# 24

# Algorithmen für verschiedene Aufgaben

Dieses Kapitel behandelt die folgenden Themen:

- Algorithmen für verschiedene Zwecke
- Auf Folgen arbeitende Algorithmen
- Komplexe Zahlen in C++
- Zufallszahlen
- Grenzwerte von Zahltypen ermitteln
- 2-dimensionale Matrix mit zusammenhängendem Speicher

Für viele Probleme gibt es bereits fertige Lösungen. Dieses Kapitel hat seinen Schwerpunkt in den Lösungen, die von der C++-Standardbibliothek angeboten werden. Um die Anwendung zu erleichtern, gibt es zu fast jedem Rezept ein konkretes Beispiel.

**Hinweis**

Wenn nicht anders angegeben, gilt für die Algorithmen dieses Kapitels der Header `<algorithm>`.

Das Verständnis mancher Tipps setzt die Kenntnis der Wirkungsweise der STL voraus (Kapitel 11, 28, 29 und 30). Manche Algorithmen verwenden möglicherweise Ihnen unbekannte Funktionen. Beschreibungen dazu können Sie mithilfe des Registers finden.

# 24.1 Algorithmen mit Strings

**Hinweis**

Eine Menge Algorithmen, die mit Objekten oder Parametern der Klasse `string` arbeiten, finden Sie in Kapitel 32. Abschnitt 31.2 verhilft bei Bedarf zu besserem Verständnis der sprachabhängigen Funktionen.

## 24.1.1 String splitten

Auf Seite 643 wird ein C-String zerlegt. Der C-String wird dabei modifiziert. In diesem Abschnitt wird etwas Ähnliches unternommen, aber für echte `string`-Objekte und ohne deren Zerstörung. In Java gibt es eine nützliche Funktion `split(regex)`, die ein String-Array zurückgibt. `regex` ist ein regulärer Ausdruck mit den Trennzeichen. An dieser Funktion habe ich mich orientiert, wobei an die Stelle von `regex` ein String mit den Trennzeichen tritt. Die C++-Entsprechung des Java-Aufrufs `String[] arr = text.split("[ ,.]");` ist `split(text, ",.", v);`, wobei `v` der Vektor ist, in dem das Ergebnis abgelegt werden soll. Falls das Trennzeichen nicht vorkommt, enthält der Vektor im einzigen Element den ganzen String.

**Listing 24.1:** String splitten

```
// cppbuch/k24/strings/split.cpp
#include<iostream>
#include<string>
#include<vector>
#include<showSequence.h>

void split(const std::string& s, const std::string& trenn,
           std::vector<std::string>& v) {
   size_t start = 0;
   size_t pos;
   do {
      pos = s.find_first_of(trenn, start);
      v.push_back(s.substr(start, pos-start));
      start = pos+1;
   } while(pos != std::string::npos);
}

using namespace std;

int main() {
   string text("The quick brown fox jumps over the lazy dog's back.");
   vector<string> v;
   split(text, " ,", v);
   showSequence(v, "", "*\n");
}
```

Einfacher geht es, wenn die Boost-Library installiert ist:

**Listing 24.2:** String splitten mit Boost

```
// cppbuch/k24/strings/boostsplit.cpp
#include<iostream>
#include<string>
#include<vector>
#include<boost/algorithm/string/split.hpp>
#include<boost/algorithm/string/classification.hpp> // is_any_of
#include<showSequence.h>

using namespace std;
using namespace boost::algorithm;

int main() {
   string text("The quick brown fox jumps over the lazy dog's back.");
   vector<string> v;
   split(v, text, is_any_of(" ,."));
   showSequence(v, "", "*\n");
}
```

### 24.1.2 String in Zahl umwandeln

Für die häufige Aufgabe, einen String in eine Zahl umwandeln zu müssen, gibt es verschiedene Lösungsansätze, die im Folgenden beschrieben werden.

- Funktionen der Standardbibliothek, die mit `string`-Parametern arbeiten.
- Den String in einen Stream schreiben und als Zahl wieder auslesen.
- Funktion `lexical_cast` der Boost-Library.
- Funktionen der C-Library.
- Eigene Umwandlungsfunktion.

#### string-Funktionen der Standardbibliothek

Die Funktionsnamen fangen mit `sto` (für *string to*) an, gefolgt von einem oder zwei Buchstaben für den Typ (`i` für `int`, `ul` für `unsigned long` usw.). Eine Aufzählung finden Sie auf Seite 847; hier seien exemplarisch nur die Umwandlungen in eine `double`- und eine `int`-Zahl gezeigt. Die Schnittstelle für `stoi()`:

```
int stoi(const string& str,  // der umzuwandelnde String
         size_t* endeptr = 0,  // dort wird die erste nicht mehr ausgewertete Position
                               // hinterlegt
         int base = 10);       // Basis der Umwandlung
```

Die Funktionen liefern eine Exception, wenn eine Konversion nicht möglich oder die Zahl außerhalb des Bereichs für den Typ ist. So ist zum Beispiel die Zeichenkette *FF* nur als Hexadezimal interpretierbar, und *FFFFFFFF* überschreitet den Bereich einer 32-Bit-`int`-Zahl. Das folgende Programm zeigt die Möglichkeiten:

**Listing 24.3:** String mit std-Funktionen in Zahl umwandeln

```
// cppbuch/k24/strings/string2zahl/std.cpp
#include<iostream>
#include<string>
```

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

int main() {
   cout << "Bitte Zeichenkette eingeben (z.B. 3.14 12345 0x78 FF 0123): ";
   string zeile;
   getline(cin, zeile);
   size_t endpos; // erste nicht mehr ausgewertete Position
   int modus[] = {-1, 0, 2, 8, 10, 16};
   string modusText[] = {"double",                  // -1
                         "Standard (0 okt/0x hex/dez)", // 0
                         "Binärzahl",               // 2
                         "Oktalzahl",               // 8
                         "Dezimalzahl",             // 10
                         "Hexadezimalzahl"};        // 16
   size_t anzahl = sizeof(modus)/sizeof(modus[0]);
   for(size_t i = 0; i < anzahl; ++i) {
      try {
         cout << "Interpretation als " << modusText[i] << ": ";
         if(-1 == modus[i]) { // double
            cout << stod(zeile, &endpos);
         }
         else {               // int
            cout << stoi(zeile, &endpos, modus[i]);
         }
         if(endpos < zeile.length()) {
            cout << " nicht ausgewertet: " << (zeile.c_str() + endpos);
         }
         cout << endl;
      } catch(const invalid_argument&) {
         cerr << "Konversion ist nicht möglich!" << endl;
      } catch(const out_of_range&) {
         cerr << "Zahl ist außerhalb des Bereichs für diesen Typ!" << endl;
      }
   }
}
```

## Umwandlung mit stringstream

Diese Variante nutzt aus, dass in einen Stringstream wie nach `cout` geschrieben werden kann. Anschließend wird er wie ein Eingabe-Stream (wie `cin`) verwendet. Bei nicht konvertierbaren Strings gibt es keine Exception.

**Listing 24.4:** String mit stringstream in Zahl umwandeln

```
// cppbuch/k24/strings/string2zahl/sstream.cpp
#include<iostream>
#include<string>
#include<sstream>
using namespace std;

int main() {
   cout << "Bitte Zeichenkette eingeben (z.B. 3.14 12345) : ";
```

```
   string zeile;
   getline(cin, zeile);
   stringstream wandler;
   wandler << zeile; // String schreiben
   double d;
   wandler >> d;      // als double-Zahl lesen
   cout << "Zahl =" << d << endl;
}
```

### Umwandlung mit Boost-Funktion

Wenn Sie Boost installiert haben, empfehle ich die Funktion `lexical_cast<T>(arg)`. Sie wandelt den Parameter `arg` in den gewünschten Typ `T` um. Bei Konvertierungsfehlern wird eine Exception geworfen. Auch wird erwartet, dass ein String vollständig umwandelbar ist – einen Zeiger auf den nicht konvertierten Rest gibt es nicht.

**Listing 24.5:** String mit lexical_cast in Zahl umwandeln

```
// cppbuch/k24/strings/string2zahl/boost.cpp
#include<iostream>
#include<string>
#include<boost/lexical_cast.hpp>
using namespace std;

int main() {
   cout << "Bitte Zeichenkette eingeben (z.B. 3.14 12345) : ";
   string zeile;
   getline(cin, zeile);
   try {
      double d = boost::lexical_cast<double>(zeile);
      cout << "Zahl =" << d << endl;
   }
   catch(boost::bad_lexical_cast & blc) {
      cerr << blc.what() << endl;
   }
}
```

### C-Funktionen

Die folgenden Funktionen setzen das Einbinden des Headers `<cstdlib>` voraus. Sie stammen aus der Sprache C und sind Grundlage der Implementierung der oben beschriebenen Funktionen, die mit `string`-Objekten arbeiten (`stoi()` usw.).

- `long strtol(const char* s, char* rest, int basis)` wandelt den C-String `s` in eine `long`-Zahl um und gibt diese zurück. Falls `s` nach der Zahl noch weitere, nicht konvertierbare Zeichen enthält, werden sie in `rest` abgelegt, sofern `rest` ungleich NULL ist. `basis` ist die Basis.
- `unsigned long strtoul(const char* s, char* rest, int basis)` macht dasselbe für `unsigned long`-Zahlen.
- `double strtod(const char* s, char* rest)` leistet Entsprechendes für `double`-Zahlen.

- `atoi(const char* s)` entspricht `(int) strol(s, NULL, 10)`.
- `atol(const char* s)` entspricht `strol(s, NULL, 10)`.
- `atof(const char* s)` entspricht `strtod(s, NULL)`.

Die Funktionen können für ein `string`-Objekt `einString` mit Hilfe der Elementfunktion `c_str()` genutzt werden, also etwa `strol(einString.c_str(), NULL, 10)`. Im Fehlerfall wird 0 zurückgegeben. Für die Angabe der Basis gibt es einige Regeln:

- Basis 0: Die Basis wird entsprechend den Konventionen der Programmiersprache ermittelt: Die Zahl 0777 wird als Oktalzahl, die Zahl 0x1abc als Hexadezimalzahl interpretiert. Alle anderen Zahlen werden als Dezimalzahlen betrachtet.
- Basis 8 : Die Zahl wird in jedem Fall als Oktalzahl interpretiert. 0777 und 777 ergeben dasselbe.
- Basis 16 : Die Zahl wird in jedem Fall als Hexadezimalzahl interpretiert. 0x1abc und 1abc ergeben dasselbe.

Für die Basis 8 wird 1abc daher 1 ergeben, mit abc als Rest. Natürlich sind auch andere Basen möglich. Die Eingabe 1000 mit der Basis 2 führt zum Ergebnis 8. Das folgende Programm zeigt die verschiedenen Interpretationen für einen eingegebenen String an:

**Listing 24.6:** String mit C-Funktionen in Zahl umwandeln

