endl;
```

```
   string s = b1.to_string<char,char_traits<char>,allocator<char> >();
   cout << s << endl;
}
```

Programm 5.15 liefert die folgende Ausgabe:

```
1010111111111110 = 45054 = 0xaffe = 0127776
1010111111111110
```

5.2.6 Einlesen und Ausgeben von bitset-Objekten

Zum Einlesen und Ausgeben von `bitset`-Objekten sind die beiden folgenden Operatoren überladen:

```
template<class CharT, class Traits, size_t N>
basic_ostream<CharT, Traits>&
operator<<(basic_ostream<CharT, Traits>& os, const bitset<N>& x)
```

Der Ausgabeoperator << konvertiert das `bitset`-Objekt `x` mittels `to_string()` in einen String, den er dann ausgibt.

```
template<class CharT, class Traits, size_t N>
basic_istream<CharT, Traits>&
operator>>(basic_istream<CharT, Traits>& is, bitset<N>& x)
```

Der Eingabeoperator >> liest bis zu `N` Zeichen aus dem Eingabe-Stream `is`, speichert diese in einem temporären `string`-Objekt `str` und wertet dann folgenden Ausdruck aus.
`x = bitset<N>(str)`
Als Rückgabe liefert der Eingabeoperator >> `is`. Der Eingabeoperator beendet das Einlesen, wenn er auf das Dateiende trifft oder ein Zeichen auftritt, das keine `0` oder `1` ist.

5.2.7 Die Hilfsklasse bitset::reference

Da ein Byte die kleinste Einheit ist, die über eine Adresse verfügt, besitzen die einzelnen Bits in einem `bitset`-Objekt keine „echten Adressen“ und somit kann auf diese nicht mittels eines Zeigers oder einer Referenz zugegriffen werden. Um dies doch zu ermöglichen, ist in der `bitset`-Klasse die Hilfsklasse `reference` definiert.

```
class reference {
     friend class bitset;
  public:
      ~reference() { }
      reference& operator=(bool x); // für b[i] = x;
      reference& operator=(const reference& j); // für b[i] = b[j];
      bool operator~() const;                   // für ~b[i]
      operator bool() const;                    // für x = b[i] und if (b[i])
      reference& flip();                        // für b[i].flip();
  private:
     reference();
};
```

5.3 vector<bool>-Objekte

Die im vorherigen Kapitel vorgestellte Klasse `bitset` hat den Nachteil, dass die Anzahl der Bits bereits bei der Definition eines `bitset`-Objekts festgelegt werden muss. Benötigt man eine variable Anzahl von Bits, sollte man `vector<bool>`-Objekte verwenden, die bereits auf Seite 152 vorgestellt wurden.

5.4 Die Klasse complex

Die Klasse `complex` ermöglicht das Arbeiten mit komplexen Zahlen, die aus einem Real- und einem Imaginärteil bestehen. Verwendet man die Klasse `complex`, muss man die folgende Headerdatei inkludieren:

```
#include <complex>
```

Nachfolgend ist eine mögliche Realisierung der Klasse `complex` gezeigt:

```
template<typename T>
class complex {
    public:
      typedef T value_type;
      complex(const T& re = T(), const T& im = T()) : r(re), i(im) {}
      template<typename U>
      complex(const complex<U>&z) : r(z.real()), i(z.imag()) {}

      T real() const { return r; } // liefert Realteil
      T imag() const { return i; } // liefert Imaginärteil

      complex<T>& operator= (const T& t) { r  = t; i = T(); return *this; }
      complex<T>& operator+=(const T& t) { r += t;          return *this; }
      complex<T>& operator-=(const T& t) { r -= t;          return *this; }
      complex<T>& operator*=(const T& t) { r *= t; i *= t;  return *this; }
      complex<T>& operator/=(const T& t) { r /= t; i /= t;  return *this; }

      template<typename U> complex& operator=(const complex<U>& z) {
         r = z.real(); i = z.imag(); return *this;
      }
      template<typename U> complex& operator+=(const complex<U>& z) {
         r += z.real(); i += z.imag(); return *this;
      }
      template<typename U> complex& operator-=(const complex<U>& z) {
         r -= z.real(); i -= z.imag(); return *this;
      }
      template<typename U> complex& operator*=(const complex<U>& z) {
         const T rtmp = r * z.real() - i * z.imag();
         i = r * z.imag() + i * z.real();
         r = rtmp;
         return *this;
      }
```

```
      template<typename U> complex& operator/=(const complex<U>& z) {
         const T rtmp =  r * z.real() + i * z.imag();
         const T n = norm(z);
         i = (i * z.real() - r * z.imag()) / n;
         r = rtmp / n;
         return *this;
      }
    private:
      T  r, i; // Real- und Imaginärteil
};
```

Mit Hilfe der Methoden in der Templateklasse `complex` werden Initialisierungen, Zuweisungen und grundlegende mathematische Operationen zwischen komplexen Zahlen verschiedener Basistypen angeboten.
Für die Gleitpunktypen `float`, `double` und `long double` werden in `complex` Spezialisierungen der Templateklasse `complex` angeboten, um implizite Konvertierungen zwischen diesen Typen zu ermöglichen:

```
template<> class complex<float>       { ... }
template<> class complex<double>      { ... }
template<> class complex<long double> { ... }
```

Programm 5.16 – `complex1.cpp`:
Erstes Demoprogramm zur Klasse complex

```
#include <vector>
#include <complex>
#include <iostream>
using namespace std;
#define COUT(c) cout << #c << " = " << c << endl;

int main(void) {
   complex<double> z0;
   complex<double> z1(1.1);
   complex<double> z2(1.1, 3.3);
   complex<double> z3(z1);
   vector<complex<double> > v;
   v.push_back(z0);
   v.push_back(z1);
   v.push_back(z2);
   v.push_back(z3);
   v.push_back(complex<double>(-2, -5));
   for (unsigned i=0; i < v.size(); i++)
      cout << "z" << i << " = " << v[i] << endl;
   COUT(z1+z2);
   COUT(z1*z2);
   z1 = v[4]; COUT(z1);
   z1 /= z2;
   cout << "z1 /= z2: " << z1 << endl;
   cout << "z1.real=" << z1.real() << ", z1.imag=" << z1.imag() << endl;
}
```

Programm 5.16 liefert die folgende Ausgabe:

```
z0 = (0,0)
z1 = (1.1,0)
z2 = (1.1,3.3)
z3 = (1.1,0)
z4 = (-2,-5)
z1+z2 = (2.2,3.3)
z1*z2 = (1.21,3.63)
z1 = (-2,-5)
z1 /= z2: (-1.54545,0.0909091)
z1.real=-1.54545, z1.imag=0.0909091
```

5.4.1 Globale binäre Operatoren und Vorzeichen

```
//........................................... Globaler operator+()
template<typename T> inline
complex<T>operator+(const complex<T>& x, const complex<T>& y) {
  return complex<T>(x) += y;
}
template<typename T> inline
complex<Tp>operator+(const complex<T>& x, const T& y) {
  return complex<T>(x) += y;
}
template<typename T> inline
complex<T>operator+(const T& x, const complex<T>& y) {
  return complex<T>(x) += y;
}
//................. Globale operator-(), operator*() und operator/()
    //... weitgehend identisch, nur dass statt + das entsprechende
    //... Operatorzeichen verwendet wird

//..................................... Globale Vorzeichenoperatoren
template<typename T> inline complex<T>operator+(const complex<T>& x) { return x; }
template<typename T> inline complex<T>operator-(const complex<T>& x) {
   return complex<T>(-x.real(), -x.imag());
}
```

5.4.2 Globale Vergleichsoperatoren

```
template<typename T> inline
bool operator==(const complex<T>& x, const complex<T>& y) {
   return x.real() == y.real() && x.imag() == y.imag();
}
template<typename T> inline
bool operator==(const complex<T>& x, const T& y) {
   return x.real() == y && x.imag() == T();
}
```

```
template<typename T> inline
bool operator==(const T& x, const complex<T>& y) {
   return x == y.real() && T() == y.imag();
}
template<typename T> inline
bool operator!=(const complex<T>& x, const complex<T>& y) {
   return x.real() != y.real() || x.imag() != y.imag();
}
template<typename T> inline
bool operator!=(const complex<T>& x, const T& y) {
   return x.real() != y || x.imag() != T();
}
template<typename T> inline
bool operator!=(const T& x, const complex<T>& y) {
   return x != y.real() || T() != y.imag();
}
```

5.4.3 Globale mathematische Funktionen

```
//.............................................. Funktionen zum Potenzieren
template<typename T> complex<T> pow(const complex<T>& z, int n);
template<typename T> complex<T> pow(const complex<T>& x, const T& y);
template<typename T> complex<T> pow(const complex<T>& x, const complex<T>& y);
template<typename T> complex<T> pow(const T& x, const complex<T>& y);
//.............................................. Trigonometrische Funktionen
template<typename T> complex<T> cos(const complex<T>& z);
template<typename T> complex<T> cosh(const complex<T>& z);
template<typename T> complex<T> exp(const complex<T>& z);
template<typename T> complex<T> log(const complex<T>& z);
template<typename T> complex<T> log10(const complex<T>& z);
template<typename T> complex<T> sin(const complex<T>& z);
template<typename T> complex<T> sinh(const complex<T>& z);
template<typename T> complex<T> sqrt(const complex<T>& z);
template<typename T> complex<T> tan(const complex<T>& z);
template<typename T> complex<T> tanh(const complex<T>& z);
//.............................................. Konjugiert komplexe Zahl
template<typename T> complex<T> conj(const complex<T>& z) {
   return complex<T>(z.real(), -z.imag()); }
//.................................... Polarkoordinaten (siehe Abbildung 5.1)
template<typename T> inline T real(const complex<T>& z) { return z.real(); }
template<typename T> inline T imag(const complex<T>& z) { return z.imag(); }
template<typename T> inline T abs(const complex<T>& z);
template<typename T> inline T arg(const complex<T>& z) {
   return atan2(z.imag(), z.real());
}
template<typename T> inline T norm(const complex<T>& z); // abs(z)*abs(z)
template<typename T> inline complex<T> polar(const T& rho, const T& theta) {
   return complex<T>(rho * cos(theta), rho * sin(theta));
}
```

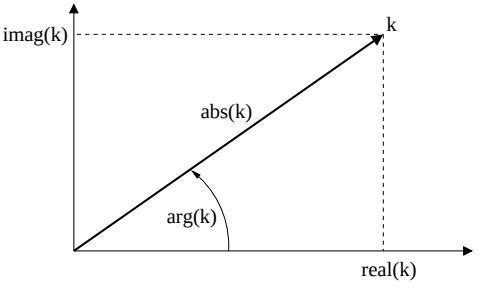


Abbildung 5.1: Zusammenhang der Polarkoordinatenfunktionen

Programm 5.17 – `complex2.cpp`:
Demoprogramm zu den globalen Operatorfunktionen

```
#include <complex>
#include <iomanip>
#include <iostream>
using namespace std;
#define COUT(c) cout << #c << " = " << c << endl;

int main(void) {
   complex<double> z1(1.1, 2.2),
                   z2(20, 0);
   COUT(z1);
   COUT(-4.0 + 5.0 * z1);
   COUT(-z1);
   cout << "------------------------------" << endl;
   cout << z1 << " == " << 1.1  << boolalpha << ": " << (z1 == 1.1)  << endl;
   cout << z2 << " == " << 20.0 << boolalpha << ": " << (z2 == 20.0) << endl;
   cout << "------------------------------" << endl;
   z1 = complex<double>(2, 3);
   COUT(z1);
   COUT(z2);
   COUT(pow(z1, 3));
   COUT(pow(z2, z1));
   cout << "------------------------------" << endl;
   COUT(conj(z1));
   cout << "------------------------------" << endl;
   COUT(z1);
   double rho   = abs(z1);
   double theta = arg(z1);
   cout << "rho=" << rho << ", theta=" << theta << endl;
   complex<double> z3 = polar(rho, theta);
   COUT(z3);
   cout << "------------------------------" << endl;
   COUT(abs(z1)*abs(z1));
   COUT(norm(z1));
}
```

Programm 5.17 liefert die folgende Ausgabe:

```
z1 = (1.1,2.2)
-4.0 + 5.0 * z1 = (1.5,11)
-z1 = (-1.1,-2.2)
------------------------------
(1.1,2.2) == 1.1: false
(20,0) == 20: true
------------------------------
z1 = (2,3)
z2 = (20,0)
pow(z1, 3) = (-46,9)
pow(z2, z1) = (-362.312,169.5)
------------------------------
conj(z1) = (2,-3)
------------------------------
z1 = (2,3)
rho=3.60555, theta=0.982794
z3 = (2,3)
------------------------------
abs(z1)*abs(z1) = 13
norm(z1) = 13
```

5.4.4 Globale Ein- und Ausgabeperatoren >> und <<