```
// cppbuch/k24/strings/string2zahl/cstdlib.cpp
#include<iostream>
#include<cstdlib>
using namespace std;

int main() {
   cout << "Bitte Zeichenkette eingeben (z.B. 3.14 12345 0x78 FF 0123): ";
   char eingabe[80];
   cin.get(eingabe, sizeof(eingabe));
   char* rest;
   int modus[] = {-1, 0, 2, 8, 10, 16};
   string modusText[] = {"double", "Standard (0 okt/0x hex/dez)",
                         "Binärzahl", "Oktalzahl", "Dezimalzahl",
                         "Hexadezimalzahl"};
   for(size_t i = 0; i < sizeof(modus)/sizeof(modus[0]); ++i) {
      cout << "Interpretation als " << modusText[i] << ": ";
      if(-1 == modus[i]) { // double
         cout << strtod(eingabe, &rest);
      }
      else {               // long
         cout << strtol(eingabe, &rest, modus[i]);
      }
      if(*rest) {
         cout << " nicht ausgewertet:" << rest;
      }
      cout << endl;
   }
}
```

### Eigene Funktion

Da es die anderen Funktionen gibt, ist eine eigene Funktion nicht notwendig. Die Funktion `s2i()` soll daher nur zeigen, wie etwa `stoi()` intern funktionieren könnte, insbesondere die Kontrolle des Überlaufs. Zur Vereinfachung wird von einer Dezimalzahl ausgegangen und die Information über den nicht ausgewerteten Rest wird nicht übergeben. In der Datei *cppbuch/k24/strings/string2zahl/s2i.cpp* finden Sie die Funktion mit einem Anwendungsbeispiel.

## 24.1.3 Zahl in String umwandeln

Ähnlich wie oben gibt es einige Varianten. Zwar ist die Richtung der Umwandlung umgekehrt, weil oben Strings in Zahlen konvertiert werden, aber es gibt strukturelle Ähnlichkeiten. Aus diesem Grund fällt die Darstellung etwas kürzer aus. Beispiele finden Sie im Verzeichnis *cppbuch/k24/strings/zahl2string/*.

- Funktionen `to_string(zahl)` der Standardbibliothek. `zahl` steht dabei für einen Parameter eines beliebigen Zahlentyps.
- Die Zahl in einen Stream schreiben und als String wieder auslesen, ähnlich wie oben (Seite 626). Das Listing auf Seite 393 zeigt ein ausführliches Beispiel, wie Zahlen formatiert in einem String abgelegt werden können.
- Funktion `lexical_cast` der Boost-Library, zum Beispiel

```
string ergebnis = boost::lexical_cast<string>(zahl);
```

- Eigene Umwandlungsfunktion. Wenn auf eine Formatierung verzichtet werden soll, kann für `int`-Zahlen die einfache Funktion `string i2string(int)` genommen werden, die Sie in *cppbuch/k24/strings/zahl2string/i2s.cpp* finden und die auf Seite 498 abgedruckt ist.

## 24.1.4 Strings sprachlich richtig sortieren

Mit der Standard-Sprachumgebung »C« kommen entsprechend der ASCII-Tabelle erst die Großbuchstaben, dann die Kleinbuchstaben, dann alle Werte danach bei 8-Bit-Zeichen. Bei einer ISO-8859-1-Codierung würden Worte, die mit Umlauten beginnen, bei dieser Sortierung erst nach den Kleinbuchstaben kommen. Im Deutschen gibt es für die Sortierung andere Regeln:

- Sie erfolgt unabhängig von der Groß- bzw. Kleinschreibung eines Worts.
- Bei sonst gleichen Wörtern kommen Kleinbuchstaben vor den Großbuchstaben.
- Umlaute wie ä, ö, ü und die zugehörigen Vokale wie a, o, u sind gleichrangig.
- Der Buchstabe ß wird wie ss einsortiert.
- Falls es zwei Wörter gibt, die sich nur dadurch unterscheiden, dass der Buchstabe ß durch ss ersetzt wurde, wird das Wort mit ss zuerst eingeordnet.

Ein Vergleich zweier Strings mithilfe der ASCII-Tabelle würde fehlerhafte Ergebnisse liefern, weil zum Beispiel »ü« im ASCII nicht definiert ist. In C++ kann die Sortierung korrekt durchgeführt werden, wenn die passende Sprachumgebung mitgegeben wird. Das Beispiel zeigt, wie ein `locale`-Objekt zur sprachlich korrekten Sortierung eingesetzt wird:

**Listing 24.7:** Sprachlich richtige Sortierung bei ISO8859-Codierung

```
// cppbuch/k24/strings/richtigsortierenISO8859.cpp
// Dieses Datei ist ISO 8859-1 codiert.
#include<algorithm>
#include<iostream>
#include<vector>
#include<showSequence.h> // siehe Seite 648
using namespace std;

int main() {
   vector<string> v;
   v.push_back("Maße");
   v.push_back("Masse");
   v.push_back("Ähnlich");
   v.push_back("Alphabet");
   v.push_back("aal");
   v.push_back("ähnlich");
   v.push_back("alphabet");
   v.push_back("Aal");
   cout << "vorher:";
   showSequence(v);
   sort(v.begin(), v.end());
   cout << "nach Sortieren ohne Compare-Objekt:\n";
   showSequence(v);
   locale deutsch("de_DE");
   sort(v.begin(), v.end(), deutsch);
   cout << "nach Sortieren (locale de_DE):\n";
   showSequence(v);
}
```

Im Falle von Multi-Byte-Sequenzen wie bei der UTF-8-Codierung ist `string` nicht geeignet und man muss mit `wstring` arbeiten:

**Listing 24.8:** Sprachlich richtige Sortierung bei UTF-8-Codierung

```
// cppbuch/k24/strings/richtigsortieren.cpp
// Diese Datei ist UTF-8 codiert
#include<algorithm>
#include<iostream>
#include<vector>
#include"printWstringVector.h" // zeigt einen wstring-Vektor an
#include<string>

using namespace std;

int main() {
   locale::global(locale("de_DE.utf-8")); // deDE global setzen
   vector<wstring> v;
   v.push_back(L"Maße");
   v.push_back(L"Masse");
```

```
   v.push_back(L"Ähnlich");
   v.push_back(L"Alphabet");
   v.push_back(L"aal");
   v.push_back(L"ähnlich");
   v.push_back(L"alphabet");
   v.push_back(L"Aal");
   wcout << "vorher:";
   printWstringVector(v);
   sort(v.begin(), v.end());
   wcout << "nach Sortieren ohne Compare-Objekt:\n";
   printWstringVector(v);
   locale deutsch("de_DE");
   sort(v.begin(), v.end(), deutsch);
   wcout << "nach Sortieren (locale de_DE):\n";
   printWstringVector(v);
}
```

### 24.1.5 Umwandlung in Klein- bzw. Großschreibung

Wie im vorherigen Abschnitt gibt es das Problem, dass die C-Standardfunktionen `toupper()` und `tolower()` Umlaute nicht korrekt umwandeln – nämlich dann, wenn die Sprachumgebung nicht richtig eingestellt ist. Für eine ISO-8859-1-Umgebung bieten sich zwei einfache Funktionen an, die mitsamt ihrer Anwendung gezeigt werden:

**Listing 24.9:** Umwandlung von Strings in Klein- bzw. Großschreibung (ISO8859-1-Codierung)

```
// cppbuch/k24/strings/kleingrosseinfachISO8859.cpp
#include<iostream>
#include<cstring>
#include<locale>
#include<string>

void kleinEinfach(std::string& str) {
   for(size_t i = 0; i < str.length(); ++i) {
      str[i] = tolower(str[i]); // benutzt eingestellte Locale
   }
}

void grossEinfach(std::string& str) {
   for(size_t i = 0; i < str.length(); ++i) {
      str[i] = toupper(str[i]); // benutzt eingestellte Locale
   }
}
using namespace std;

int main() {
   locale::global(locale("de_DE")); // de_DE global setzen
   string text("Falsches Üben von Xylophonmusik quält jeden größeren Zwerg.");
   cout << "anfangs:" << text << endl;
   grossEinfach(text);
```

```
  cout << "grossEinfach: " << text << endl;
  kleinEinfach(text);
  cout << "kleinEinfach: " << text << endl;
}
```

In einer UTF-8-Umgebung funktioniert das Programm nicht richtig, erkennbar an der falschen Ausgabe der Umlaute. Die Lösung dieses Problems folgt sofort: Falls sowohl `string` als auch `wstring`-Objekte bearbeitet werden sollen, sind Template-Funktionen sinnvoll. Um nicht Spezialisierungen mit `toupper()`, `tolower()`, `towupper()` `towlower()` schreiben zu müssen, wird auf die Funktionen von Seite 831 zurückgegriffen. Die Funktionen `klein()` und `gross()` werden in einer Template-Datei gekapselt:

**Listing 24.10:** Templates zu String-Umwandlung

```
// Auszug aus cppbuch/k24/strings/localeutils.t
#include<string>
#include<locale>

template<typename charT>
void klein(std::basic_string<charT>& s,
          const std::locale& loc = std::locale()) {
  for(size_t i = 0; i < s.length(); ++i) {
    s[i] = std::use_facet<std::ctype<charT> >(loc).tolower(s[i]);
  }
}

template<typename charT>
void gross(std::basic_string<charT>& s,
          const std::locale& loc = std::locale()) {
  for(size_t i = 0; i < s.length(); ++i) {
    s[i] = std::use_facet<std::ctype<charT> >(loc).toupper(s[i]);
  }
}
```

Wenn kein `locale`-Objekt angegeben wird, ist die global eingestellte Sprachumgebung zuständig. Die Anwendung für 1-Byte-Zeichen ist ähnlich einfach wie oben:

**Listing 24.11:** de_DE-String-Umwandlung (1-Byte-Zeichen)

```
// cppbuch/k24/strings/kleingrossISO8859.cpp
#include<iostream>
#include"localeutils.t"
using namespace std;
int main() {
  locale::global(locale("de_DE")); // de_DE global setzen
  string text("Falsches Üben von Xylophonmusik quält jeden größeren Zwerg.");
  cout << "anfangs:" << text << endl;
  gross(text);
  cout << "gross: " << text << endl;
  klein(text);
  cout << "klein: " << text << endl;
}
```

Im folgenden Beispiel wird eine UTF-8-Locale in Verbindung mit einem `wchar_t`-String (für Multibyte-Zeichen) verwendet:

**Listing 24.12:** wstring-Umwandlung

```
// cppbuch/k24/strings/kleingrossUTF8.cpp
// Diese Datei ist UTF-8 codiert!
#include<iostream>
#include"localeutils.t"
using namespace std;
// ACHTUNG: nur wcout verwenden!
// Bei GNU C++ 4.5 ändert die Benutzung von cout die Einstellungen für wcout
int main() {
   locale::global(locale("de_DE.utf-8")); // deDE global setzen
   wstring text(L"Falsches Üben von Xylophonmusik quält jeden größeren Zwerg.");
   wcout << "anfangs:" << text << endl;
   gross(text);
   wcout << "gross: " << text << endl;
   klein(text);
   wcout << "klein: " << text << endl;
}
```

Mehr über Zeichensätze und -codierung lesen Sie in Abschnitt 31.2.

### 24.1.6 Strings sprachlich richtig vergleichen

Das Problem der Sortierung, die ja einen Vergleich beinhaltet, wird schon in Abschnitt 24.1.4 auf Seite 629 gelöst. Hier bleibt deshalb nur, den sprachlich richtigen Vergleich zweier Strings zu gestalten. Dazu baut man sich einen Funktor, der die `locale`-Einstellung auswertet:

**Listing 24.13:** Funktor zum String-Vergleich

```
// Auszug aus cppbuch/k24/strings/localeutils.t
template<typename charT>
struct Stringvergleich {
   Stringvergleich(const std::locale& lo = std::locale())
      : loc(lo) {
   }
   bool operator()(const std::basic_string<charT>& s1,
                   const std::basic_string<charT>& s2) {
      return std::use_facet<std::collate<charT> >(loc)
         .compare(s1.c_str(), s1.c_str() + s1.length(),
                  s2.c_str(), s2.c_str() + s2.length()) < 0;
   }
   const std::locale loc;
};
```

Das kleine Beispielprogramm zeigt, wie der Funktor zum Vergleich zweier `char`-Strings verwendet wird:

**Listing 24.14:** String-Vergleich

```
// cppbuch/k24/strings/stringvergleich.cpp
#include<iostream>
#include"localeutils.t"
using namespace std;

int main() {
   locale::global(locale("de_DE")); // de_DE global setzen
   Stringvergleich<char> sv;
   string s1("ähnlich");
   string s2("bildschön");
   if(sv(s1, s2)) {
      cout << s1 << " kommt vor " << s2 << endl;
   }
   else {
      cout << s2 << " kommt vor " << s1 << endl;
   }
}
```

Zum Vergleich zweier `wchar_t`-Strings (`wstrings`) kann ein `Stringvergleich<wchar_t>`-Funktor genommen werden, siehe *cppbuch/k24/strings/wstringvergleich.cpp*

### 24.1.7 Von der Groß-/Kleinschreibung unabhängiger Zeichenvergleich

In Abschnitt 24.1.5 wird die Umwandlung von Strings in Klein- bzw. Großschreibung behandelt. Hier geht es nur noch um den Vergleich einzelner Zeichen. Für Nicht-ASCII-Zeichen ist auch hier die Sprachumgebung entscheidend. Um für `char` und `wchar_t`-Zeichen gewappnet zu sein, wird eine Template-Funktion genommen. Das `locale`-Objekt kann übergeben werden; wenn nicht, wird die Voreinstellung angewendet:

**Listing 24.15:** Zeichenvergleich ohne Berücksichtigung der Groß-/Kleinschreibung

```
// Auszug aus cppbuch/k24/strings/localeutils.t
template<typename charT>
bool equalIgnoreCase(charT c1, charT c2,
                     const std::locale& loc = std::locale()) {
   return std::use_facet<std::ctype<charT> >(loc).toupper(c1)
      == std::use_facet<std::ctype<charT> >(loc).toupper(c2);
}
```

Die Benutzung der Funktion zeigt dieses kleine Beispiel:

**Listing 24.16:** equalIgnoreCase() in Aktion

```
// cppbuch/k24/strings/zeichenvergleich.cpp
#include<iostream>
#include"localeutils.t"
using namespace std;

int main() {
   locale::global(locale("de_DE")); // de_DE global setzen
   string s1("ähnlich_x"); // 1-char-Zeichen, hier ISO8859-1 codiert
   string s2("ÄHNLICH_Y");
```

```
  for(size_t i = 0; i < s1.length() && i < s2.length(); ++i) {
       cout << "Die Zeichen Nr. " << i << " ("
            << s1[i]<< ", " << s2[i] << ") werden als ";
    if(!equalIgnoreCase(s1[i], s2[i])) {
       cout << "un";
    }
    cout << "gleich gewertet." << endl;
}
```

Einzelheiten zur Funktion `equalIgnoreCase()` lesen Sie in Abschnitt 31.4.2.

### 24.1.8 Von der Groß-/Kleinschreibung unabhängige Suche

Wenn in einem String unabhängig von der Groß- bzw. Kleinschreibung gesucht werden soll, gibt es zwei verschiedene Herangehensweisen:

- Die beteiligten Strings werden umgewandelt, sodass sie großgeschrieben (bzw. kleingeschrieben) sind.
  *Vorteil:* Einfach. Die Mitgliedsfunktion `find()` der Klasse `string` kann genutzt werden.
  *Nachteil:* Es werden temporäre String-Objekte erzeugt.
- Es werden Algorithmen der Standardbibliothek benutzt oder eigene Funktionen, denen jeweils ein Vergleichsobjekt mitgegeben wird.
  *Vorteil:* Es werden keine temporären String-Objekte erzeugt.
  *Nachteil:* Geringfügig komplizierter.

Wegen der Vermeidung temporärer Objekte wird nur der zweite Fall betrachtet. Es wird ein Funktor `EqualIgnoreCase` benutzt, dessen Name sich nur durch die Großschreibung von der obigen Funktion `equalIgnoreCase()` unterscheidet und der diese Funktion intern benutzt:

**Listing 24.17:** Funktor für Zeichenvergleich und Suchfunktion

```
// Auszug aus cppbuch/k24/strings/localeutils.t
template<typename charT>
struct EqualIgnoreCase {                        // Funktor
  EqualIgnoreCase(const std::locale& lo = std::locale())
    : loc(lo) {
  }
  bool operator()(charT c1, charT c2) {
    return equalIgnoreCase(c1, c2, loc);
  }
  const std::locale loc;
};

// Suchfunktion
template<typename charT>
size_t findeSuchstringIgnoreCase(const std::basic_string<charT>& text,
                                 const std::basic_string<charT>& such,
                              const std::locale& loc = std::locale("de_DE")) {
  auto pos = std::search(text.begin(), text.end(),
                         such.begin(), such.end(),
                         EqualIgnoreCase<charT>(loc)); // Funktor
```

```
   return pos ==
         text.end() ? std::basic_string<charT>::npos : (pos-text.begin());
}
```

`search()` ist ein Algorithmus der Standardbibliothek (siehe Seite 677). Die Anwendung ist einfach:

**Listing 24.18:** Beispiel zur Substring-Suche

```
// cppbuch/k24/strings/sucheIgnorecase.cpp (UTF-8 codiert)
#include<iostream>
#include"localeutils.t"

using namespace std;

int main() {
   locale::global(locale("de_DE.utf-8")); // deDE global setzen
   wstring text(L"Quer über den großen Sylter Deich");
   wstring suchstring(L"Über");
   size_t pos = findeSuchstringIgnoreCase(text, suchstring);
   if(pos != wstring::npos) {
      wcout << suchstring << " ist ab Position " << pos
           << " in " << text
           << L" enthalten (Groß-/Kleinschreibung ignoriert)." << endl;
   }
   else {
      wcout << suchstring << " ist nicht in " << text
            << " enthalten." << endl;
   }
}
```

Bei einer Codierung, die nur ein Byte pro Zeichen benötigt, wie etwa ISO8859-1, muss `wstring` durch `string` und `wcout` durch `cout` ersetzt werden. Das 'L' vor den Anführungszeichen entfällt, wie auch die utf-8-Kennung der locale.

# 24.2 Textverarbeitung

## 24.2.1 Datei durchsuchen

Das folgende Programm durchsucht eine Datei nach einem Begriff, der durch einen regulären Ausdruck definiert wird. Dabei geschieht die Auswertung wie bei dem Unix-Programm *egrep*. Bei Angabe der Option `-i` spielen Klein- und Großschreibung keine Rolle. Die Grundlagen zu den regulären Ausdrücken finden Sie in Kapitel 12.

**Listing 24.19:** Datei durchsuchen