```
template<typename T, typename CharT, class Traits> basic_ostream<CharT, Traits>&
operator<<(basic_ostream<CharT, Traits>& os, const complex<T>& x) {
   return os << '(' << x.real() << ',' << x.imag() << ')';
}
template<typename T, typename CharT, class Traits> basic_istream<CharT, Traits>&
operator>>(basic_istream<CharT, Traits>& is, complex<T>& x);
```

Komplexe Zahlen können auf drei verschiedene Arten eingelesen werden:

- **r** setzt Realteil auf `r` und Imaginärteil auf 0.
- **(r)** setzt Realteil auf `r` und Imaginärteil auf 0.
- **(r,i)** setzt Realteil auf `r` und Imaginärteil auf `i`.

Programm 5.18 – `complex3.cpp`:
Demoprogramm zum globalen Eingabeoperator >>

```
#include <complex>
#include <iomanip>
#include <iostream>
using namespace std;
#define COUT(c) cout << #c << " = " << c << endl;
int main(void) {
   complex<double> z;
   cout << "1. Zahl: ";  cin >> z;  COUT(z)
   cout << "2. Zahl: ";  cin >> z;  COUT(z)
   cout << "3. Zahl: ";  cin >> z;  COUT(z)
}
```

Möglicher Ablauf von Programm 5.18:

```
1. Zahl: 2 ⏎
z = (2,0)
2. Zahl: (3) ⏎
z = (3,0)
3. Zahl: (4,5) ⏎
z = (4,5)
```

5.5 Die Klasse numeric_limits

Die Templateklasse `numeric_limits` stellt Informationen über implementierungsspezifische Datentypen zur Verfügung. Dazu wird die Klasse `numeric_limits` für die in der folgenden Tabelle angegebenen Typen spezialisiert:

`bool`			
`char`	`unsigned char`	`signed char`	`wchar_t`
`short`	`unsigned short`		
`int`	`unsigned int`		
`long`	`unsigned long`		
`long long`	`unsigned long long`		
`float`	`double`	`long double`	

Früher wurden hierfür Konstanten aus den Headerdateien `<climits>` (`limits.h`) und `<cfloat>` (`float.h`) wie z. B. `CHAR_BIT` oder `FLT_MIN` benutzt. Wollte man dann z. B. die Anzahl der Bits für `unsigned long` erfragen, musste man `CHAR_BIT * sizeof(unsigned long)` angeben. Mit der Klasse `numeric_limits` lässt sich diese Bitanzahl nun mit `numeric_limits<unsigned long>::digits` ermitteln.
Verwendet man die Klasse `numeric_limits`, muss man die folgende Headerdatei inkludieren:

```
#include <limits>
```

5.5.1 Aufzählungstypen in der Headerdatei <limits>

```
enum float_round_style { // Rundungsart
    round_indeterminate       = -1, // nicht festgelegt
    round_toward_zero         = 0,  // Richtung 0
    round_to_nearest          = 1,  // zum nähesten
    round_toward_infinity     = 2,  // Richtung Unendlich
    round_toward_neg_infinity = 3   // Richtung negativen Unendlich
};
enum float_denorm_style { // Nicht normalisierte Darstellung möglich
    denorm_indeterminate = -1, // nicht festgelegt
    denorm_absent        = 0,  // Nicht normalisierte Darstellung nicht möglich
    denorm_present       = 1   // Nicht normalisierte Darstellung möglich
};
```

5.5.2 Die Templateklasse numeric_limits

Die Basisklasse numeric_limits_base

Die in der folgenden Basisklasse `numeric_limits_base` angegebenen Membervariablen können als ganzzahlige Konstanten verwendet werden, wie z. B. als `case`-Marken oder Arraygrössen. Die voreingestellten Werte `false` und `0` werden in den spezialisierten Klassen verwendet, wenn dort das jeweilige Element für diesen Typ keine Bedeutung hat. Bei wichtigen Membervariablen ist nachfolgend vor diesen eine kurze Erklärung als Kommentar angegeben.

```
class numeric_limits_base {
  public:
    // is_specialized: nur true, wenn für einen Typ (wie z.B. double) eine
    // spezialisierte Klasse vorhanden ist; für vector<int> ist es z.B. false
   static const bool is_specialized = false;
    // digits: - bei ganzzahligen Typen: Anzahl der Bits ohne Vorzeichenbit
    //         - bei Gleitpunkttypen:    Anzahl der Bits für die Mantisse
   static const int  digits = 0;
    // digits10: Anzahl der Ziffern der grössten darstellb. Zahl bei Basis 10
    //           (nur sinnvoll bei is_bounded==true)
   static const int  digits10 = 0;
    // is_signed: nur bei vorzeichenbehafteten Typen true, sonst false
   static const bool is_signed = false;
    // is_integer: nur bei ganzzahligen Typen true, sonst false
   static const bool is_integer = false;
    // is_exact: nur true, wenn Wert genau (ohne Rundungsfehler) im Typ
    //           darstellbar ist (bei ganzzahligen: true, bei Gleitpunkt: false)
   static const bool is_exact = false;
    // radix: - bei Gleitpunkttypen: Basis des Exponenten und bei
    //        - bei ganzzahligen Typen die verwendete Basis (üblicherweise 2)
   static const int  radix = 0;
    // min_exponent: bei Gleitpunkttyp kleinst mögl. Exponenten zur Basis radix
   static const int  min_exponent = 0;
    // min_exponent10: bei Gleitpunkttyp kleinst mögl. Exponenten zur Basis 10
   static const int  min_exponent10 = 0;
    // max_exponent: bei Gleitpunkttyp grösst mögl. Exponenten zur Basis radix
   static const int  max_exponent = 0;
    // max_exponent10: bei Gleitpunkttyp grösst mögl. Exponenten zur Basis 10
   static const int  max_exponent10 = 0;
   static const bool has_infinity = false;
   static const bool has_quiet_NaN = false;
   static const bool has_signaling_NaN = false;
   static const float_denorm_style has_denorm = denorm_absent;
   static const bool has_denorm_loss = false;
   static const bool is_iec559 = false;
    // is_bounded: true, wenn Wertebereich endlich ist
   static const bool is_bounded = false;
    // is_modulo: true, wenn modulo def. ist (bei ganzz. Typen);
    //            false bei bei Gleitpunkttypen
   static const bool is_modulo = false;
```

```
   static const bool traps = false;
   static const bool tinyness_before = false;
   static const float_round_style round_style = round_toward_zero;
};
```

Die von numeric_limits_base abgeleitete Klasse numeric_limits

```
template<typename T>
class numeric_limits : public numeric_limits_base {
  public:
    // min(): liefert kleinsten darstellbaren Wert (nur sinnvoll, wenn
    //        is_bounded || (!is_bounded && !is_signed) gilt)
   static T min()           throw() { return static_cast<T>(0); }
    // max(): liefert grössten darstellb. Wert (nur sinnvoll bei is_bounded==true)
   static T max()           throw() { return static_cast<T>(0); }
    // epsilon(): liefert kleinste Zahl eps, für die gilt: z < z +eps
   static T epsilon()       throw() { return static_cast<T>(0); }
    // round_error(): liefert für Gleitpunkttypen max. mögl. Rundungsfehler
   static T round_error()   throw() { return static_cast<T>(0); }
   static T infinity()      throw() { return static_cast<T>(0); }
   static T quiet_NaN()     throw() { return static_cast<T>(0); }
   static T signaling_NaN() throw() { return static_cast<T>(0); }
   static T denorm_min()    throw() { return static_cast<T>(0); }
};
```

5.5.3 Spezialisierungen zu numeric_limits

Zu den in der Tabelle auf Seite 380 gezeigten Typen sind nun Spezialisierungen in der Headerdatei `<limits>` vorhanden:

```
template<> class numeric_limits<bool>          { ... }
template<> class numeric_limits<char>          { ... }
template<> class numeric_limits<unsigned char> { ... }
.....
template<> class numeric_limits<float>         { ... }
template<> class numeric_limits<double>        { ... }
template<> class numeric_limits<long double>   { ... }
```

Nachfolgend werden zwei Beispiele für spezialisierte Klassen gezeigt.

Die spezialisierte Klasse numeric_limits<bool>

Ein mögliches Aussehen der spezialisierten Klasse `numeric_limits<bool>` wäre z. B.:

```
template<> class numeric_limits<bool> {
  public:
   static const bool is_specialized = true;
   static bool  min() throw() { return false; }
   static bool  max() throw() { return true;  }
   static const int  digits   = 1;
   static const int  digits10 = 0;
   static const bool is_signed  = false;
```

```
   static const bool is_integer = true;
   static const bool is_exact   = true;
   static const int  radix = 2;
   static bool  epsilon()     throw() { return false; }
   static bool  round_error() throw() { return false; }
   static const int  min_exponent   = 0;
   static const int  min_exponent10 = 0;
   static const int  max_exponent   = 0;
   static const int  max_exponent10 = 0;
   static const bool has_infinity  = false;
   static const bool has_quiet_NaN = false;
   static const bool has_signaling_NaN = false;
   static const float_denorm_style has_denorm = denorm_absent;
   static const bool has_denorm_loss = false;
   static bool infinity()      throw() { return false; }
   static bool quiet_NaN()     throw() { return false; }
   static bool signaling_NaN() throw() { return false; }
   static bool denorm_min()    throw() { return false; }
   static const bool is_iec559  = false;
   static const bool is_bounded = true;
   static const bool is_modulo  = false;
   static const bool traps      = true
   static const bool tinyness_before = false;
   static const float_round_style round_style = round_toward_zero;
};
```

Die spezialisierte Klasse numeric_limits<float>

Ein mögliches Aussehen der spezialisierten Klasse `numeric_limits<float>` ist nachfolgend gezeigt, wobei die hier verwendeten `FLT...`-Konstanten in `<cfloat>` definiert sind.

```
template<> class numeric_limits<float> {
  public:
   static const bool is_specialized = true;
   static float min() throw() { return FLT_MIN; }
   static float max() throw() { return FLT_MAX; }
   static const int  digits     = FLT_MANT_DIG;
   static const int  digits10   = FLT_DIG;
   static const bool is_signed  = true;
   static const bool is_integer = false;
   static const bool is_exact   = false;
   static const int  radix      = FLT_RADIX;
   static float epsilon()     throw() { return FLT_EPSILON; }
   static float round_error() throw() { return 0.5F;        }
   static const int  min_exponent   = FLT_MIN_EXP;
   static const int  min_exponent10 = FLT_MIN_10_EXP;
   static const int  max_exponent   = FLT_MAX_EXP;
   static const int  max_exponent10 = FLT_MAX_10_EXP;
   static const bool has_infinity      = true;
   static const bool has_quiet_NaN     = true;
```

```
   static const bool has_signaling_NaN = true;
   static const float_denorm_style has_denorm
                          = FLT_DENORM_MIN ? denorm_present : denorm_absent;
   static const bool has_denorm_loss = false;
   static float infinity()      throw() { return implementierungsspezif. Wert; }
   static float quiet_NaN()     throw() { return implementierungsspezif. Wert; }
   static float signaling_NaN() throw() { return implementierungsspezif. Wert; }
   static float denorm_min()    throw() { return FLT_DENORM_MIN;             }
   static const bool is_iec559
                  = has_infinity && has_quiet_NaN && has_denorm == denorm_present;
   static const bool is_bounded = true;
   static const bool is_modulo  = false;
   static const bool traps      = false;
   static const bool tinyness_before = false;
   static const float_round_style round_style = round_to_nearest;
};
```

Programm zur Ausgabe der Werte von spezialisierten Klassen

Programm 5.19 – `numlimits1.cpp`:
Ausgabe der Werte von spezialisierten Klassen

```
#include <limits>
#include <iomanip>
#include <iostream>
using namespace std;

#define COUT(c) cout << setw(20) << #c << " = " << c << endl;

int main(void)
{
   numeric_limits<long double> k; // long double evtl. durch anderen Typ ersetzen
   cout << setiosflags(ios::left) << boolalpha;
   COUT(k.is_specialized)
   COUT(k.min())
   COUT(k.max())
   COUT(k.digits)
   COUT(k.digits10)
   COUT(k.is_signed)
   COUT(k.is_integer)
   COUT(k.is_exact)
   COUT(k.radix)
   COUT(k.epsilon())
   COUT(k.round_error())
   COUT(k.min_exponent)
   COUT(k.min_exponent10)
   COUT(k.max_exponent)
   COUT(k.max_exponent10)
   COUT(k.has_infinity)
   COUT(k.has_quiet_NaN)
   COUT(k.has_signaling_NaN)
```