```
// cppbuch/k24/textverarbeitung/regex/finde.cpp
#include <iostream>
```

```
#include <fstream>
#include<boost/regex.hpp>
#include<string>
using namespace std;

int main(int argc, char* argv[]) {
   // nicht nach ECMA-Standard, sondern wie egrep auswerten:
   boost::regex::flag_type flags = boost::regex::egrep;
   string gebrauch("Gebrauch: finde.exe [-i] \"regex\" \"dateiname\"" );
   string option;
   int start = 1;
   // Ist die Option -i gesetzt?
   if(4 == argc) {
      option = argv[1];
      if(option != "-i") {
         cerr << "Falsche Option: " << gebrauch << endl;
         return 1;                          // EXIT
      }
      ++start;
      flags |= boost::regex::icase;  // Klein-/Großschreibung ignorieren
   }
   else if(3 != argc) {
      cout << gebrauch << endl;
      return 2;                          // EXIT
   }
   // Datei durchsuchen
   try {
      // flags transportiert die Einstellungen egrep und icase
      boost::regex gesucht(argv[start], flags);
      ifstream quelle(argv[start+1]);
      size_t zeilennr = 0;
      if(!quelle.good()) {
         throw ios::failure("Dateifehler");
      }
      while(quelle.good()) {
         string zeile;
         getline(quelle, zeile);
         ++zeilennr;
         if(boost::regex_search(zeile, gesucht)) {
            cout << zeilennr << ": " << zeile << endl;
         }
      }
   }
   catch(boost::regex_error& re) {
      cerr << "Regex-Fehler: " << re.what() << endl;
   }
   catch(ios::failure& e) {
      cerr << e.what() << endl;
   }
}
```

Das Programm gibt die Zeilen, in denen der Suchbegriff gefunden wird, mit der Zeilennummer aus. Beispiele:

- `finde.exe "hallo" "text.txt"` findet jedes Auftreten von `hallo` in der Datei.
- `finde.exe -i "hallo" "text.txt"` findet auch `Hallo`, `HALLO`, `hallO` usw.
- `finde.exe "\\([^)]*\\)" "text.txt"` findet alle Zeilen mit öffnender und schließender Klammer. Die Escape-Zeichen sind notwendig, weil sonst die Klammern als Meta-Zeichen für eine Gruppe interpretiert würden. Die runde Klammer innerhalb einer Zeichenklasse ist kein Meta-Zeichen.
- `finde.exe "\\\\" "text.txt"` findet alle Zeilen mit Backslash. Wie im vorherigen Beispiel zu sehen, wird ein Backslash, der als Escape-Zeichen wirken soll, durch `\\` dargestellt. Wenn er selbst gemeint ist, ist er zu verdoppeln – deswegen vier Backslashes.
- `finde.exe "\\bif *\\(" "text.txt"` findet alle Zeilen mit einer if-Anweisung. Dabei verhindert `\b` (Wortgrenze), dass zum Beispiel auch `exif(...` gefunden wird. `*` bedeutet, dass das davor stehende Leerzeichen beliebig oft zwischen `if` und der Klammer vorkommen darf.
- `finde.exe "\\bhttp://[^:/]\\.de" "text.txt"` findet alle Zeilen mit einer auf `.de` endenden HTTP-Adresse (URI bzw. URL). Zur Zeichenklasse: Nach `:` folgt der Port, der deswegen nicht im Host-Namen enthalten sein darf. `/` kann allenfalls nach dem Host-Namen folgen. Die Syntax ist extrem vereinfacht. Die vollständige URI-Syntax finden Sie unter [URI]. Dort wird auch auf den Unterschied zwischen URI und URL eingegangen.

**Hinweis**

Die Anführungszeichen sind nicht immer notwendig. So kann ebenso gut `text.txt` statt `"text.txt"` geschrieben werden. Sie sind dann erforderlich, wenn die Auswertung durch den Kommandointerpreter verhindert werden muss – zum Beispiel bei Dateinamen oder regulären Ausdrücken, die Leerzeichen enthalten.

### 24.2.2 Ersetzungen in einer Datei

Das obige Programm durchsucht eine Datei wie *zeilenweise*, was normalerweise zur Suche genügt. Bei Textersetzungen kann es jedoch vorkommen, dass zeilenübergreifendes Suchen gewünscht wird. Beispiele: Nach jeder Zeile eine Leerzeile oder vor jeder Zeile *** einfügen. Aus diesem Grund wird im nächsten Programm eine Datei als Ganzes eingelesen und dann verarbeitet. Das Ergebnis wird auf der Standardausgabe ausgegeben und kann mit `>` in eine Datei umgeleitet werden.

**Listing 24.20:** Ersetzen in einer Datei

```
// cppbuch/k24/textverarbeitung/regex/ersetze.cpp
#include <iostream>
#include <fstream>
#include<boost/regex.hpp>
#include<string>

std::string ersetzeInDatei(const boost::regex& gesucht,
                           const std::string& ersatz,
                           const char* dateiname) {
```

```
   std::ifstream quelle(dateiname, std::ios::binary | std::ios::in);
   if(!quelle.good()) {
      throw std::ios::failure("Dateifehler");
   }
   std::string alles;
   while(quelle.good()) {
      char c = (char)quelle.get();
      if(!quelle.fail()) {
         alles += c;
      }
   }
   return boost::regex_replace(alles, gesucht, ersatz);
}

using namespace std;

int main(int argc, char* argv[]) {
   boost::regex::flag_type flags = 0; // s.unten icase, falls gefordert
   string gebrauch("Gebrauch: ersetze.exe [-i] \"regex\" \"ersatz\" \"dateiname\"" );
   string option;
   int start = 1;
   // Ist die Option -i gesetzt?
   if(5 == argc) {
      option = argv[1];
      if(option != "-i") {
         cerr << "Falsche Option: " << gebrauch << endl;
         return 1;                          // EXIT
      }
      ++start;
      flags |= boost::regex::icase;  // Klein-/Großschreibung ignorieren
   }
   else if(4 != argc) {
      cout << gebrauch << endl;
      return 2;                          // EXIT
   }

   // Datei durchsuchen
   try {
      boost::regex gesucht(argv[start], flags);
      string ersatz(argv[start+1]);
      string ergebnis = ersetzeInDatei(gesucht, argv[start+1], argv[start+2]);
      cout << ergebnis << endl;
   }
   catch(boost::regex_error& re) {
      cerr << "Regex-Fehler: " << re.what() << endl;
   }
   catch(ios::failure& e) {
      cerr << e.what() << endl;
   }
}
```

Auch hier wird gegebenenfalls die Option -i wirksam. Einige Beispiele:

- `ersetze.exe -i "hallo""hello" text.txt`
  ersetzt »Hallo«, »hallo« und »HALLO« usw. durch »hello«.
- `ersetze.exe "\\n" "\\n\\n" text.txt` fügt nach jeder Zeile eine Leerzeile ein.
- `ersetze.exe "\\n" "***\\n" text.txt` fügt *** am Ende jeder Zeile ein.

Dies sind nur einfache String-Ersetzungen. Es ist aber auch möglich, sich auf Capturing Groups zu beziehen. Um auf das Beispiel am Anfang des Kapitels 12 zurückzukommen:

```
ersetze.exe "Version +1\\.(2|3)" "Version \\1.0" text.txt
```

wandelt den Text : `Nach Version 1.1 kamen Version 1.2 und Version 1.3`

in : `Nach Version 1.1 kamen Version 2.0 und Version 3.0`

um. Das +-Zeichen besagt, dass ein oder mehr Leerzeichen vor der Ziffer 1 stehen dürfen. 2 oder 3 wird jeweiliger Inhalt der Capturing Group. Im Ersatzstring wird \\1 durch den Inhalt der aktuellen ersten (und hier einzigen) Capturing Group ersetzt, gefolgt von einem Punkt und der 0.

### 24.2.3 Code-Formatierer

Die meisten IDEs haben einen Code-Formatierer, der unsauber gesetzte Klammern zurechtrückt. Diese Code-Formatierer sind aber nicht geeignet, um etwa mit einem Skript alle Dateien eines Verzeichnisses zu formatieren. Der folgende Code-Formatierer arbeitet nach dem Filterprinzip: Jedes Zeichen wird gelesen, analysiert und im Allgemeinen sofort wieder ausgegeben. Im Besonderen werden jedoch alle Leerzeichen und Tabulatorzeichen am Zeilenanfang ignoriert und Leerzeichen für die Einrückung eingefügt. Die geschweiften Klammern bestimmen die Einrückung. Die Zeilenstruktur bleibt erhalten, d.h. eine Zeile erhält eine neue Anfangsposition, wird aber sonst nicht verändert.

**Listing 24.21:** Code-Formatierer

```
// cppbuch/k24/textverarbeitung/codeformatter.cpp
#include<iostream>
#include<fstream>
#include<cstring>
using namespace std;

int main(int argc, char* argv[]) {
   if(argc != 3) {
      cerr << "Gebrauch: codeformatter eingabe.cpp ausgabe.cpp" << endl;
      return 1;
   }
   const char* const EINRUECKUNG = " ";
   const size_t ANZAHLCHARS = strlen(EINRUECKUNG);
   bool warEOL = true; // War das letzte Zeichen ein Zeilenende?
   ifstream is(argv[1]);
   ofstream os(argv[2]);
   size_t level = 0;
   char c;
   while(is.good()) {
      do { // nächstes Zeichen lesen, ggf. Leerzeichen schlucken
         is.get(c);
```

```
    } while(is.good() && warEOL && (c == ' ' || c == '\t'));
    if(is.good()) {
       if (c == '}') {
          --level;
       }
       if (warEOL) {
          size_t leveltmp = level+1;
          while (--leveltmp > 0)
             os.write(EINRUECKUNG, ANZAHLCHARS);
       }
       os.put(c);
       warEOL = (c == '\n');
       if (c == '{') {
          ++level;
       }
    }
  }
}
```

Dieses Programm hat noch eine kleine Schwäche: Geschweifte Klammern in Strings, Zeichenliteralen oder in Kommentaren werden nicht ignoriert.

**Übung**

24.1 Erweitern Sie das Programm, um die erwähnte Schwäche zu beheben. Denken Sie daran, dass ein Anführungszeichen auch maskiert sein kann, mit einem Backslash oder in einem Zeichenliteral wie '"'.

### 24.2.4 Lines of Code (LOC) ermitteln

Die Anzahl der Code-Zeilen eines Programms nach Abzug aller Kommentare und Leerzeilen (englisch *Lines of Code*), oft mit LOC abgekürzt, ist ein in der Softwaretechnik verwendetes Maß für die Arbeitsleistung, die in einem Programm steckt. Dieses Maß ist eine nur grobe Näherung, unter anderem, weil von dem Schwierigkeitsgrad eines Algorithmus abstrahiert wird. Dennoch wird dieses Maß häufig benutzt, zum Beispiel für statistische Zwecke (Anzahl Fehler pro 1000 LOC). Das folgende Programm ermittelt die Anzahl der Code-Zeilen mithilfe regulärer Ausdrücke, indem alles, was nicht zur Anzahl beiträgt, entfernt wird. Anschließend wird die Zahl der übrig gebliebenen Zeilen ermittelt. Die Kommentare im Programm erläutern die einzelnen Schritte. Die Grundlagen zu den regulären Ausdrücken finden Sie in Kapitel 12.

**Listing 24.22:** Lines of code zählen

```
// cppbuch/k24/textverarbeitung/regex/loc.cpp
#include <algorithm>
#include <iostream>
#include <fstream>
#include<boost/regex.hpp>
#include<string>

int zaehleLOC(const char* dateiname) {
```

```
   std::ifstream quelle(dateiname, std::ios::binary | std::ios::in);
   if(!quelle.good()) {
      throw std::ios::failure("Dateifehler");
   }
   std::string alles;
   while(quelle.good()) {
      char c = (char)quelle.get();
      if(!quelle.fail()) {
         alles += c;
      }
   }

   // Zeilenendekennung vereinheitlichen
   alles = boost::regex_replace(alles, boost::regex("(\r\n)"), "\n");

   // Leer- und Tabulatorzeichen entfernen
   alles = boost::regex_replace(alles, boost::regex("( |\t)"), "");

   // Escape-Zeichen löschen
   alles = boost::regex_replace(alles, boost::regex("\\\\."), "");

   // Alle Strings durch "" nicht-greedy ersetzen
   alles = boost::regex_replace(alles, boost::regex("\".*?\""), "\"\"");

   // alle /* enthaltenen // Kommentare ersetzen. Leerzeichen
   // muss sein, um nicht auf // /** -> //* reinzufallen
   alles = boost::regex_replace(alles, boost::regex("//[^\n]*/\\*"), "// ");

   // mit /* beginnende Kommentare löschen
   alles = boost::regex_replace(alles, boost::regex("(?s)/\\*.*?\\*/"), "");

   // alle // Kommentare löschen
   alles = boost::regex_replace(alles, boost::regex("//[^\n]*"), "");

   // Führende Whitespaces entfernen.
   // Damit werden auch Leerzeilen entfernt (\n am Zeilenanfang)
   alles = boost::regex_replace(alles, boost::regex("^\\s+"), "");

   // Anzahl der Zeilen zurückgeben
   return 1 + std::count(alles.begin(), alles.end(), '\n');
}

using namespace std;

int main(int argc, char* argv[]) {
   string gebrauch("Gebrauch: loc.exe dateiname" );
   if(2 != argc) {
      cout << gebrauch << endl;
      return 1;                              // EXIT
```

```
  }
  try {
    cout << "Die Datei " << argv[1] << " hat "
         << zaehleLOC(argv[1]) << " Lines of Code (LOC)." << endl;
  }
  catch(boost::regex_error& re) {
    cerr << "Regex-Fehler: " << re.what() << endl;
  }
  catch(ios::failure& e) {
    cerr << e.what() << endl;
  }
}
```

### 24.2.5 Zeilen, Wörter und Zeichen einer Datei zählen

Diese Aufgabe übernimmt das Unix-Programm `wc` (word count). Das Kommando `wc datei` gibt die Anzahl der Zeilen, Wörter und Zeichen in dieser Reihenfolge aus. Ein vergleichbares C++-Programm ist die Lösung zur Aufgabe 2.4 auf Seite 99, wobei Umlaute dort keine Berücksichtigung finden. Dies lässt sich leicht ändern:

- `locale::global(locale("de_DE"));` einstellen.
- Die Abfrage `if (c >= 'A'&& c <= 'Z'|| c >= 'a'&& c <= 'z')` durch `if (isalpha(c))` ersetzen (Header `<cstring>`). `isalpha(c)` berücksichtigt die globale `locale`-Einstellung (siehe auch *cppbuch/k24/textverarbeitung/wcErsatz.cpp*).

### 24.2.6 CSV-Datei lesen

Tabellenkalkulationsprogramme können Tabellen im CSV-Format exportieren. CSV bedeutet »comma separated values« und ist ein Textformat, also nicht binär. Im folgenden Beispiel wird eine im CSV-Format vorliegende Tabelle mit ganzen Zahlen eingelesen und in einem `vector<vector<int> >` abgelegt, auf den wie auf ein normales zweidimensionales Array zugegriffen werden kann.

**Listing 24.23:** CSV-Datei lesen

```
// cppbuch/k24/textverarbeitung/csvauswerten.cpp
#include<string>
#include<cstring>
#include<vector>
#include<cstdlib> // für exit( )
#include<fstream>
#include<iostream>

std::vector<int> splitInt(char* buf, const char* trennzeichen) { // siehe Text
  std::vector<int> vec;
  char* str = strtok(buf, trennzeichen);
  while(str) {
    vec.push_back(atoi(str));
    str = strtok(NULL, trennzeichen);
  }
  return vec;
}
```

```
using namespace std;

int main( ) {
   // Definieren der Eingangsdatei
   ifstream quelle;          // Datentyp für Eingabestrom
   string csvDateiname;
   cout << "Csv-Datei? ";
   cin >> csvDateiname;
   quelle.open(csvDateiname.c_str(), ios::in);
   if (!quelle) { // Fehlerabfrage
      cerr << csvDateiname << " kann nicht geöffnet werden!\n";
      exit(-1);
   }
   const size_t N = 1000;
   char buf[N];        // Muss groß genug für eine Zeile sein
   vector<vector<int> > tabelle; // Vektor von Tabellenzeilen
   while(quelle.good()) {
      quelle.getline(buf, N);
      if(!quelle.fail()) {
         tabelle.push_back(splitInt(buf, ","));
      }
   }
   quelle.close();
   // Tabelle ausgeben
   for(size_t i=0; i < tabelle.size(); ++i) {
      for(size_t j=0; j < tabelle[i].size(); ++j) {
         cout << tabelle[i][j] << " ";
      }
      cout << endl;
   }
}
```

Die Zeilen werden mit `getline()` eingelesen. Weil die Datei nicht mit `ios::binary` eröffnet wird, werden sowohl die Unix- als auch die Windows-Zeilenendekennung akzeptiert.

Jede eingelesene Zeile wird von der Funktion `splitInt(char* buf, const char* trennzeichen)` analysiert. `trennzeichen` ist dabei eine Folge von Trennzeichen, die in diesem Fall nur aus einem Komma besteht. Die C-Funktion `strtok()` zerlegt die Zeichenkette in Token, die durch Kommata getrennt sind. Der Parameter ist `char*` anstatt `const char*`, weil `strtok()` jedes Trennzeichen durch ein Null-Byte ersetzt. Die Variable `str` verweist jeweils auf den Beginn des nächsten, mit einem Null-Byte abgeschlossenen Tokens. Die C-Funktion `atoi()` (für ASCII to integer) gibt das Token als `int`-Zahl zurück, die an den Vektor angehängt wird. `splitInt()` gibt also jede CSV-Zeile als `vector<int>` zurück, der an die Variable `tabelle` angehängt wird. Wie am Ende des Beispiels zu sehen, kann auf `tabelle` wie auf ein Array zugegriffen werden (Grundlagen dazu siehe Abschnitt 9.8).

Vorteil dieser Vorgehensweise ist, dass die Datei nur einmal gelesen werden muss. Wollte man eine Tabelle mit festen Ausmaßen definieren, müsste die Datei einmal gelesen werden, um die Anzahl der benötigten Zeilen festzustellen, und ein zweites Mal, um die Werte auszulesen – oder man müsste die gesamte Datei als String-Vektor zwischenspeichern.

### 24.2.7 Kreuzreferenzliste

Eine Kreuzreferenzliste ist eine Liste, die die Worte oder Bezeichner eines Textes alphabetisch mit den Positionen des Vorkommens, hier den Zeilennummern, enthält. Hier ist der Anfang der Kreuzreferenzliste zum Programm *crossrefISO8859_1.cpp*:

*_ : 54 63 65*
*a : 22 23 25*
*abgespeichert : 11*
*an : 68*
*Anfang : 53*

*argc : 30 31 44*
*argv : 30 32 34 35 38 40 45*
*auch : 14*
*auf : 14 19*
*auto : 74*

Das Programm wurde mit der Locale »de_DE« aufgerufen. Andernfalls, also bei der Standard-Locale C, gäbe es nur eine Auswertung nach dem ASCII, sodass »berücksichtigt« in zwei Bezeichner »ber« und »cksichtigt« zerfiele. Die passende Datenstruktur ist ein Map-Container. Die Wertepaare bestehen aus dem Bezeichner vom Typ `string` als Schlüssel und aus einem Set mit den Zeilennummern. Aufgrund der sortierten Ablage ist kein besonderer Sortiervorgang notwendig. Weil das Programm zeilenweise liest, wäre statt eines Sets eine Liste denkbar. Ein Set hat aber den Vorteil, dass ein mehrfaches Vorkommen eines Worts in einer Zeile nur einfach gezählt wird.

**Listing 24.24:** Programm für eine Kreuzreferenzliste

```
// cppbuch/k24/textverarbeitung/crossrefISO8859_1.cpp
#include<cctype>
#include<fstream>
#include<iostream>
#include<set>
#include<locale>
#include<map>
#include<string>
#include<showSequence.h>
// Test für Locale de_DE: Mucke Mücke Süßholz Süsse
/* Diese Datei ist im ISO 8859-1 Format abgespeichert, so dass
   Umlaute nur einem Byte entsprechen.
   Konvertierung nach UTF-8 bewirkt, dass der obige Test fehlschlägt!
   Zur korrekten Dartsellung sollte das Terminal auch auf ISO 8859-1
   eingestellt sein.
*/
// Um eine unterschiedliche Sortierung für Groß- und Kleinschreibung
// zu vermeiden, wird die Klasse Vergleich eingesetzt, die die zu
// vergleichenden Strings vorher auf Kleinschreibung normiert.

struct Vergleich {
   bool operator()(std::string a, std::string b) const { // per Wert
      for(size_t i=0; i< a.length(); ++i) a[i]=std::tolower(a[i]);
      for(size_t i=0; i< b.length(); ++i) b[i]=std::tolower(b[i]);
      return a < b; // Stringvergleich berücksichtigt Locale
```