```
   COUT(k.has_denorm)
   COUT(k.has_denorm_loss)
   COUT(k.infinity())
   COUT(k.quiet_NaN())
   COUT(k.signaling_NaN())
   COUT(k.denorm_min())
   COUT(k.is_iec559)
   COUT(k.is_bounded)
   COUT(k.is_modulo)
   COUT(k.traps)
   COUT(k.tinyness_before)
   COUT(k.round_style)
}
```

Programm 5.19 liefert z. B. die folgende Ausgabe:

```
k.is_specialized     = true
k.min()              = 3.3621e-4932
k.max()              = 1.18973e+4932
k.digits             = 64
k.digits10           = 18
k.is_signed          = true
k.is_integer         = false
k.is_exact           = false
k.radix              = 2
k.epsilon()          = 1.0842e-19
k.round_error()      = 0.5
k.min_exponent       = -16381
k.min_exponent10     = -4931
k.max_exponent       = 16384
k.max_exponent10     = 4932
k.has_infinity       = true
k.has_quiet_NaN      = true
k.has_signaling_NaN  = true
k.has_denorm         = 1
k.has_denorm_loss    = false
k.infinity()         = inf
k.quiet_NaN()        = nan
k.signaling_NaN()    = nan
k.denorm_min()       = 3.6452e-4951
k.is_iec559          = true
k.is_bounded         = true
k.is_modulo          = false
k.traps              = false
k.tinyness_before    = false
k.round_style        = 1
```

Eigene Spezialisierung von numeric_limits

Hat man z. B. eine eigene Klasse `CBruch` für das Arbeiten mit Brüchen erstellt, kann man dazu eine eigene Spezialisierung von `numeric_limits` zur Verfügung stellen, wie z. B.:

```
class CBruch {
  public:
     ...
  private:
     ...
};

template<> class numeric_limits<CBruch> {
  public:
   static const bool is_specialized = true;
   static float min() throw() { return numeric_limits<int>::min(); }
   static float max() throw() { return numeric_limits<int>::max(); }
   static const int  digits     = numeric_limits<int>::digits;
   static const int  digits10   = numeric_limits<int>::digits10;
   static const bool is_signed  = true;
   static const bool is_integer = false;
   static const bool is_exact   = true;
   static const int  radix      = FLT_RADIX;
   static const bool is_bounded = true;
   static const bool is_modulo  = true;
   //... andere Members für CBruch ohne Bedeutung
};
```

Anwendungsbeispiel zu den spezialisierten Klassen

Programm 5.20 – `numlimits2.cpp`:
Ausgabe einer ganzen Dezimalzahl als Bitmuster

```
#include <limits>
#include <bitset>
#include <iostream>
using namespace std;

int main(void) {
   short zahl;
   cout << "Gib eine ganze Zahl ein: ";
   cin >> zahl;
   cout << " = " << bitset<numeric_limits<short>::digits+1>(zahl) << endl;
}
```

Mögliche Abläufe des Programms 5.20:

```
Gib eine ganze Zahl ein: 45 ↩
 = 0000000000101101
```

```
Gib eine ganze Zahl ein: -7 ↩
 = 1111111111111001
```

5.6 Die Klasse valarray und zugehörige Klassen

Die C++-Standardbibliothek definiert noch die folgenden Klassen:

```
template<class T> class valarray;

class slice;
template<class T> class slice_array;

class gslice;
template<class T> class gslice_array;

template<class T> class mask_array;

template<class T> class indirect_array;
```

Verwendet man eine oder mehrere dieser Klassen, muss man die folgende Headerdatei inkludieren:

```
#include <valarray>
```

5.6.1 Die Klasse valarray

Die wichtigste dieser Klassen in der Headerdatei `<valarray>` ist dabei die Templateklasse `valarray` für eindimensionale Arrays, deren Elemente den Typ `T` besitzen:

```
template<class T>
class valarray
{
  public:
     typedef T value_type;
     //... Methoden (werden nachfolgend vorgestellt)
};
```

Die Klasse `valarray` und die später vorgestellten Klassen sind für schnelle numerische Berechungen, insbesondere auf Parallelrechner ausgelegt.
Als Elemente (vom Typ `T`) können diese Klassen aus `<valarray>` zum einen alle vordefinierten Datentypen (`int`, `float` usw.) sowie alle Klassen aufnehmen, die die üblichen numerischen Operationen unterstützen (wie z. B. `complex<double>` usw.).

Konstruktoren der Klasse valarray

`valarray()`
: Default-Konstruktor; legt leeres `valarray`-Objekt an, dessen Größe später mit der Methode `resize()` entsprechend angepasst werden kann.

`explicit valarray(size_t n)`
: legt ein `valarray`-Objekt mit `n` Elementen an, die mittels des Default-Konstruktors des Typs `T` initialisiert werden.

`valarray(const T& val, size_t n)`
: legt `valarray`-Objekt mit `n` Elementen an, die alle den Wert val enthalten.

`valarray(const T& arr, size_t n)`
: legt ein `valarray`-Objekt mit `n` Elementen an, die mit den ersten `n` Werten des C-Arrays `arr` initialisiert werden. Enthält das C-Array `arr` weniger als `n` Werte, ist das Verhalten undefiniert.

`valarray(const valarray& varr)`
: Kopierkonstruktor

`valarray(const slice_array<T>& sarr)`
`valarray(const gslice_array<T>& garr)`
`valarray(const mask_array<T>& marr)`
`valarray(const indirect_array<T>& iarr)`
: legen ein `valarray`-Objekt mit den Elementen aus dem jeweils übergebenen Array an. Diese vier hier als Parameter angegebenen Klassen werden in den späteren Abschnitten näher vorgestellt.

Programm 5.21 – `valarray1.cpp`:
Demoprogramm zu den valarray-Konstruktoren

```
#include <valarray>
#include <iostream>
using namespace std;
template <typename T>
ostream& operator<< (ostream& os, const valarray<T>& v) {
   for (unsigned i=0; i < v.size(); i++)
      os << v[i] << ", ";
   return os << endl;
}
int main(void) {
   const int carr[] = { 1, 2, 3, 4, 5 };
   valarray<int> eins, zwei(carr+2, 3), drei(5), vier(4, 10), five(vier);
   cout << "eins: " << eins;
   cout << "zwei: " << zwei;
   cout << "drei: " << drei;
   cout << "vier: " << vier;
   cout << "five: " << five;
   eins.resize(10, 8); // Nun 10 Elemente, die alle Wert 8 besitzen
   cout << "eins: " << eins;
}
```

Programm 5.21 liefert die folgende Ausgabe:

```
eins:
zwei: 3, 4, 5,
drei: 0, 0, 0, 0, 0,
vier: 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
five: 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
eins: 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
```

Zuweisungsoperatoren der Klasse valarray

`valarray<T>& operator=(const T& val)`
: weist allen Elementen des `valarray`-Objekts den Wert `val` zu.

`valarray<T>& operator=(const valarray<T>& varr)`
: überschreibt alle Werte des `valarray`-Objekts mit den Werten aus `varr`. Beide `valarray`-Objekte müssen die gleiche Größe besitzen, sonst ist das Verhalten undefiniert.

`valarray<T>& operator=(const slice_array<T>& sarr)`
`valarray<T>& operator=(const gslice_array<T>& garr)`
`valarray<T>& operator=(const mask_array<T>& marr)`
`valarray<T>& operator=(const indirect_array<T>& iarr)`
: weisen dem `valarray`-Objekt die Elemente aus dem jeweils übergebenen Array zu. Sollten dabei Abhängigkeiten zwischen den Elementen des `valarray`-Objekts und denen des jeweiligen übergebenen Array bestehen, ist das Verhalten undefiniert. Diese vier hier als Parameter angegebenen Klassen werden in den späteren Abschnitten näher vorgestellt.

Programm 5.22 – `valarray2.cpp`:
Demoprogramm zu den Zuweisungsoperatoren von valarray

```
#include <valarray>
#include <iostream>
using namespace std;
template <typename T>
ostream& operator<< (ostream& os, const valarray<T>& v) {
   for (unsigned i=0; i < v.size(); i++)
      os << v[i] << ", ";
   return os << endl;
}
int main(void) {
   const int carr[] = { 1, 2, 3, 4, 5 };
   valarray<int> eins(carr, 3), zwei(carr+2, 3);
   cout << "eins: " << eins;   cout << "zwei: " << zwei;
   eins = zwei;
   zwei = 8;
   cout << "eins: " << eins;   cout << "zwei: " << zwei;
}
```

Programm 5.22 liefert die folgende Ausgabe:

```
eins: 1, 2, 3,
zwei: 3, 4, 5,
eins: 3, 4, 5,
zwei: 8, 8, 8,
```

Indexoperatoren der Klasse valarray

`const T& operator[](size_t n) const`
: liefert den Wert des Elements an der Position `n`.

`T& operator[](size_t n)`
: liefert eine Referenz auf das Element an der Position `n`. Diese Referenz ist solange gültig, bis das Element (Objekt) zerstört wird, oder aber die Methode `resize()` für das `valarray`-Objekt aufgerufen wird.

`valarray<T> operator[](slice sarr) const`
`valArray<T> operator[](const gslice& garr) const`
`valarray<T> operator[](const valarray<bool>& varr) const`
`valArray<T> operator[](const valarray<size_t>& varr) const`
ermöglichen Zugriff auf Teilmenge der Elemente des `valarray`-Objekts. Alle liefern dabei ein neues `valarray`-Objekt als Rückgabe.

`slice_array<T> operator[](slice sarr)`
`gslice_array<T> operator[](const gslice& garr)`
`mask_array<T> operator[](const valarray<bool>& varr)`
`indirect_array<T> operator[](const valarray<size_t>& varr)`
ermöglichen Zugriff auf Teilmenge der Elemente des `valarray`-Objekts. Sie liefern dabei Objekte der entsprechenden Hilfsklassen, die Elemente des `valarray`-Objekts referenzieren, als Rückgabe. Diese vier hier als Rückgabe angegebenen Klassen werden in den späteren Abschnitten näher vorgestellt.

Unäre Operatoren der Klasse valarray

`valarray<T> operator+() const`
`valarray<T> operator-() const`
`valarray<T> operator ~() const`
`valarray<bool> operator!() const`
liefern als Rückgabe ein `valarray`-Objekt, auf dessen Elemente der entsprechende unäre Operator angewendet wurde. Diese Operatoren müssen natürlich auf den Typ `T` anwendbar sein.

Programm 5.23 – `valarray3.cpp`:
Demoprogramm zu den unären Operatoren von valarray

```
#include <valarray>
#include <iomanip>
#include <iostream>
using namespace std;
template <typename T>
ostream& operator<< (ostream& os, const valarray<T>& v) {
   for (unsigned i=0; i < v.size(); i++)
      os << v[i] << ", ";
   return os << endl;
}
int main(void) {
   const int  carr1[] = { 1, 2, -3, 4, -5 };
   const bool carr2[] = { true, false, true };
   valarray<int>  eins(carr1, 5), zwei(5);
   valarray<bool> drei(carr2, 3);
   cout << boolalpha;
   cout << "   eins: " << eins;
   zwei = -eins;      cout << "  -eins: " << zwei;
   zwei = ~eins + 1;  cout << "~eins+1: " << zwei;  // Zweier-Komplement
   zwei = +eins;      cout << "  +eins: " << zwei;
   cout << " drei: " << drei;
   drei = !drei;      cout << "!drei: " << drei;
}
```

Programm 5.23 liefert die folgende Ausgabe:

```
   eins: 1, 2, -3, 4, -5,
  -eins: -1, -2, 3, -4, 5,
~eins+1: -1, -2, 3, -4, 5,
  +eins: 1, 2, -3, 4, -5,
 drei: true, false, true,
!drei: false, true, false,
```

Zusammengesetzte Zuweisungsoperatoren der Klasse valarray

`valarray<T>& operator*=(const T& val)`

`valarray<T>& operator/=(const T& val)`

`valarray<T>& operator%=(const T& val)`

`valarray<T>& operator+=(const T& val)`

`valarray<T>& operator-=(const T& val)`

`valarray<T>& operator^=(const T& val)`

`valarray<T>& operator&=(const T& val)`

`valarray<T>& operator|=(const T& val)`

`valarray<T>& operator<<=(const T& val)`

`valarray<T>& operator>>=(const T& val)`

führen die entsprechende Operation für alle Elemente des `valarray`-Objekts mit dem Wert `val` aus und liefern eine Referenz auf das so modifizierte `valarray`-Objekt. Diese Operatoren müssen natürlich auf den Typ `T` anwendbar sein.

`valarray<T>& operator*=(const valarray<T>& varr)`

`valarray<T>& operator/=(const valarray<T>& varr)`

`valarray<T>& operator%=(const valarray<T>& varr)`

`valarray<T>& operator+=(const valarray<T>& varr)`

`valarray<T>& operator-=(const valarray<T>& varr)`

`valarray<T>& operator^=(const valarray<T>& varr)`

`valarray<T>& operator|=(const valarray<T>& varr)`

`valarray<T>& operator&=(const valarray<T>& varr)`

`valarray<T>& operator<<=(const valarray<T>& varr)`

`valarray<T>& operator>>=(const valarray<T>& varr)`

führen die entsprechende Operation für jedes Element des `valarray`-Objekts mit den jeweiligen Element des übergebenen `varr` aus und liefern eine Referenz auf das so modifizierte `valarray`-Objekt. Diese Operatoren müssen natürlich auf den Typ `T` anwendbar sein. Beide `valarray`-Objekte müssen die gleiche Größe besitzen, sonst ist das Verhalten undefiniert. Sollten dabei Abhängigkeiten zwischen den Elementen des `valarray`-Objekts und denen des übergebenen Arrays bestehen, ist das Verhalten undefiniert.

Programm 5.24 – `valarray4.cpp`:
Demoprogramm zu den zusammengesetzten Zuweisungsoperatoren von valarray

```
#include <valarray>
#include <iomanip>
#include <iostream>
using namespace std;

template <typename T>
ostream& operator<< (ostream& os, const valarray<T>& v) {
   for (unsigned i=0; i < v.size(); i++)
      os << setw(5) << v[i];
   return os << endl;
}

int main(void)
{
   const int carr1[] = { 1,  2,  3,  4,  5  };
   const int carr2[] = { 10, 20, 30, 40, 50 };
   valarray<int>  eins(carr1, 5),
                  zwei(carr2, 5);

   cout << "        eins: " << eins;
   cout << "        zwei: " << zwei;
   cout << "zwei += eins: " << (zwei += eins);
   cout << "   zwei *= 3: " << (zwei *= 3);
   cout << "eins += eins: " << (eins += eins);
   cout << "zwei %= eins: " << (zwei %= eins);
}
```

Programm 5.24 liefert die folgende Ausgabe:

```
        eins:     1    2    3    4    5
        zwei:    10   20   30   40   50
zwei += eins:    11   22   33   44   55
   zwei *= 3:    33   66   99  132  165
eins += eins:     2    4    6    8   10
zwei %= eins:     1    2    3    4    5
```

Methoden der Klasse valarray

`size_t size() const`
: liefert die Anzahl der Elemente im `valarray`-Objekt.

`T sum() const`
: liefert unter Verwendung von `operator+=()` die Summe aller Elemente im `valarray`-Objekt.

`T min() const`

`T max() const`
: liefern unter Verwendung von `operator<()` das kleinste bzw. größte Elemente im `valarray`-Objekt.

`valarray<T> shift(int n) const`
: liefert ein `valarray`-Objekt, bei dem die Elemente um `n` Positionen nach vorne verschoben sind.

`valarray<T> cshift(int n) const`
: liefert ein `valarray`-Objekt, bei dem die Elemente um `n` Positionen rotiert sind.

`valarray<T> apply(T func(T val)) const`

`valarray<T> apply(T func(const T& val)) const`
: liefern ein `valarray`-Objekt, bei dem die Elemente durch Aufrufe der Funktion `func()` für jedes Element als Argument zu dieser Funktion initialisiert sind.

`void resize(size_t size, T c = T())`
: setzt die Größe des `valarray`-Objekts auf `size` und initialisiert alle Elemente mit dem Wert `c`, wobei deren alter Inhalt überschrieben wird. Zeiger und Referenzen auf Elemente des `valarray`-Objekts verlieren dadurch ihre Gültigkeit.

Programm 5.25 – `valarray5.cpp`: Demoprogramm zu den Methoden von valarray

```
#include <valarray>
#include <iomanip>
#include <iostream>
using namespace std;

template <typename T>
ostream& operator<< (ostream& os, const valarray<T>& v) {
   for (unsigned i=0; i < v.size(); i++)
      os << setw(3) << v[i];
   return os << endl;
}

int malDrei(int elem) { return elem*3; }

int main(void)
{
   const int carr1[] = { 3, 5, 2, 6, 4, 1 };
   valarray<int>  eins(carr1, 6),
                  zwei;

   cout << "  Summe=" << (eins.sum());
   cout << "; Min="   << (eins.min());
   cout << "; Max="   << (eins.max()) << endl;
   cout << "          eins:" << eins;
   cout << " eins.shift(2):" << (eins.shift(2));
   cout << "eins.cshift(3):" << (eins.cshift(3));
   cout << "               eins:" << eins;
   eins = eins.apply(malDrei);
   cout << "eins.apply(malDrei):" << eins;
   zwei.resize(10, 8); cout << "zwei:" << zwei;
   zwei.resize(5);     cout << "zwei:" << zwei;
}
```

Programm 5.25 liefert die folgende Ausgabe:

```
  Summe=21; Min=1; Max=6
          eins:  3  5  2  6  4  1
 eins.shift(2):  2  6  4  1  0  0
eins.cshift(3):  6  4  1  3  5  2
              eins:  3  5  2  6  4  1
eins.apply(malDrei):  9 15  6 18 12  3
zwei:  8  8  8  8  8  8  8  8  8  8
zwei:  0  0  0  0  0
```

Globale Operatorfunktionen zur Klasse valarray

Für die Klasse `valarray` stehen noch eine Vielzahl von globalen Operatorfunktionen zur Verfügung, die jeweils in dreifacher Ausfertigung angeboten werden.

Globale binäre Operatoren

```
template<typename T>
valarray<T> operator+(const valarray<T>& varr1, const valarray<T>& varr2);
template<typename T>
valarray<T> operator+(const valarray<T>& varr, const T& val);
template<typename T>
valarray<T> operator+(const T& val, const valarray<T>& varr);
//... Diese drei Varianten werden noch für die folgenden Operatoren angeboten:
template<typename T> valarray<T> operator- (...);
template<typename T> valarray<T> operator* (...);
template<typename T> valarray<T> operator/ (...);
template<typename T> valarray<T> operator% (...);
template<typename T> valarray<T> operator& (...);
template<typename T> valarray<T> operator| (...);
template<typename T> valarray<T> operator^ (...);
template<typename T> valarray<T> operator<<(...);
template<typename T> valarray<T> operator>>(...);
```

Zwei `valarray`-Objekte, die mit einem dieser Operatoren verknüpft werden, müssen die gleiche Größe besitzen, ansonsten ist das Verhalten undefiniert. Die Elemente der beiden `valarray`-Objekte werden dabei paarweise mit dem jeweiligen Operator verknüpft, und das Ergebnis wird als `valarray`-Objekt zurückgegeben. Ist ein Argument vom Typ `T`, werden alle Elemente des `valarray`-Objekts mit diesem Argument verknüpft. In diesem Fall muss natürlich der entsprechende Operator für den Typ `T` definiert sein.

Globale Vergleichsoperatoren

```
template<typename T>
valarray<bool> operator==(const valarray<T>& varr1, const valarray<T>& varr2);
template<typename T>
valarray<bool> operator==(const valarray<T>& varr, const T& val);
template<typename T>
valarray<bool> operator==(const T& val, const valarray<T>& varr);
```

```
//... Diese drei Varianten werden noch für die folgenden Operatoren angeboten:
template<typename T> valarray<bool> operator!=(...);
template<typename T> valarray<bool> operator< (...);
template<typename T> valarray<bool> operator<=(...);
template<typename T> valarray<bool> operator> (...);
template<typename T> valarray<bool> operator>=(...);
template<typename T> valarray<bool> operator&&(...);
template<typename T> valarray<bool> operator||(...);
```

Zwei `valarray`-Objekte, die mit einem dieser Operatoren verknüpft werden, müssen die gleiche Größe besitzen, ansonsten ist das Verhalten undefiniert. Die Elemente der beiden `valarray`-Objekte werden dabei paarweise mit dem jeweiligen Operator verglichen, und das Ergebnis wird als `valarray<bool>`-Objekt zurückgegeben. Ist ein Argument vom Typ `T`, werden alle Elemente des `valarray`-Objekts mit diesem Argument verglichen. In diesem Fall muss natürlich der entsprechende Operator für den Typ `T` definiert sein. Auch in diesem Fall wird ein `valarray<bool>`-Objekt zurückgegeben.

Programm 5.26 – `valarray6.cpp`:
Demoprogramm zu den globalen Operatorfunktionen

```
#include <valarray>
#include <iomanip>
#include <iostream>
using namespace std;

template <typename T>
ostream& operator<< (ostream& os, const valarray<T>& v)
{
   for (unsigned i=0; i < v.size(); i++)
      os << setw(6) << v[i];
   return os << endl;
}
int main(void)
{
   const int carr1[] = { 3, 5, 2, 6, 5, 1 };
   const int carr2[] = { 1, 6, 4, 3, 5, 2 };
   valarray<int>  eins(carr1, 6),
                  zwei(carr2, 6),
                  drei(6);
   cout << "       eins:" << eins;
   cout << "       zwei:" << zwei;
   drei = eins * zwei; cout << "eins * zwei:" << drei;
   drei = 10 - zwei;   cout << "10   - zwei:" << drei;
   cout << "-------------------------------------------------" << endl;
   valarray<bool> vergl(6);
   cout << "        eins:" << eins;
   cout << "        zwei:" << zwei;
   vergl = eins <  zwei;  cout << "eins < zwei :" << boolalpha << vergl;
   vergl = eins == zwei;  cout << "eins == zwei:" << boolalpha << vergl;
   vergl = zwei >= 4;     cout << "zwei >= 4   :" << boolalpha << vergl;
}
```

Programm 5.26 liefert die folgende Ausgabe:

```
       eins:     3     5     2     6     5     1
       zwei:     1     6     4     3     5     2
eins * zwei:     3    30     8    18    25     2
10   - zwei:     9     4     6     7     5     8
-------------------------------------------------
        eins:     3     5     2     6     5     1
        zwei:     1     6     4     3     5     2
eins < zwei : false  true  true false false  true
eins == zwei: false false false false  true false
zwei >= 4   : false  true  true false  true false
```

Globale mathematische Funktionen mit einem Parameter

```
template<typename T> valarray<T> abs  (const valarray<T>& varr);
template<typename T> valarray<T> acos (const valarray<T>& varr);
template<typename T> valarray<T> asin (const valarray<T>& varr);
template<typename T> valarray<T> atan (const valarray<T>& varr);
template<typename T> valarray<T> cos  (const valarray<T>& varr);
template<typename T> valarray<T> cosh (const valarray<T>& varr);
template<typename T> valarray<T> exp  (const valarray<T>& varr);
template<typename T> valarray<T> log  (const valarray<T>& varr);
template<typename T> valarray<T> log10(const valarray<T>& varr);
template<typename T> valarray<T> sin  (const valarray<T>& varr);
template<typename T> valarray<T> sinh (const valarray<T>& varr);
template<typename T> valarray<T> sqrt (const valarray<T>& varr);
template<typename T> valarray<T> tan  (const valarray<T>& varr);
template<typename T> valarray<T> tanh (const valarray<T>& varr);
```

Als Ergebnis liefern diese Funktionen ein `valarray`-Objekt, in dem sich jedes Element des übergebenen `valarray`-Objekts befindet, nachdem die entsprechende mathematische Funktion auf jedes Element angewendet wurde.

Globale mathematische Funktionen mit zwei Parametern

```
template<typename T>
valarray<T> pow(const valarray<T>& varr1, const valarray<T>& varr2);
template<typename T>
valarray<T> pow(const valarray<T>& varr, const T& val);
template<typename T>
valarray<T> pow(const T& val, const valarray<T>& varr);


template<typename T>
valarray<T> atan2(const valarray<T>& varr1, const valarray<T>& varr2);
template<typename T>
valarray<T> atan2(const valarray<T>& varr, const T& val);
template<typename T>
valarray<T> atan2(const T& val, const valarray<T>& varr);
```

Zwei `valarray`-Objekte, die mit einer dieser Funktionen verknüpft werden, müssen die gleiche Größe besitzen, ansonsten ist das Verhalten undefiniert. Die Elemente der beiden `valarray`-Objekte werden dabei paarweise mit der jeweiligen Funktion als Argumente aufgerufen, und das Ergebnis wird als `valarray`-Objekt zurückgegeben. Ist ein Argument vom Typ `T`, wird für alle Elemente des `valarray`-Objekts die entsprechende Funktion mit diesem Argument (als erstes bzw. zweites) aufgerufen.

Programm 5.27 – `valarray7.cpp`:
Demoprogramm zu den globalen mathematischen Funktionen