```
  }
};

using namespace std;
int main(int argc, char* argv[]) {
  if(argc == 1) {
    cout << "Gebrauch: " << argv[0] << " Dateiname [locale]\n"
      " Beispiele (vorgegebene Locale: C)\n"
        << argv[0] << " crossrefISO8859_1.cpp de_DE\n"
        << argv[0] << " crossrefISO8859_1.cpp" << endl;
    return 0;
  }
  ifstream quelle(argv[1]);
  if(!quelle.good()) {
    cout << argv[1] << " nicht gefunden!\n";
    return 1;
  }

  if(argc == 3) {
    locale::global(locale(argv[2])); // ggf. global setzen
  }

  map<string, set<int>, Vergleich> bezeichnerZeilenMap;

  int zeilennr = 1;
  while(quelle.good()) {
    char c = '\0';
    // Anfang des Bezeichners finden
    while(quelle.good() && !(isalpha(c) || '_' == c)) {
      quelle.get(c);
      if(c == '\n') {
        ++zeilennr;
      }
    }
    if(quelle.good()) {
      string bezeichner(1,c);
      // Rest des Bezeichners einsammeln
      while(quelle.good() && (isalnum(c) || '_' == c)) {
        quelle.get(c);
        if(isalnum(c) || '_' == c)
          bezeichner += c;
      }
      quelle.putback(c); // zurück an den Eingabestrom
      if(c) {  // Bezeichner gefunden?
        bezeichnerZeilenMap[bezeichner].insert(zeilennr); // Eintrag
      }
    }
  }
  auto iter = bezeichnerZeilenMap.begin();
  while(iter != bezeichnerZeilenMap.end()) {
    cout << (*iter).first;        // Bezeichner
    cout.width(20 - (*iter).first.length()); // Position bis : einstellen
    cout << ": ";
```

```
      showSequence((*iter++).second); // Zeilennummern
   }
}
```

Das Eintragen der Zeilennummer in die Sets nutzt aus, dass `BezeichnerZeilenMap::operator[]()` eine Referenz auf den Eintrag zurückgibt, auch wenn dieser erst angelegt werden muss, weil der Schlüssel noch nicht existiert. Der Eintrag zu dem Schlüssel `bezeichner` ist ein Set, sodass von vornherein die richtige Reihenfolge sichergestellt ist. Die Ausgabe der Kreuzreferenzliste profitiert von der sortierten Speicherung. Das Element `first` eines Wertepaares ist der Bezeichner (Schlüssel), das Element `second` ist der Set, der mit dem bekannten Template `showSequence` ausgegeben wird.

Der Operator `operator()(string a, string b)` zum Vergleich zweier Strings unabhängig von Groß- und Kleinschreibung mag nicht besonders performant erscheinen, weil eine Kopie der Argumente notwendig und jede Kopie implizit mit einer `new`-Operation verbunden ist. Weil jedoch die Größe der Sets relativ klein ist (proportional zur Anzahl der Zeilen, in denen der Bezeichner vorkommt), hält sich die Anzahl der Aufrufe in Grenzen. Auf die Kopie der Strings zu verzichten, ist nur dann sinnvoll, wenn der Vergleich ausschließlich auf dem ASCII beruhen kann, etwa:

```
// nur für Locale C geeignet!
bool operator()(const std::string& x, const std::string& x) const {
   size_t kleinereLaenge = std::min(x.length(), y.length());
   for(size_t i=0; i < kleinereLaenge; ++i) {
      char cx = std::tolower(x[i]);
      char cy = std::tolower(y[i]);
      if(cx != cy) {
         return cx < cy; // Zeichenvergleich berücksichtigt nicht Locale
      }
   }
   // Falls alle Zeichen gleich sind, ist der kürzere String kleiner
   return x.length() < y.length();
}
```

Für andere Locales ist dieses Verfahren unbrauchbar, weil bei einem sprachlich richtigen Vergleich mehr als ein Zeichen beteiligt sein kann. Beispiel: Nach deutschen Sortierregeln kommt »Masse« vor »Maße«.

## 24.3 Operationen auf Folgen

Diese Algorithmen beschreiben allgemeine numerische Operationen auf Containern. Im Allgemeinen werden die Container Zahlen enthalten. Aufgrund der Formulierung der Algorithmen als Template ist dies jedoch nicht zwingend; es müssen nur die benötigten Operationen vorhanden sein. So steht das +-Zeichen bei Zahlen für die Addition, bei Strings für das Verketten. Der in Abschnitt 24.3.5 verwendete Algorithmus zur Summenbildung wird daher bei einem Container mit Strings als Ergebnis die Verkettung aller Container-Elemente liefern.

### 24.3.1 Container anzeigen

In diesem Buch wird zur Abkürzung gelegentlich die Hilfsfunktion `showSequence()` verwendet, um einen Container auf dem Bildschirm anzuzeigen. Sie liegt im Verzeichnis *cppbuch/include* der Beispiele und ist wie folgt definiert:

**Listing 24.25:** Hilfsfunktion zur Anzeige von Containern

```
// cppbuch/include/showSequence.h
#ifndef SHOWSEQ_H
#define SHOWSEQ_H
#include<iostream>

template<class Container>
void showSequence(const Container& s,
                  const char* abschluss = "\n",
                  const char* trennzeichen = " ") {
   auto iter = s.begin();
   while(iter != s.end()) {
      std::cout << *iter++ << trennzeichen;
   }
   std::cout << abschluss;
}
#endif
```

### 24.3.2 Folge mit gleichen Werten initialisieren

Wenn eine Sequenz ganz oder teilweise mit immer gleichen Werten, nämlich Kopien von `value`, vorbesetzt werden soll, eignen sich die Algorithmen `fill()` oder `fill_n()`:

```
template<class ForwardIterator, class T>
void fill(ForwardIterator first, ForwardIterator last, const T& value);
```

```
// Beispiel: alle Werte eines Vektors v mit 0.45 besetzen
fill(v.begin(), v.end(), 0.45);
```

```
template<class OutputIterator, class Size, class T>
OutputIterator fill_n(OutputIterator first, Size n, const T& value);
```

Im Unterschied zu `fill()` erwartet `fill_n()` die Angabe `n`, wie viele Elemente der Sequenz, auf die `first` verweist, mit `value` vorbesetzt werden sollen. Zurückgegeben wird ein Iterator auf das Ende des modifizierten Bereichs.

```
// Auszug aus cppbuch/k24/vermischtes/fill.cpp
// Beispiel: die erste Hälfte eines Vektors v mit 0.45 besetzen
fill_n(v.begin(), v.size()/2, 0.45);
```

### 24.3.3 Folge mit Werten eines Generators initialisieren

Ein Generator im Algorithmus `generate()` ist ein Funktionsobjekt oder eine Funktion, die ohne Parameter aufgerufen und deren Ergebnis den Elementen der Sequenz der Reihe nach zugewiesen wird. Wie bei `fill()` gibt es eine Variante, die ein Iteratorpaar erwartet, und eine Variante, die den Anfangsiterator und eine Stückzahl benötigt:

```
template<class ForwardIterator, class Generator>
void generate(ForwardIterator first, ForwardIterator last, Generator gen);
```

```
template<class OutputIterator, class Size, class Generator>
OutputIterator generate_n(OutputIterator first, Size n, Generator gen);
```

Das Beispiel zeigt die beide Varianten:

- Im ersten Teil wird als Generator eine Funktion genommen, die Zweierpotenzen erzeugt.
- Im zweiten Teil wird die erste Hälfte des Vektors mit zufälligen Zahlen versehen. Als Generator dient der Zufallszahlengenerator von Seite 712.

**Listing 24.26:** Zahlenfolgen erzeugen

```
// cppbuch/k24/vermischtes/generate.cpp
#include<algorithm>
#include<vector>
#include<showSequence.h>
#include<Random.h>
using namespace std;

int zweierpotenz() {
   static int wert = 1; // mit 1 anfangen
   return wert <<= 1; // Wert verdoppeln
}

int main() {
    vector<int> v(10);
    generate(v.begin(), v.end(), zweierpotenz);
    showSequence(v); // 2 4 8 16 32 64 128 256 512 1024

    Random zufall(10000);
    generate_n(v.begin(), v.size()/2, zufall);
    showSequence(v);
}
```

### 24.3.4 Folge mit fortlaufenden Werten initialisieren

Die Funktion `iota()` (Header `<numeric>`) füllt ein Intervall mit fortlaufenden Werten. Iota heißt der neunte Buchstabe des griechischen Alphabets ($\iota$). Das entsprechende deutsche Wort Jota bedeutet etwa sehr kleine Menge oder das Geringste. Der Name wurde jedoch nicht deswegen, sondern in Anlehnung an den $\iota$-Operator der Programmiersprache APL gewählt. Die APL-Anweisung $\iota\ n$ liefert als Indexgenerator einen Vektor mit einer ansteigenden Folge der Zahlen 1 bis $n$. Im Beispiel des nächsten Abschnitts füllt `iota()` einen Vektor. Die Funktion selbst ist recht einfach, wie an der Definition zu sehen ist:

**Listing 24.27:** Algorithmus iota

```
template<class ForwardIterator, class T>
void iota(ForwardIterator first, ForwardIterator last, T value) {
  while (first != last)
     *first++ = value++;
}
```

### 24.3.5 Summe und Produkt

Der Algorithmus `accumulate()` (Header `<numeric>`) wendet auf alle Werte `*i` eines Iterators `i` von `first` bis `last` den Plus-Operator an. Es gibt eine interne Variable `acc`, die mit `init` initialisiert wird. Anschließend wird für jeden Iterator die Operation `acc += *i` ausgeführt. Falls statt dessen eine andere Operation treten soll, existiert eine überladenen Variante, der die Operation als letzter Parameter übergeben wird. Die jeweils ausgeführte Operation ist dann `acc = binOp(acc, *i)`. Die Prototypen sind:

```
template<class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last, T init);

template<class InputIterator, class T, class binaryOperation>
T accumulate(InputIterator first, InputIterator last, T init,
             binaryOperation binOp);
```

Bei numerischen Elementen und ohne Angabe einer Operation berechnet `accumulate()` für einen Vektor `v` mit den Elementen $v_i$ (i = 0 ... `v.size()-1`) die Summe `init` + $\sum_i v_i$. Im Beispiel wird als Startwert für `init` der Wert 0 angenommen. Im zweiten Teil wird der Operator `multiplies` (siehe Seite 753) eingesetzt. Das Ergebnis ist `init` · $\prod_i v_i$. Weil der Vektor im Beispiel mit der Folge der natürlichen Zahlen initialisiert wird und der Startwert gleich 1 ist, ist das Produkt gleich der Fakultät von 10.

Entsprechend der Bedeutung des +-Operators als Verkettungsoperation bei Strings liefert `accumulate()`, auf einen Container mit Strings angewendet, deren Verkettung, wie am Ende des Beispiels gezeigt wird – auch wenn das nicht gerade die übliche Anwendung ist. Wie man einen eigenen Operator für `accumulate()` schreibt, zeigt der folgende Abschnitt »Mittelwert und Standardabweichung«.

**Listing 24.28:** Summen- und Produktbildung

```
// cppbuch/k24/folgen/accumulate.cpp
#include<vector>
#include<numeric>
#include<iostream>
#include<string>
using namespace std;

int main() {
   vector<int> v(10);
   iota(v.begin(), v.end(), 1);
   cout << "Summe = " << accumulate(v.begin(), v.end(), 0) // 55
        << endl;
   cout << "Produkt = " << accumulate(v.begin(), v.end(), 1L,
                                      multiplies<long>())        // 3628800
        << endl;
   // accumulate() mit string:
   vector<string> vstr(26);
   iota(vstr.begin(), vstr.end(), 'A'); // Vektor mit Buchstaben füllen
   cout << accumulate(vstr.begin(), vstr.end(), string("Alphabet: "))
        << endl;
}
```

### 24.3.6 Mittelwert und Standardabweichung

Wie oben beschrieben, wird in `accumulate()` für jeden Iterator `i` intern die Anweisung `acc = binOp(acc, *i)` ausgeführt. Der Operator verknüpft `acc` mit dem Wert, auf den der Iterator zeigt, und gibt den neuen Wert von `acc` zurück. Im Beispiel wird dieses Wissen genutzt, um einen Operator als Funktor zur Berechnung des Abweichungsquadrats zu formulieren:

**Listing 24.29:** Mittelwert und Standardabweichung berechnen

```
// cppbuch/k24/folgen/statistik.cpp
#include<vector>
#include<cmath>
#include<numeric>
#include<iostream>
#include<Random.h>

// Funktor zur Berechnung des Quadrats der Differenz zu einem Wert
template<class T>
class Abweichungsquadrat {
public:
   Abweichungsquadrat(T m)
      : mittel(m) {
   }
   T operator()(const T& acc, const T& iterWert) const {
      const T d = iterWert - mittel;
      return acc + d*d;
   }
private:
   T mittel;
};
using namespace std;

int main() {
   vector<double> v(6);
   iota(v.begin(), v.end(), 1);   // Vektor füllen
   double summe = accumulate(v.begin(), v.end(), 0.0); // 0.0 (double), nicht: 0
         // weil der Typ den Typ des Ergebnisses bestimmt (sonst Genauigkeitsverlust).
   cout << "Summe         : " << summe << endl;
   double mittelwert = summe/v.size();
   cout << "Mittelwert    : " << mittelwert << endl;
   Abweichungsquadrat<double> aq(mittelwert);
   double varianz = accumulate(v.begin(), v.end(), 0.0, aq)/v.size();
   cout << "Varianz       : " << varianz << endl;
   cout << "Standardabweichung: " << sqrt(varianz) << endl;
}
```

### 24.3.7 Skalarprodukt

Der Algorithmus `inner_product()` (Header `<numeric>`) addiert das Skalarprodukt zweier Container $u$ und $v$, die meistens Vektoren sein werden, auf den Anfangswert `init`:

Ergebnis = `init` + $\sum_i v_i \cdot u_i$

Anstelle der Addition und Multiplikation können auch andere Operationen gewählt werden. Die Prototypen sind:

```
template<class InputIterator1, class InputIterator2, class T>
T inner_product(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, T init);
```

```
template<class InputIterator1, class InputIterator2, class T,
         class binaryOperation1, class binaryOperation2>
T inner_product(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, T init,
                binaryOperation1 binOp1, binaryOperation2 binOp2);
```

In einem euklidischen n-dimensionalen Raum $R^n$ ist die Länge eines Vektors durch die Wurzel aus dem Skalarprodukt des Vektors mit sich selbst definiert. Das Beispiel berechnet zuerst die Länge eines Vektors (1, 1, 1, 1) im $R^4$. Der Wert für `init` muss 0 sein. Im zweiten Teil des Beispiels wird die Entfernung zwischen zwei Punkten, die durch zwei verschiedene Vektoren repräsentiert werden, berechnet.

**Listing 24.30:** Länge und Entfernung mit inner_product()

```
// cppbuch/k24/folgen/innerproduct.cpp
#include<numeric>
#include<vector>
#include<cmath>
#include<iostream>

// Funktor zur Berechnung des Quadrats einer Differenz
template<class T>
struct difference_square {
   T operator()(const T& x, const T& y) {
      const T d = x - y;
      return d*d;
   }
};
using namespace std;

int main() {
   const int DIMENSION = 4;
   vector<int> v(DIMENSION, 1);

   cout << "Länge des Vektors v = "
        << sqrt( (double)inner_product(v.begin(), v.end(), v.begin(), 0))
        << endl;

   // Um die Anwendung anderer mathematischer Operatoren zu zeigen,
   // wird im Folgenden die Entfernung zwischen zwei Punkten berechnet.

   // 2 Punkte p1 und p2
   vector<double> p1(DIMENSION,1.0); // Einheitsvektor
   vector<double> p2(DIMENSION);
   iota(p2.begin(), p2.end(), 1.0); // beliebiger Vektor
```

```
   cout << "Entfernung zwischen p1 und p2 = "
        << sqrt( inner_product(p1.begin(), p1.end(),
                               p2.begin(), 0.0,
                               plus<double>(),
                               difference_square<double>()))
        << endl;
}
```

### 24.3.8 Folge der Teilsummen oder -produkte

Die Partialsummenbildung (Header `<numeric>`) funktioniert ähnlich wie `accumulate()`, nur dass das Ergebnis eines jeden Schritts in einem Ergebniscontainer abgelegt wird, der durch den Iterator `result` gegeben ist, und dass es keinen `init`-Wert gibt. Die Prototypen sind:

```
template<class InputIterator, class OutputIterator>
OutputIterator partial_sum(InputIterator first, InputIterator last,
                           OutputIterator result);
```

```
template<class InputIterator, class OutputIterator,
         class binaryOperation>
OutputIterator partial_sum(InputIterator first, InputIterator last,
                           OutputIterator result,binaryOperation binOp);
```

Das Beispiel zeigt beide Varianten. Die jeweils letzte Zahl einer Folge korrespondiert mit dem Ergebnis von `accumulate()` aus dem obigen Beispiel.

**Listing 24.31:** Folge der Teilsummen

```
// cppbuch/k24/folgen/partialsum.cpp
#include<numeric>
#include<vector>
#include<showSequence.h>
using namespace std;

int main() {
   vector<long> v(10), partialsummen(10);
   iota(v.begin(), v.end(), 1); // natürliche Zahlen 1 bis 10
   partial_sum(v.begin(), v.end(), partialsummen.begin());
   cout << "Partialsummen = ";
   showSequence(partialsummen);    // 1 3 6 10 15 21 28 36 45 55
   cout << "Folge von Fakultäten = ";
   partial_sum(v.begin(), v.end(), v.begin(), multiplies<long>());
   showSequence(v); // 1 2 6 24 120 720 5040 40320 362880 3628800
}
```

### 24.3.9 Folge der Differenzen

Der Algorithmus `adjacent_difference()` (Header `<numeric>`) berechnet die Differenz zweier aufeinanderfolgender Elemente eines Containers $v$ und schreibt das Ergebnis in einen Ergebniscontainer $e$. Auf diesen Ergebniscontainer verweist der Iterator `result`. Da es genau einen Differenzwert weniger als Elemente gibt, bleibt das erste Element erhalten. Wenn das erste Element den Index 0 trägt, gilt also:

$e_0 = v_0$

$e_i = v_i - v_{i-1}; \quad i > 0$

Außer der Differenzbildung sind andere Operationen möglich. Die Prototypen sind:

```
template<class InputIterator, class OutputIterator>
OutputIterator adjacent_difference(InputIterator first, InputIterator last,
                                   OutputIterator result);
```

```
template<class InputIterator, class OutputIterator,
        class binaryOperation>
OutputIterator adjacent_difference(InputIterator first, InputIterator last,
                                   OutputIterator result,
                                   binaryOperation binOp);
```

Das Beispiel zeigt beide Varianten. In der ersten werden Differenzwerte berechnet, in der zweiten eine Folge von Fibonacci-Zahlen. Leonardo von Pisa, genannt Fibonacci, war ein italienischer Mathematiker und lebte ca. 1180–1240.

**Listing 24.32:** Folge der Differenzen und Fibonacci-Zahlen

```
// cppbuch/k24/folgen/adjacent_difference.cpp
#include<numeric>
#include<vector>
#include<showSequence.h>
using namespace std;

int main() {
   vector<long> v(10), adjacDiff(10);
   iota(v.begin(), v.end(), 0);        // 0 1 2 3 4 5 6 7 8 9
   cout << "Differenzen    = ";
   adjacent_difference(v.begin(), v.end(),  // Quelle
                       adjacDiff.begin());     // Ziel
   showSequence(adjacDiff);            // 0 1 1 1 1 1 1 1 1 1
   // Zweite Variante: Statt der (vorgegebenen) Differenz wird
   // ein anderer Operator, hier plus<int>() übergeben.
   // Im Beispiel werden damit Fibonacci-Zahlen erzeugt.
   vector<int> fib(16);
   fib[0] = 1;                         // Anfangswert
   // Ein Startwert genügt hier, weil der erste Wert
   // an Position 1 eingetragen wird (Formel e_i = v_i - v_i-1 oben)
   // und sich damit der zweite Wert von selbst ergibt
   // (beachte den um 1 verschobenen result-Iterator in der Parameterliste).
   cout << "Fibonacci-Zahlen = ";
   adjacent_difference(fib.begin(), fib.end()-1,  // Quelle
                       fib.begin()+1,             // Ziel
                       plus<int>());              // Funktor
   showSequence(fib);     // 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
}
```

Wenn anstatt der vorgegebenen Differenz die Summe der beiden Vorgänger genommen wird, füllt sich der Ergebniscontainer mit einer Folge der Fibonacci-Zahlen. Fibonacci fragte sich, wie viele Kaninchen-Pärchen es wohl nach *n* Jahren gibt, wenn jedes Pärchen ab dem zweiten Jahr pro Jahr ein weiteres Pärchen erzeugt. Dass Kaninchen irgendwann

sterben, wurde bei der Fragestellung ignoriert. Die Antwort auf diese Frage ist, dass die Anzahl der Kaninchen im Jahre $n$ gleich der Summe der Jahre $n-1$ und $n-2$ ist. Die Fibonacci-Zahlen spielen in der Informatik eine Rolle ([CLR]). Man beachte, dass bei der Erzeugung der Folge der Iterator `result` zu Beginn gleich `fib.begin()+1` sein muss.

### 24.3.10 Minimum und Maximum

Die Templates `min_element()` und `max_element()` geben jeweils einen Iterator auf das kleinste (bzw. das größte) Element in einem Intervall `[first, last)` zurück. Bei Gleichheit der Iteratoren wird der erste zurückgegeben. Die Komplexität ist linear. Die Prototypen sind:

```
template<class ForwardIterator>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last,
                        Compare comp);

template<class ForwardIterator>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last,
                        Compare comp);
```

`minmax_element()` gibt dementsprechend ein Paar von Iteratoren auf das kleinste und auf das größte Element zurück.