```
#include <valarray>
#include <iomanip>
#include <iostream>
using namespace std;

template <typename T>
ostream& operator<< (ostream& os, const valarray<T>& v) {
   for (unsigned i=0; i < v.size(); i++)
      os << setw(9) << v[i];
   return os << endl;
}
int main(void)
{
   const int    carr1[] = { 2, 3, 4, 5 };
   const int    carr2[] = { 5, 6, 2, 3 };
   const double carr3[] = { 2, 9, 4, 6 };
   const double carr4[] = { 3, 2, 4, 2 };
   valarray<int>  eins(carr1, 4),
                  zwei(carr2, 4),
                  drei(4);
   valarray<double> gltPkt(carr3, 4),
                    hoch(carr4, 4),
                    wurzel(4);

   cout << "            eins:" << eins;
   cout << "            zwei:" << zwei;
   drei   = abs(zwei-eins); cout << "abs(zwei - eins):" << drei;
   cout << "-----------------------------------------------------" << endl;
   cout << "      gltPkt:" << gltPkt;
   wurzel = sqrt(gltPkt); cout << "sqrt(gltPkt):" << wurzel;
   cout << "-----------------------------------------------------" << endl;
   cout << "           gltPkt:" << gltPkt;
   cout << "             hoch:" << hoch;
   gltPkt = pow(gltPkt, hoch); cout << "pow(gltPkt, hoch):" << gltPkt;
   cout << "           gltPkt:" << gltPkt;
   cout << "-----------------------------------------------------" << endl;
   gltPkt = pow(gltPkt, 2.0); cout << " pow(gltPkt, 2.0):" << gltPkt;
   cout << "             hoch:" << hoch;
   gltPkt = pow(4.0, hoch);   cout << "   pow(4.0, hoch):" << gltPkt;
}
```

Programm 5.27 liefert die folgende Ausgabe:

```
            eins:        2        3        4        5
            zwei:        5        6        2        3
abs(zwei - eins):        3        3        2        2
-----------------------------------------------------
      gltPkt:        2        9        4        6
sqrt(gltPkt):  1.41421        3        2  2.44949
-----------------------------------------------------
           gltPkt:        2        9        4        6
             hoch:        3        2        4        2
pow(gltPkt, hoch):        8       81      256       36
           gltPkt:        8       81      256       36
-----------------------------------------------------
 pow(gltPkt, 2.0):       64     6561    65536     1296
             hoch:        3        2        4        2
   pow(4.0, hoch):       64       16      256       16
```

5.6.2 Die Klassen slice und slice_array

Mit einem Objekt der Klasse `slice` lässt sich ab einem Startindex jedes n. te Element eines `valarray`-Objekts referenzieren:

```
class slice
{
  public:
    slice() {} //...legt leeres slice-Objekt an
      //... legt slice-Objekt zu n Indizes an, wobei o der erste Index ist,
      //... und die folg. Indizes immer einen Abstand s zum vorherigen haben.
    slice(size_t o, size_t n, size_t s) : m_offset(o), m_anzahl(n), m_schrittw(s) {}

    size_t start()  const { return m_offset;   }
    size_t size()   const { return m_anzahl;   }
    size_t stride() const { return m_schrittw; }

  private:
    size_t m_offset;   // Startindex
    size_t m_anzahl;   // Anzahl von Indizes
    size_t m_schrittw; // Schrittweite zwischen den Indizes
};
```

Beipiele:

`slice(1, 5, 2)` spezifiziert die Elemente mit den Indizes 1, 3, 5, 7 und 9

`slice(20, 4, -3)` spezifiziert die Elemente mit den Indizes 20, 17, 14 und 11

Die Klasse `slice_array` ist lediglich eine Hilfsklasse, die als Argument beim Konstruktor bzw. Zuweisungsoperator der Klasse `valarray` als Argumenttyp sowie als Rückgabetyp beim Indexoperator der Klasse `valarray` angeboten wird. Ein `slice_array`-Objekt referenziert eine durch ein `slice`-Objekt festgelegte Teilmenge eines `valarray`-Objekts.

```
template<typename T>
class slice_array {
    friend class valarray<T>;
  public:
    slice_array(const slice_array&);             // Kopierkonstruktor
    slice_array& operator=(const slice_array&);  // Zuweisungsoperator
     // (Zusammengesetzte) Zuweisungsoperatoren:
     // weisen den entspr. Arrayelementen die entspr. Elemente von varr zu
    void operator=  (const valarray<T>& varr) const;
    void operator*= (const valarray<T>& varr) const;
    void operator/= (const valarray<T>& varr) const;
    void operator%= (const valarray<T>& varr) const;
    void operator+= (const valarray<T>& varr) const;
    void operator-= (const valarray<T>& varr) const;
    void operator^= (const valarray<T>& varr) const;
    void operator&= (const valarray<T>& varr) const;
    void operator|= (const valarray<T>& varr) const;
    void operator<<=(const valarray<T>& varr) const;
    void operator>>=(const valarray<T>& varr) const;
     // weist den entspr. Arrayelementen den Wert val zu
    void operator=  (const T& val) const;
    ....
  private:
    ....
    slice_array(); // nicht implementiert
};
```

Programm 5.28 – `slice1.cpp`:
Demoprogramm zur Klasse slice

```
#include <string>
#include <valarray>
#include <iomanip>
#include <iostream>
using namespace std;
template <typename T>
ostream& operator<< (ostream& os, const valarray<T>& v) {
   for (unsigned i=0; i < v.size(); i++) os << setw(4) << v[i];
   return os << endl;
}
int main(void) {
   const int carr1[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   const int carr2[] = { 10, 20, 30, 40 };
   valarray<int>    a(carr1, 10), b(carr2, 4);
   valarray<string> x("   ", 10), y("---", 4);
   cout << "                      a: " << a;
   cout << "                      b: " << b;
   cout << setfill('-') << setw(70) << "-" << endl << setfill(' ');
   a[slice(1, 4, 2)]  = b;    cout << "a[slice(1, 5, 2)]  = b  : " << a;
   x[slice(1, 4, 2)]  = y;    cout << "                          " << x;
   a[slice(2, 3, 3)] *= b;    cout << "a[slice(2, 3, 3)] *= b  : " << a;
```

```
   x[slice(0,10, 1)]  = "  ";
   x[slice(2, 3, 3)]  = y;    cout << "                          " << x;
   a[slice(2, 4, 2)]  = 100;  cout << "a[slice(2, 4, 2)]  = 100: " << a;
   x[slice(0,10, 1)]  = "  ";
   x[slice(2, 4, 2)]  = y;    cout << "                          " << x;
}
```

Programm 5.28 liefert die folgende Ausgabe:

```
                      a:    1   2   3   4   5   6   7   8   9  10
                      b:   10  20  30  40
----------------------------------------------------------------------
a[slice(1, 5, 2)]  = b  :    1  10   3  20   5  30   7  40   9  10
                             ---     ---     ---     ---
a[slice(2, 3, 3)] *= b  :    1  10  30  20   5 600   7  40 270  10
                                 ---         ---         ---
a[slice(2, 4, 2)]  = 100:    1  10 100  20 100 600 100  40 100  10
                                 ---     ---     ---     ---
```

Programm 5.29 zeigt, wie man mittels der eindimensionalen Klasse `valarray` ein zweidimensionales Array nachbilden kann.

Programm 5.29 – `slice2.cpp`:
Nachbilden eines zweidimensionales Array bei der Klasse valarray

```
#include <valarray>
#include <iomanip>
#include <iostream>
using namespace std;
template <typename T> ostream& operator<< (ostream& os, const valarray<T>& v) {
   for (unsigned i=0; i < v.size(); i++) {
      os << setw(4) << v[i];
      if ((i+1)%spalten == 0)
         cout << endl;
   }
   return os;
}
const unsigned  zeilen = 4, spalten = 6;
int main(void) {
  unsigned      i, j;
  valarray<int> tabelle(zeilen * spalten);
   for (i = 0; i < tabelle.size(); i++)
      tabelle[i] = (i / spalten + 1) * (i % spalten + 1);
   cout << tabelle << "Zeilen: " << endl;
   for (i = 0; i < zeilen; i++)
      cout << "  zeil[" << i << "] ="
           << valarray<int>(tabelle[slice(i * spalten, tabelle.size()/zeilen, 1)]);
   cout << "Spalten: " << endl;
   for (j = 0; j < spalten; j++)
      cout << "  spalt[" << j << "] ="
           << valarray<int>(tabelle[slice(j, tabelle.size() / spalten, spalten)]);
}
```

Programm 5.29 liefert die folgende Ausgabe:

```
   1   2   3   4   5   6
   2   4   6   8  10  12
   3   6   9  12  15  18
   4   8  12  16  20  24
Zeilen:
  zeil[0] =   1   2   3   4   5   6
  zeil[1] =   2   4   6   8  10  12
  zeil[2] =   3   6   9  12  15  18
  zeil[3] =   4   8  12  16  20  24
Spalten:
  spalt[0] =   1   2   3   4
  spalt[1] =   2   4   6   8
  spalt[2] =   3   6   9  12
  spalt[3] =   4   8  12  16
  spalt[4] =   5  10  15  20
  spalt[5] =   6  12  18  24
```

5.6.3 Die Klassen gslice und gslice_array

Die Klasse `gslice` ermöglicht das Referenzieren mehrerer `slice`-Objekte, so dass man basierend auf einem eindimensionalen `valarray`-Objekt mehrdimensionale Arrays nachbilden kann:

```
class gslice
{
  public:
     gslice(); //...legt leeres gslice-Objekt an
      // legt gslice-Objekt, wobei
      //   o der erste Index ist.
      //   narr: valarray, dessen Elemente die Anzahl der jeweiligen Indizes angeben
      //   sarr: valarray, dessen Elemente die Abstände angeben
     gslice(size_t o, const valarray<size_t>& narr, const valarray<size_t>& sarr);

     size_t           start()  const; // liefert Startindex
     valarray<size_t> size()   const; // liefert narr
     valarray<size_t> stride() const; // liefert sarr
     ....
};
```

Die Klasse `gslice_array` ist lediglich eine Hilfsklasse, die als Argument beim Konstruktor bzw. Zuweisungsoperator der Klasse `valarray` als Argumenttyp sowie als Rückgabetyp beim Indexoperator der Klasse `valarray` angeboten wird. Ein `gslice_array`-Objekt referenziert eine durch ein `gslice`-Objekt festgelegte Teilmenge eines `valarray`-Objekts.

```
template<typename T>
class gslice_array {
    friend class valarray<T>;
  public:
    gslice_array(const gslice_array&);             // Kopierkonstruktor
    gslice_array& operator=(const gslice_array&);  // Zuweisungsoperator
     // (Zusammengesetzte) Zuweisungsoperatoren:
     // weisen den entspr. Arrayelementen die entspr. Elemente von varr zu
    void operator=  (const valarray<T>& varr) const;
    void operator*= (const valarray<T>& varr) const;
    void operator/= (const valarray<T>& varr) const;
    void operator%= (const valarray<T>& varr) const;
    void operator+= (const valarray<T>& varr) const;
    void operator-= (const valarray<T>& varr) const;
    void operator^= (const valarray<T>& varr) const;
    void operator&= (const valarray<T>& varr) const;
    void operator|= (const valarray<T>& varr) const;
    void operator<<=(const valarray<T>& varr) const;
    void operator>>=(const valarray<T>& varr) const;
     // weist den entspr. Arrayelementen den Wert val zu
    void operator=  (const T& val) const;
    ....
  private:
    ....
    gslice_array(); // nicht implementiert
};
```

Programm 5.30 – `gslice1.cpp`:
Demoprogramm zur Klasse gslice

```
#include <valarray>
#include <iomanip>
#include <iostream>
using namespace std;
template<class T>
void print(const valarray<T>& va, int dim1, int dim2) {
    for (unsigned i=0; i<va.size()/(dim1*dim2); ++i) {
        cout << i+1 << ". Gruppe:" << endl;
        for (int j=0; j<dim2; ++j) {
            cout << setw(5) << " ";
            for (int k=0; k<dim1; ++k)
                cout << setw(3) << va[i*dim1*dim2+j*dim1+k];
            cout << endl;
        }
    }
    cout << "------------------"<< endl;
}
int main(void) {
   valarray<double> va(24); // valarray mit 24 Elemente (2 Matrizen: 4 x 3)
    for (int i=0; i<24; i++) // Werte zuweisen
       va[i] = i;
```

```
    print (va, 3, 4); // Ausgabe der beiden Gruppen
    // 2 zweidim. Untermengen von 2 x drei Werten in zwei Arrays mit 12 Elementen
    size_t anz[] = {  2, 3 };
    size_t sw[]  = { 12, 3 };
    valarray<size_t> len(anz,2);
    valarray<size_t> abst(sw, 2);
    // 2. Spalte der ersten 3 Zeilen der 1. Spalte in ersten 3 Zeilen zuweisen
    va[gslice(0, len, abst)] = valarray<double>(va[gslice(1, len, abst)]);
    print(va, 3, 4);
    // 3. Spalte der ersten 3 Zeilen
    // mit 1. Spalte in ersten 3 Zeilen addieren und zuweisen
    va[gslice(0, len, abst)] += valarray<double>(va[gslice(2, len, abst)]);
    print(va, 3, 4);
}
```

Programm 5.30 liefert die folgende Ausgabe:

```
1. Gruppe:
       0  1  2
       3  4  5
       6  7  8
       9 10 11
2. Gruppe:
      12 13 14
      15 16 17
      18 19 20
      21 22 23
------------------
1. Gruppe:
       1  1  2
       4  4  5
       7  7  8
       9 10 11
2. Gruppe:
      13 13 14
      16 16 17
      19 19 20
      21 22 23
------------------
1. Gruppe:
       3  1  2
       9  4  5
      15  7  8
       9 10 11
2. Gruppe:
      27 13 14
      33 16 17
      39 19 20
      21 22 23
------------------
```

Programm 5.31 – `gslice2.cpp`:
Weiteres Demoprogramm zur Klasse gslice

```
#include <valarray>
#include <iomanip>
#include <iostream>
using namespace std;
template<class T>
void print(const valarray<T>& va, int dim1, int dim2) {
   for (unsigned i=0; i < va.size()/(dim1*dim2); ++i)
       for (int j=0; j<dim2; ++j) {
           cout << setw(5) << " ";
           for (int k=0; k<dim1; ++k)
               cout << setw(3) << va[i*dim1*dim2+j*dim1+k];
           cout << endl;
       }
   cout << "----------------------"<< endl;
}
int main(void) {
  valarray<int> va(4 * 6);
   for (unsigned i = 0; i < va.size(); i++)
      va[i] = (i / 6 + 1) * (i % 6 + 1);
   print(va, 6, 4);
    // Eine zweidim. Untermenge von 2 x 4 Werten (mittleren 8 Werte)
   size_t anz2[] = { 2, 4 };
   size_t sw2[]  = { 6, 1 };
   valarray<size_t> len2(anz2, 2);
   valarray<size_t> abst2(sw2, 2);
   valarray<int> va2(2 * 4);
   va2 = valarray<int>(va[gslice(7, len2, abst2)]);
   print(va2, 4, 2);
    // 3., 4. und 5. Spalte
   size_t anz3[] = { 4, 3 };
   size_t sw3[]  = { 6, 1 };
   valarray<size_t> len3(anz3, 2);
   valarray<size_t> abst3(sw3, 2);
   valarray<int> va3(4 * 3);
   va3 = valarray<int>(va[gslice(2, len3, abst3)]);
   print(va3, 3, 4);
    // rechtes unteres Viertel der Matrix
   size_t anz4[] = { 2, 3 };
   size_t sw4[]  = { 6, 1 };
   valarray<size_t> len4(anz4, 2);
   valarray<size_t> abst4(sw4, 2);
   valarray<int> va4(3 * 2);
   va4 = valarray<int>(va[gslice(15, len4, abst4)]);
   print(va4, 3, 2);
}
```

Programm 5.31 liefert die folgende Ausgabe:

```
       1  2  3  4  5  6
       2  4  6  8 10 12
       3  6  9 12 15 18
       4  8 12 16 20 24
----------------------
       4  6  8 10
       6  9 12 15
----------------------
       3  4  5
       6  8 10
       9 12 15
      12 16 20
----------------------
      12 15 18
      16 20 24
----------------------
```

5.6.4 Die Klasse mask_array

Die Klasse `mask_array` ist lediglich eine Hilfsklasse, die als Argument beim Konstruktor bzw. Zuweisungsoperator der Klasse `valarray` als Argumenttyp sowie als Rückgabetyp beim Indexoperator der Klasse `valarray` angeboten wird. Ein `mask_array`-Objekt referenziert die Elemente eines `valarray`-Objekts, die in einem `valarray<bool>`-Objekt den Wert `true` besitzen.

```
template<typename T> class mask_array {
    friend class valarray<T>;
  public:
    mask_array(const mask_array&);             // Kopierkonstruktor
    mask_array& operator=(const mask_array&);  // Zuweisungsoperator
     // (Zusammengesetzte) Zuweisungsoperatoren:
     // weisen den entspr. Arrayelementen die entspr. Elemente von varr zu
    void operator=  (const valarray<T>& varr) const;
    void operator*= (const valarray<T>& varr) const;
    void operator/= (const valarray<T>& varr) const;
    void operator%= (const valarray<T>& varr) const;
    void operator+= (const valarray<T>& varr) const;
    void operator-= (const valarray<T>& varr) const;
    void operator^= (const valarray<T>& varr) const;
    void operator&= (const valarray<T>& varr) const;
    void operator|= (const valarray<T>& varr) const;
    void operator<<=(const valarray<T>& varr) const;
    void operator>>=(const valarray<T>& varr) const;
     // weist den entspr. Arrayelementen den Wert val zu
    void operator=  (const T& val) const;
    ....
  private:
    ....
    mask_array(); // nicht implementiert
};
```

Programm 5.32 – `maskarray.cpp`:
Demoprogramm zur Klasse mask_array

```
#include <valarray>
#include <iomanip>
#include <iostream>
using namespace std;
template<class T>
void print (const valarray<T>& va, int n) {
   for (unsigned i=0; i<va.size()/n; ++i) {
      for (int j=0; j<n; ++j)
         cout << setw(6) << va[i*n+j];
      cout << endl;
   }
   cout << "-----------------------------------" << endl;
}
int main(void) {
    valarray<int> va(4 * 6);
    for (int i=0; i<24; i++)
       va[i] = i;
    print(va, 6);
    // allen durch 5 teilbaren Werten -1 zuweisen
    va[va != 0 && va%5 == 0] = -1;
    print(va, 6);
    // alle Werte im Intervall (5,15) mit 2 multiplizieren
    va[va>5 && va<15] = valarray<int>(va[va>5 && va<15]) *2;
    print(va, 6);
    // boolesches Array zu den Werten im Intervall (5,20)
    valarray<bool>b(va<5 || va>20);
    cout << boolalpha;
    print(b, 6);
    // alle Werte im Intervall (5,20) auf 0 setzen
    va[b] = 0;
    print(va, 6);
}
```

Programm 5.32 liefert die folgende Ausgabe:

```
     0     1     2     3     4     5
     6     7     8     9    10    11
    12    13    14    15    16    17
    18    19    20    21    22    23
-----------------------------------
     0     1     2     3     4    -1
     6     7     8     9    -1    11
    12    13    14    -1    16    17
    18    19    -1    21    22    23
-----------------------------------
     0     1     2     3     4    -1
    12    14    16    18    -1    22
    24    26    28    -1    16    17
    18    19    -1    21    22    23
```

```
-----------------------------------
  true  true  true  true  true  true
 false false false false  true  true
  true  true  true  true false false
 false false  true  true  true  true
-----------------------------------
     0     0     0     0     0     0
    12    14    16    18     0     0
     0     0     0     0    16    17
    18    19     0     0     0     0
-----------------------------------
```

5.6.5 Die Klasse indirect_array

Die Klasse `indirect_array` ist lediglich eine Hilfsklasse, die als Argument beim Konstruktor bzw. Zuweisungsoperator der Klasse `valarray` als Argumenttyp sowie als Rückgabetyp beim Indexoperator der Klasse `valarray` angeboten wird. Ein `indirect_array`-Objekt referenziert die Elemente eines `valarray`-Objekts, deren Indizes es in einem `valarray<size_t>`-Objekt enthält. Dabei sollte der gleiche Index nicht mehrmals enthalten sein.

```
template<typename T>
class indirect_array
{
    friend class valarray<T>;
    friend class gslice_array<T>;

  public:
    indirect_array(const indirect_array&);             // Kopierkonstruktor
    indirect_array& operator=(const indirect_array&);  // Zuweisungsoperator
     // (Zusammengesetzte) Zuweisungsoperatoren:
     // weisen den entspr. Arrayelementen die entspr. Elemente von varr zu
    void operator=  (const valarray<T>& varr) const;
    void operator*= (const valarray<T>& varr) const;
    void operator/= (const valarray<T>& varr) const;
    void operator%= (const valarray<T>& varr) const;
    void operator+= (const valarray<T>& varr) const;
    void operator-= (const valarray<T>& varr) const;
    void operator^= (const valarray<T>& varr) const;
    void operator&= (const valarray<T>& varr) const;
    void operator|= (const valarray<T>& varr) const;
    void operator<<=(const valarray<T>& varr) const;
    void operator>>=(const valarray<T>& varr) const;
     // weist den entspr. Arrayelementen den Wert val zu
    void operator=  (const T& val) const;
    ....
  private:
    ....
    indirect_array(); // nicht implementiert
};
```

Programm 5.33 – `indirectarray.cpp`:
Demoprogramm zur Klasse indirect_array

```
#include <valarray>
#include <iomanip>
#include <iostream>
using namespace std;
template<class T>
void print(const valarray<T>& va, int n) {
   for (unsigned i=0; i < va.size()/n; i++) {
      for (int j=0; j<n; j++)
         cout << setw(3) << va[i*n+j];
      cout << endl;
   }
   cout << "------------------------------" << endl;
}
int main(void) {
    unsigned      i;
    valarray<int> va(4 * 5);
    for (i=0; i < va.size(); i++)
       va[i] = (i / 5 + 1) * (i % 5 + 1);
    print(va, 5);
    valarray<size_t> idx(10); // Anlegen eines Arrays mit Indizes
    for (i=0; i < 10; i++)
       idx[i] = i*2; // nur gerade Indizes
    print(valarray<int>(va[idx]), 10); // Ausgabe aller Elem. mit geraden Indizes
    va[2] = -1;  va[10] = -1; // va[2] und va[10] erhalten neuen Wert
    print(valarray<int>(va[idx]), 10); // Ausgabe aller Elem. mit geraden Indizes
    valarray<size_t> idx2(4); // Anlegen eines Arrays mit Indizes
    idx2[0] =  4;   idx2[1] =  9;
    idx2[2] = 14;   idx2[3] = 19;
    va[idx2] = -2;
    print(va, 5);
}
```

Programm 5.33 liefert die folgende Ausgabe:

```
  1  2  3  4  5
  2  4  6  8 10
  3  6  9 12 15
  4  8 12 16 20
------------------------------
  1  3  5  4  8  3  9 15  8 16
------------------------------
  1 -1  5  4  8 -1  9 15  8 16
------------------------------
  1  2 -1  4 -2
  2  4  6  8 -2
 -1  6  9 12 -2
  4  8 12 16 -2
------------------------------
```

Kapitel 6

Lösungen zu den Übungen

6.1 Strings

6.1.1 Das Focaultsche Pendel

Programm 6.1 – `focault.cpp`:
Das Focaultsche Pendel

```
#include  <string>
#include  <iostream>
using namespace std;

int main(void) {
   string  zeile, satz,
           wortZeichen("abcdefghijklmnopqrstuvwxyzäöü"
                       "ABCDEFGHIJKLMNOPQRSTUVWXYZÄÖÜ");
   while (getline(cin, zeile)) {
      string::size_type start, ende = 0;
      while ((start = zeile.find_first_of(wortZeichen, ende)) != string::npos) {
         char z = (tolower(zeile[start]) - 'a' + 1) % 26 + 'a';
         satz.append(1, z);
         ende = zeile.find_first_not_of(wortZeichen, start+1);
      }
   }
   cout << "...." << satz << endl;
}
```

6.1.2 Ver-/Entschlüsseln eines Gedichts von Ringelnatz

Programm 6.2 – `machebi.cpp`:
Verschlüsseln eines Gedichts nach Ringelnatz

```
#include <string>
#include <iostream>
using namespace std;

int  main(void) {
   string  zeile, vokal("aeiouAEIOU");

   while (getline(cin, zeile)) {
      for (unsigned i=0; i<zeile.length(); i++) {
         cout << zeile[i];
         if (zeile.find_first_of(vokal, i) == i &&
             zeile.find_first_not_of(vokal, i+1) == i+1)
            cout << "bi";
      }
      cout << endl;
   }
}
```

Programm 6.3 – `entferbi.cpp`:
Entschlüsseln eines Gedichts nach Ringelnatz

```
#include <string>
#include <iostream>
using namespace std;

int  main(void) {
   string  zeile,  vokal("aeiouAEIOU");

   while (getline(cin, zeile)) {
      for (unsigned i=0; i<zeile.length(); i++) {
         if (zeile.find_first_of(vokal, i) == i &&
             zeile.substr(i+1, 2) == "bi" )
            zeile.erase(i+1, 2);
      }
      cout << zeile << endl;
   }
}
```

6.1.3 Finden von Zahlwörtern in Strings

Programm 6.4 – `wortadd.cpp`:
Finden von Zahlwörtern in Strings

```
#include  <string>
#include  <iostream>
#include  <iomanip>
using namespace std;

string zahl[] = {
      "null",  "eins",   "zwei", "drei", "vier", "fünf",
      "sechs", "sieben", "acht", "neun", "zehn",  "elf" };
string wort[] = {
       "Endreim",   "Kurzweil",  "Nachtfalter", "Wohnviertel",  "Neunauge",
       "Weinstein", "Erdreich",  "Achtung",     "Segelflieger", "Pfalzwein",
       "Radreifen", "Gehhelfer", "Leinsamen"                                };

#define ZAHLEN_MAX   sizeof(zahl) / sizeof(*zahl)
#define WORT_MAX     sizeof(wort) / sizeof(*wort)

//......................... liefert zu einem String kleingeschriebene Kopie
string toLower(string& s) {
  char *str = new char[s.length()];
  s.copy(str, s.length());
  for (unsigned i = 0; i < s.length(); i++)
     str[i] = tolower(str[i]);
  string s2(str, s.length());
  delete str;
  return s2;
}
//.............................................................. main
int  main(void) {
   unsigned sum=0;

   for (unsigned w=0; w<WORT_MAX; w++) {
      cout << setw(20) << wort[w];
      for (unsigned z=0; z<ZAHLEN_MAX; z++)
         if (toLower(wort[w]).find(zahl[z]) != string::npos) {
            cout << "... " << setw(10) << zahl[z] << "..." << setw(5) << z;
            sum += z;
            break;
         }
      cout << endl;
   }
   cout << "-------------------------------------------------" << endl
        << setw(42) << sum << endl;
}
```

6.1.4 Palindrome durch Addition von Kehrzahlen

Programm 6.5 – `kehrzahl.cpp`:
Palindrome durch Addition von Kehrzahlen

```
#include <string>
#include <iostream>
using namespace std;
string toString(unsigned long zahl) {
   string str;
   while (zahl > 0) {
      str.insert(0, 1, zahl % 10 + '0');
      zahl /= 10;
   }
   return str;
}
int main(void) {
  string  anfang, ende, alt, neu;
   while (true) {
      cout << "Ab welcher Zahl: ";   cin >> anfang;
      if (anfang.find_first_not_of("0123456789") == string::npos)
         break;
      cerr << ".....keine gültige ganze Zahl" << endl;
   }
   while (true) {
      cout << "Bis zu welcher Zahl: ";   cin >> ende;
      if (ende.find_first_not_of("0123456789") == string::npos)
         break;
      cerr << ".....keine gültige ganze Zahl" << endl;
   }
   cout << endl;
   for (long i=atol(anfang.c_str()); i<=atol(ende.c_str()); i++) {
      string alt = toString(i);
      string neu(alt.rbegin(), alt.rend());
      cout << alt << "+" << neu;
      while (1) {
         alt = toString(atol(alt.c_str()) + atol(neu.c_str()));
         neu.assign(alt.rbegin(), alt.rend());
         if (alt == neu ){
            cout << " = " << alt << endl;
            break;
         } else {
            cout << "--> " << alt << "+" << neu;
            if (alt.length() >= 9) {
               cout << "= ..... Abbruch (Zahl zu gross)" << endl;
               break;
            }
         }
      }
   }
}
```

6.2 Die I/O-Stream-Bibliothek

6.2.1 Ausgeben einer Datei ab einer bestimmten Zeile

Programm 6.6 – `abzeile.cpp`:
Datei ab bestimmter Zeile ausgeben

```
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[]) {
   if (argc < 2) {
      cerr << "usage: " << argv[0] << " dateiname [zeilennr]" << endl;
      exit(1);
   }
   ifstream datei(argv[1]);

   if (datei) {
      int zeilennr = (argc >= 3) ? atoi(argv[2]) : 1;
      for (int i=1; i<zeilennr; i++) // Anfangszeilen überspringen
         datei.ignore(1000, '\n');
      cout << datei.rdbuf(); // Rest der Datei auf Standardausg. ausgeben
   } else
      cerr << "Kann Datei '" << argv[1] << "'nicht öffnen" << endl;
}
```

6.2.2 Erstellen eigener Manipulatoren

Programm 6.7 – `eigmanip.cpp`:
Erstellen eigener Manipulatoren

```
#include <string>
#include <iostream>
using namespace std;
//.............................................................. cHeader
ostream& cHeader(ostream &os) {
   return os << "#include <iostream>" << endl
             << "using namespace std;" << endl << endl
             << "int main(void) {" << endl << "}" << endl;
}
//................................................. Manipulator pkte(..)
class pkte {
   int zahl;
 public:
   pkte(int n) : zahl(n) { }
   friend ostream& operator<<(ostream& os, const pkte& p) {
      int  z = p.zahl;
      while (z-- > 0)
         os << ".";
      return os;
   }
};
//..................................... Binäre Ausgabe einer ganzen Zahl
class romzahl {
   string  romZahl;
 public:
   romzahl(unsigned long zahl) {
      struct {
         char  *symbol;
         int   wert;
      } rom[] = { "I",     1, "IV",    4, "V",     5, "IX",    9,
                  "X",    10, "XL",   40, "VL",   45, "IL",   49,
                  "L",    50, "XC",   90, "VC",   95, "IC",   99,
                  "C",   100, "CD",  400, "LD",  450, "XD",  490,
                  "VD",  495, "ID",  499, "D",   500, "CM",  900,
                  "LM",  950, "XM",  990, "VM",  995, "IM",  999, "M",  1000 };
      for (int i=sizeof(rom)/sizeof(*rom)-1; i>=0; i--) {
         for (unsigned j=0; j<zahl/rom[i].wert; j++)
            romZahl.append(rom[i].symbol);
         zahl %= rom[i].wert;
      }
   }
   friend ostream& operator<<(ostream& os, const romzahl& r) {
      return os << r.romZahl;
   }
};
int main(void) { ... }
```

6.3 Die Standard Template Library (STL)

6.3.1 Primzahlen mit dem Sieb des Eratosthenes

Programm 6.8 – `primsieb.cpp`:
Primzahlen mit dem Sieb des Eratosthenes

```
#include  <vector>
#include  <iomanip>
#include  <iostream>
using namespace std;

int main(void) {
   unsigned long i, j, n, z=0;
   vector<bool>  array;

   cout << "Primzahlen bis wohin: ";
   cin >> n;

   for (i=1; i<=n ; i++)
      array.push_back(true);
   for (i=2; i<=n ; i++)
      if (array[i])
         for (j=2*i ; j<=n ; j += i)
            array[j] = false;
   for (i=2; i<=n ; i++)
      if (array[i])
         cout << setw(8) << i << ( (++z%10==0) ? "\n" : "" );
   cout << endl;
}
```

6.3.2 Sortieren von Dateien

Programm 6.9 – `sortfile.cpp`:
Sortieren von Dateien

```
#include  <string>
#include  <vector>
#include  <fstream>
#include  <iostream>
using namespace std;

int main(int argc, char *argv[])
{
   if (argc != 3) {
      cerr << "usage: " << argv[0] << " quelldatei zieldatei" << endl;
      exit(1);
   }
   string zeile;
   vector<string> v;
   ifstream eing(argv[1]); // Öffnen der zu lesenden Datei
   ofstream ausg(argv[2]); // Öffnen der zu schreibenden Datei

   if (eing && ausg) // konnten beide Dateien erfolgreich geÃ¶ffnet werden
      while (getline(eing, zeile))
         v.push_back(zeile);

   for (unsigned i = 0; i < v.size()-1; i++)
      for (unsigned j = i+1; j < v.size(); j++)
         if (v[i] > v[j]) {
            string h = v[i];
            v[i] = v[j];
            v[j] = h;
         }
   for (unsigned z = 0; z < v.size(); z++)
      ausg << v[z] << endl;
}
```

6.3.3 Folgen von Nullen und Einsen

Programm 6.10 – `nulleins.cpp`:
Folgen von Nullen und Einsen

```
#include <cstdlib>
#include <ctime>
#include <deque>
#include <iostream>
using namespace std;

int  main(void)
{
   deque<int> folge1, folge2;
   unsigned   i, laenge, z1, z2;

   cout << "Länge der 0/1-Folge: ";
   cin >> laenge;

   srand(time(0));
   for (i=1; i<=laenge; i++)
      folge1.push_back(rand()%2);

   for (i=0; i< folge1.size(); i++)
      cout << folge1[i];
   cout << endl;
   while (folge1.size() > 1) {
      folge2.clear();
      while (!folge1.empty()) {
         z1 = folge1.front();
         folge1.pop_front();
         z2 = 2; // falls nichts mehr da --> ungleich
         if (!folge1.empty()) {
            z2 = folge1.front();
            folge1.pop_front();
         }
         folge2.push_back(z1 != z2);
      }
      for (i=0; i< folge2.size(); i++)
         cout << folge2[i];
      cout << endl;
      folge1 = folge2;
   }
}
```

6.3.4 Sortieren und Mischen von Listen

Programm 6.11 – `listmisch.cpp`:
Sortieren und Mischen von Listen

```
#include <cstdlib>
#include <ctime>
#include <string>
#include <list>
#include <iomanip>
#include <iostream>
using namespace std;

bool groesser(int l, int r)  return l > r;
//---------------------------------------------------------------------------ausgabe
template <typename T>
void ausgabe(T l, string str) {
   cout << setw(30) << str;
   for (typename T::iterator it = l.begin(); it != l.end(); it++)
      cout << *it << ",";
   cout << endl;
}
//-----------------------------------------------------------------------------main
int main(void)
{
   unsigned i;

   srand(time(0));
   list<int> zahlen1, zahlen2, beide;
   for (i=1; i <= 10; i++) {
      zahlen1.push_back(rand()%10);
      zahlen2.push_back(rand()%10);
   }
   ausgabe(zahlen1, string("zahlen1 (unsortiert): "));
   ausgabe(zahlen2, string("zahlen2 (unsortiert): "));
   zahlen1.sort();
   ausgabe(zahlen1, string("zahlen1 (sortiert): "));
   zahlen2.sort();
   ausgabe(zahlen2, string("zahlen2 (sortiert): "));
   beide = zahlen1;
   beide.merge(zahlen2);
   ausgabe(beide, string("gemischt: "));
   beide.unique();
   ausgabe(beide, string("gemischt (keine doppelten): "));
}
```

6.3.5 Eine Ringliste

Programm 6.12 – `josephus.cpp`:
Eine Ringliste

```
#include <list>
#include <iomanip>
#include <iostream>
using namespace std;

int main(void)
{
   unsigned i, persZahl, nr;
   bool     aufListeAnfang = true;

   cout << "Wie viele Personen: ";
   cin >> persZahl;

   cout << "Der wievielte soll immer ausgesondert werden: ";
   cin >> nr;

   list<int> kreis;
   list<int>::iterator it;

   for (i=1; i <= persZahl; i++)
      kreis.push_back(i);

   cout << "In folgender Reihenfolge wird ausgesondert:" << endl;
   while (!kreis.empty()) {
      if (aufListeAnfang)
         it = kreis.begin();
      for (i=1; i < nr; i++)
         if (++it == kreis.end())
            it = kreis.begin();
      cerr << *it << ", ";
      it = kreis.erase(it);
      aufListeAnfang = (it == kreis.end());
   }
   cout << endl;
}
```

6.3.6 Rückwärtige Ausgabe einer Datei

Programm 6.13 – `stackrueck.cpp`:
Rückwärtige Ausgabe einer Datei

```
#include  <stack>
#include  <fstream>
#include  <iostream>
using namespace std;

int main(int argc, char *argv[])
{
   if (argc != 2) {
      cerr << "usage: " << argv[0] << " datei" << endl;
      exit(1);
   }
   stack<char>  st;
   char zeich;
   ifstream eing(argv[1]); // Öffnen der zu lesenden Datei

   if (eing) { // konnte Dateien erfolgreich geöffnet werden
      while ( eing.get(zeich) )
         st.push(zeich);
      while (!st.empty()) {
        cout << st.top();
        st.pop();
      }
   }
}
```

6.3.7 Kubikzahlen über Polyas Sieb

Programm 6.14 – `polya.cpp`:
Kubikzahlen über Polyas Sieb

```
#include <queue>
#include <iomanip>
#include <iostream>
using namespace std;

int main(void)
{
   unsigned zahlen, z = 0, summe = 0;

   cout << "Wie viele Zahlen: ";
   cin >> zahlen;
   queue<unsigned> queue1, queue2;

   //.... Aufeinanderfolgende Zahlen in queue1 schieben
   for (unsigned i=1; i <= zahlen; i++) {
      queue1.push(i);
      cout << queue1.back() << ", ";
   }
   cout << endl;
   //.... Aus queue1 ausser jeden dritten die Zahlen in queue2 schieben
   while (!queue1.empty()) {
      if ( (z = ++z % 3) != 0) {
         queue2.push(queue1.front());
         cout << queue2.back() << ", ";
      }
      queue1.pop();
   }
   cout << endl;
   //.... Summenfolgen aus queue2 in queue1 schieben
   while (!queue2.empty()) {
      summe += queue2.front();
      queue1.push(summe);
      cout << queue1.back() << ", ";
      queue2.pop();
   }
   cout << endl;
   //.... Aus queue1 ausser jeden zweiten die Zahlen in queue2 schieben
   z = 0;
   while (!queue1.empty()) {
      if ( (z = ++z % 2) != 0) {
         queue2.push(queue1.front());
         cout << queue2.back() << ", ";
      }
      queue1.pop();
   }
   cout << endl;
```

```
   //.... Summenfolgen aus queue2 in queue1 schieben
   summe = 0;
   while (!queue2.empty()) {
      summe += queue2.front();
      queue1.push(summe);
      cout << queue1.back() << ", ";
      queue2.pop();
   }
   cout << endl;
}
```

6.3.8 Das Phänomen gleicher Geburtstage

Programm 6.15 – `gebwahr.cpp`:
Das Phänomen gleicher Geburtstage

```
#include <cstdlib>
#include <ctime>
#include <iterator>
#include <set>
#include <iostream>
using namespace std;

int main(void)
{
   unsigned persZahl, simul, gleich=0;

   srand(time(0));

   set<int> gebtag; // int-Set
   pair<set <int>::iterator, bool> p;

   cout << "Wie viele Personen: ";
   cin >> persZahl;
   cout << "Wie viele Simulationen: ";
   cin >> simul;

   for (unsigned i=1; i<=simul; i++) {
      gebtag.clear();
      for (unsigned j=1; j<=persZahl; j++)
         if (!gebtag.insert( rand()%365 ).second) {
            gleich++;
            break;
         }
   }
   cout << "Die Wahrscheinlichkeit, dass mind. 2 Personen am gleichen Tag" << endl
        << "Geburtstag haben ist: " << (float)gleich / simul * 100 << "%" << endl;
}
```

6.3.9 Umsatzberechnung (Maps in einer Map)

Programm 6.16 – `mapumsatz.cpp`:
Umsatzberechnung (Maps in einer Map)

```
#include <fstream>
#include <map>
#include <iomanip>
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {
   string     str;
   int        monat, jahr;
   double     betrag;

   if (argc !=2) {
      cerr << "usage: " << argv[0] << " umsatzDatei" << endl;
      exit(1);
   }
   ifstream datei(argv[1]); // Input-Stream auf Datei
   if (!datei) {
      cerr << "Kann Datei '" << argv[1] << "'nicht Ã¶ffnen" << endl;
      exit(1);
   }
   map<int, map<int, double> > jahrUmsatz;
   while (!datei.eof()) {
      if (datei >> monat) {
         datei.ignore(100, '/');
         datei >> jahr;
         datei.ignore(100, ':');
         datei >> betrag;
         datei.ignore(100, '\n');
         jahrUmsatz[jahr][monat] += betrag;
      }
   }
   map<int, map<int, double> >::iterator jit;
   map<int, double>::iterator mit;
   cout << setiosflags(ios::fixed) << setprecision(2) << setiosflags(ios::right);
   for (jit = jahrUmsatz.begin(); jit != jahrUmsatz.end(); jit++) {
      double gesamt = 0;
      cout << "..............................." << jit->first << endl;
      for (mit = jahrUmsatz[jit->first].begin();
           mit != jahrUmsatz[jit->first].end();
           mit++) {
         cout << setw(12) << mit->first << ": " <<setw(12) << mit->second <<endl;
         gesamt += mit->second;
      }
      cout << setw(14) << "Gesamt: " << setw(12) << gesamt << endl;
   }
}
```

Literaturverzeichnis

Die Programiersprache C

C Programming Language; *Brian W. Kernighan, Dennis M. Ritchie; Prentice Hall*
Kernighan und Ritchie sind die Erfinder von C und beschreiben in diesem Buch den ersten C-Standard.

http://gcc.gnu.org/
Diese Webseite ist die Homepage für GNU C Compiler `gcc` und `g++`.

C-Programmierung unter Linux, Unix und Windows; *Helmut Herold; millin Verlag*
Dieses Buch begnügt sich nicht mit der Vorstellung der einzelnen C-Sprachelemente, sondern vermittelt auch Einblicke in wichtige Grundlagen der Informatik. Darüber hinaus werden zu den einzelnen C-Konstruktionen effektive Programmiertechniken aus der Praxis und typische Anwendungsgebiete im Detail vorgestellt. Zu den wichtigsten Themen finden sich zudem vielfältige Tipps sowie Hinweise zur Vermeidung von „Fallgruben“, die in C leider nicht allzu selten sind. Mit der im Rahmen dieses Buches entwickelten Grafikbibliothek LCGI (Linux C Graphics Interface) ist eine einfache Grafikprogrammierung unter Linux möglich. Am Ende jedes Kapitels finden sich Übungen, die dem Leser eine Selbstkontrolle der jeweils erworbenen Kenntnisse ermöglichen.

C-Kompaktreferenz; *Helmut Herold; Addison-Wesley Verlag*
Dieses Buch ist eine Kurzfassung zur Programmiersprache C, die für das schnelle Nachschlagen von C-Funktionen, C-Konstrukten und allgemeinen Algorithmen konzipiert wurde. Es beschreibt kurz und prägnant die einzelnen C-Konstrukte und standardisierten Headerdateien. Zudem stellt es wesentliche Programmiertechniken, wichtige Algorithmen und nützliche Programme vor, die beim tagtäglichen Programmieren hilfreich sein können.

Die Programmiersprache C++

The C++-Programming Language; *Bjarne Stroustrup; Addison-Wesley Verlag*
In diesem Buch präsentiert der Autor *Bjarne Stroustrup*, Urheber von C++, die vollständige Spezifikation für die Programmiersprache C++ und die Standardbibliothek.

C++, UML und Design Patterns; *Helmut Herold, Michael Klar, Susanne Klar; Addison-Wesley Verlag*
Dieses 1200 seitige Buch erklärt anhand von über 500 Programmbeispielen nicht nur

sehr tiefgehend die Programmiersprache C++, sondern stellt auch nach der Devise „Füge zusammen, was zusammengehört“ auch die UML *(Unified Modeling Language)* und Design Patterns vor, da ein erfolgreiches Programmieren in C++ Hand in Hand mit dem Entwurf geht.

Effektiv C++ programmieren; *Scott Meyers; Addison-Wesley Verlag*
In diesem Buch werden 50 Richtlinien vorgestellt, die man einhalten sollte, um klaren, korrekten und effizienten C++-Code zu erzeugen.

Mehr Effektiv C++ programmieren; *Scott Meyers; Addison-Wesley Verlag*
Dieses Buch stellt 35 weitere Regeln zu C++ vor, die sehr hilfreich sind, um Programm und Design zu verbessern.

Unified Modeling Language

http://www.uml.org/
Dies ist die Homepage zu UML.

Entwurfsmuster (Design Patterns)

Design Patterns; *E. Gamma, R. Helm, R. Johnson, J. Vlissides; Addison-Wesley Verlag*
Dies ist das Standardwerk zu Design Patterns, das man scherzhaft auch „*The GoF-Book*“ nennt, wobei *GoF* für „*Gang of Four*“ (*Viererbande*) steht. Im GoF-Buch werden 23 Lösungen zu bestimmten Problemklassen präsentiert.

Die C++-Bibliothek Qt zur portablen GUI-Programmierung

http://www.trolltech.com/
Dies ist die Homepage der Firma *Trolltech*, die Qt entwickelt hat.

Das Qt-Buch – Portable GUI-Programmierung unter Linux, Unix, Windows;
Helmut Herold; millin Verlag
Dieses Buch stellt die Qt-Klassenbibliothek vor, mit der sich leicht Grafik-Oberflächen entwerfen lassen, die plattformunabhängig sind und ohne jeglichen Portierungsaufwand unter den heute weit verbreiteten Betriebssystemen Linux/Unix und Windows verwendet werden können.

Allgemein zu Linux/Unix

Linux-Unix Grundlagen; *Helmut Herold; Addison-Wesley Verlag*
Dieses Buch ist eine Einführung in das Betriebssystem Unix und geht insbesondere auf das immer beliebtere und frei verfügbare System Linux ein. Es macht den Leser anhand von leicht nachvollziehbaren Beispielen mit den grundlegenden Linux/Unix-Kommandos und -Konzepten vertraut.

Linux-Unix-Grundlagenreferenz; *Helmut Herold; Addison-Wesley Verlag*
In diesem Buch werden die wichtigsten Linux/Unix-Kommandos nach Anwendungsfällen gegliedert und anhand kleiner Beispiele wird deren typischer Einsatz beschrieben. Eine alphabetische Kommandoübersicht hilft dem Leser, nach einzelnen Kommandos zu suchen und sich so schnell zurecht zu finden.

Linux-Unix Shells; *Helmut Herold; Addison-Wesley Verlag*
Dieses Buch behandelt die fünf heute am weitest verbreiteten Unix-Shells: Bourne-Shell, Korn-Shell, C-Shell, bash und tcsh. Es beschreibt die einzelnen Shells und ihre Konstrukte ausführlich und leicht nachvollziehbar anhand von über 200 Shell-Programmbeispielen, die online über den Verlag zu beziehen sind. Die meisten Unix-Systeme bieten standardgemäß mehrere Shells, manche Systeme wie Linux bieten standardgemäß sogar alle fünf Shells an.

make – das Werkzeug zur automatischen Generierung von Programmen;
Helmut Herold; Addison-Wesley Verlag
Ein Softwarepaket besteht aus mehreren Modulen, die durch ein Netz von Abhängigkeiten miteinander verbunden sind. Nimmt man an einer Stelle dieses Pakets Änderungen vor, werden mit Hilfe von `make` alle von Änderungen betroffenen Module automatisch kompiliert. Anwender, Softwareentwickler und Administratoren erfahren hier alles über die zu Grunde liegende Technik, die Funktionsweise und die Anwendungsgebiete von `make`. Darüber hinaus stellt dieses Buch auch fortgeschrittene Themen wie Techniken für das Projektmanagement oder die automatische Makefile-Generierung vor.

Linux-Unix Kurzreferenz; *Helmut Herold; Addison-Wesley Verlag*
Dieses Buch ist eine Kurzreferenz zu den Büchern *Linux-Unix Grundlagen*, *Linux-Unix Shells*, *make*, *awk & sed*, *lex & yacc* und *Linux-Unix Systemprogrammierung*. Es enthält neben der Beschreibung wichtiger Linux/Unix-Kommandos und -Tools auch eine Kurzfassung zu allen typischen Aufrufformen der zur Systemprogrammierung benötigten Funktionen. Die Systemaufrufe werden dabei ebenso wie alle C-Funktionen nicht nur kurz vorgestellt, sondern oft wird noch ein kleiner Codeausschnitt angegeben, der zeigt, wie diese Funktionen zu verwenden sind.

Systemprogrammierung unter Linux/Unix

Advanced Programming in the Unix Environment; *W. R. Stevens; Addison-Wesley Verlag*
Dies ist das Standardwerk zum Programmieren unter Unix. Es beschreibt die gesamte Breite der Systemaufrufe von BSD4.3 über SVR4 bis zum POSIX-Standard.

Linux-Unix Systemprogrammierung; *Helmut Herold; Addison-Wesley Verlag*
Das Buch wendet sich an alle, die mehr über die Interna von Linux/Unix wissen möchten. Es behandelt die Systemprogrammierung unter Linux/Unix und gibt auch Einblicke in die Datenstrukturen und Algorithmen, um dem interessierten Leser die Realisierung von Systemaufrufen und Betriebssystemkonzepten an einem konkreten System zu verdeutlichen. Aufgrund der über 200 Beispiel- und Übungsprogramme eignet sich dieses Buch sowohl zum Selbststudium als auch zum Nachschlagen. Neu hinzugekommen sind in der dritten Auflage ein Kapitel zur Netzwerkprogrammierung mit Sockets und ein Kapitel zur Threadprogrammierung.

Index