```
template<class ForwardIterator>
pair<ForwardIterator, ForwardIterator>
minmax_element(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
pair<ForwardIterator, ForwardIterator>
minmax_element(ForwardIterator first, ForwardIterator last, Compare comp);
```

Das folgende Beispiel enthält einen Funktor (Funktionsobjekt) `Absolutbetrag`, um eine Variante mit dem Vergleichsobjekt zu zeigen.

**Listing 24.33:** Minimum und Maximum finden

```
// cppbuch/k24/folgen/minmax.cpp
#include<algorithm>
#include<vector>
#include<numeric>
#include<utility> // pair
#include<showSequence.h>
using namespace std;

struct Absolutbetrag {
  bool operator()(int x, int y) {
    return abs(x) < abs(y);
```

```
  }
};

int main() {
   vector<long> v(10);
   iota(v.begin(), v.end(), -5); // -5 ... +4
   showSequence(v);
    cout << "Minimum: " << *min_element(v.begin(), v.end()) << endl; // -5
    cout << "Maximum: " << *max_element(v.begin(), v.end()) << endl; // 4
    cout << "Minimum des Absolutbetrags: "
         << *min_element(v.begin(), v.end(), Absolutbetrag()) << endl; // 0

    cout << "Minimum und Maximum mit nur einem Funktionsaufruf: ";
    typedef vector<long>::iterator iter; // zur Abkürzung
    pair<iter, iter> p = minmax_element(v.begin(), v.end());
    cout << *p.first << " " << *p.second << endl;                // -5 4
}
```

Mehr über `pair` lesen Sie in Abschnitt 27.3.

Anstelle eines Funktors ist auch die Übergabe einer Funktion möglich:

```
bool abskleiner(int x, int y) {
   return abs(x) < abs(y);
}
// ... Rest weggelassen
    cout << "Minimum des Absolutbetrags mit Funktion statt Funktor: "
         << *min_element(v.begin(), v.end(), abskleiner) << endl;
```

### 24.3.11 Elemente rotieren

Der Algorithmus `rotate()` verschiebt die Elemente einer Sequenz nach links, wobei die vorne herausfallenden am Ende wieder eingefügt werden.

```
template<class ForwardIterator>
void rotate(ForwardIterator first, ForwardIterator middle,
            ForwardIterator last);
```

```
template<class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle,
                           ForwardIterator last, OutputIterator result);
```

Es handelt sich um eine Linksrotation. `first` und `last` geben den Bereich an, in dem rotiert werden soll. Der Iterator `middle` zeigt auf das Element, das nach der Rotation am Anfang der Sequenz stehen soll. Der erste Teil des Beispiels zeigt die vollständige Rotation in einem Vektor `v` um eins und dann um zwei Elemente. Die Laufvariable `shift` bestimmt, um wie viel rotiert wird. Der zweite Teil zeigt `rotate_copy()`. Dieser Algorithmus lässt den Container unverändert `v`, schreibt aber das Ergebnis in einen anderen Container, hier `erg`.

**Listing 24.34:** Rotation von Elementen

```
// cppbuch/k24/folgen/rotate.cpp
#include<algorithm>
#include<vector>
#include<numeric>
#include<showSequence.h>
using namespace std;

int main() {
   vector<int> v(10);
   iota(v.begin(), v.end(), 0);
   for(size_t shift = 1; shift < 3; shift++) {
      cout << "Rotation um " << shift << endl;
      for(size_t i = 0; i < v.size()/shift; ++i) {
         showSequence(v);
         rotate(v.begin(), v.begin() + shift, v.end());
      }
   }
   cout << "Rotation mit Kopie:" << endl;
   vector<int> erg(10);
   rotate_copy(v.begin(), v.begin() + 3, v.end(), erg.begin());
   showSequence(erg);
}
```

Das Programm gibt aus:

```
Rotation um 1
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 0
2 3 4 5 6 7 8 9 0 1
...
9 0 1 2 3 4 5 6 7 8

Rotation um 2
0 1 2 3 4 5 6 7 8 9
2 3 4 5 6 7 8 9 0 1
4 5 6 7 8 9 0 1 2 3
...
8 9 0 1 2 3 4 5 6 7
Rotation mit Kopie:
3 4 5 6 7 8 9 0 1 2
```

### 24.3.12 Elemente verwürfeln

Der Algorithmus `random_shuffle()` dient zum Mischen der Elemente einer Sequenz, also zur zufälligen Änderung ihrer Reihenfolge. Die Sequenz muss Random-Access-Iteratoren zur Verfügung stellen, zum Beispiel `vector` oder `deque`. Der Algorithmus ist in zwei Varianten vorhanden:

```
template<class RandomAccessIterator>
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last);
```

```
template<class RandomAccessIterator, class RandomNumberGenerator>
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last,
                    RandomNumberGenerator& rand);
```

Die Mischung der Elemente soll gleichverteilt sein; dies hängt natürlich vom verwendeten Zufallszahlengenerator ab. Die erste Variante benutzt eine interne, d.h. nicht in [ISOC++] spezifizierte Zufallsfunktion, die zweite den Zufallszahlengenerator von Seite 712.

**Listing 24.35:** Elemente verwürfeln

```
// cppbuch/k24/folgen/rshuffle.cpp
#include<algorithm>
#include<vector>
#include<numeric>
#include<showSequence.h>
#include<Random.h> // eigener Zufallszahlengenerator
using namespace std;

int main() {
   vector<int> v(12);
   iota(v.begin(), v.end(), 0); // 0 1 2 3 4 5 6 7 8 9 10 11
   // Benutzung des internen Zufallszahlengenerators:
   random_shuffle(v.begin(), v.end());
   showSequence(v);
   // Benutzung eines eigenen Zufallszahlengenerators:
   Random zufall;
   random_shuffle(v.begin(), v.end(), zufall);
   showSequence(v);
}
```

### 24.3.13 Dubletten entfernen

Der Algorithmus `unique()` löscht gleiche aufeinanderfolgende Elemente bis auf eins und ist als Elementfunktion von Listen bekannt (Seite 774). Er wird außerdem als globale Funktion mit einer zusätzlichen kopierenden Variante zur Verfügung gestellt:

```
template<class ForwardIterator>
ForwardIterator unique(ForwardIterator first, ForwardIterator last);
```

```
template<class ForwardIterator, class BinaryPredicate>
ForwardIterator unique(ForwardIterator first,
                       ForwardIterator last,
                       BinaryPredicate binary_pred);
```

```
template<class InputIterator, class OutputIterator>
OutputIterator unique_copy(InputIterator first,
                           InputIterator last,
                           OutputIterator result);
```

```
template<class InputIterator, class OutputIterator, class BinaryPredicate>
OutputIterator unique_copy(InputIterator first,
                           InputIterator last,
                           OutputIterator result,
                           BinaryPredicate binary_pred);
```

Im folgenden Beispiel werden überzählige gleiche Elemente eines Vektors entfernt, sodass im Ergebnis kein Element doppelt auftritt.

**Listing 24.36:** Dubletten entfernen

```
// cppbuch/k24/folgen/unique.cpp
#include<iostream>
#include<algorithm>
#include<vector>
#include<iterator>
#include<showSequence.h>
using namespace std;

int main() {
    vector<int> v(20);
    // Folge mit benachbarten gleichen Elementen
    for(size_t i = 0; i < v.size(); ++i) {
        v[i] = i/3;
    }
    showSequence(v);       // 0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6
    // Voraussetzung: Container ist sortiert
    vector<int>::iterator last = unique(v.begin(), v.end());
    showSequence(v);       // 0 1 2 3 4 5 6 2 2 3 3 3 4 4 4 5 5 5 6 6
    v.erase(last, v.end());
    showSequence(v);       // 0 1 2 3 4 5 6
}
```

**Hinweis**

`unique()` schiebt alle voneinander verschiedenen Werte nach vorn und gibt die letzte gültige Position zurück. Die Länge des Containers wird nicht verändert!

Das ist im Beispiel der Grund für das Löschen mit `erase()`. Die beiden letzten Schritte können zusammengefasst werden:

```
// Dubletten beseitigen und Vektor entsprechend kürzen
v.erase(unique(v.begin(), v.end()), v.end());
```

Sie fragen sich vielleicht, was die `unique()`-Variante mit dem binären Prädikat soll. Man kann sich vorstellen, dass ein Container, um Platz zu sparen oder aus anderen Gründen, *Zeiger* auf Objekte hält. Die Funktion `unique()` vergleicht die Objekte, um Duplikate erkennen zu können – aber natürlich ist es dabei nicht sinnvoll, Zeiger zu vergleichen. Hier kommt das binäre Prädikat ins Spiel. Im Beispiel gibt es einen Vektor `v` als Container, der Zeiger auf `string`-Objekte aufnimmt.

Schon um diesen Container sortieren zu können, dürfen nicht die Zeiger verglichen werden, sondern die daran hängenden Objekte. Aus diesem Grund wird unten dem Aufruf zum Sortieren des Vektors die binäre Vergleichsfunktion `kleiner()` und dem Aufruf von `unique()` die binäre Vergleichsfunktion `gleich()` mitgegeben. Deren beider Definition sehen Sie unten.

**Listing 24.37:** Dubletten in Container mit Zeigern entfernen

```
// cppbuch/k24/folgen/uniqueptr.cpp
#include<iostream>
#include<algorithm>
#include<vector>
#include<iterator>
#include<showSequence.h>
using namespace std;
void anzeige(const vector<string*>& v) { // Hilfsfunktion zur Abkürzung
   for(size_t i = 0; i < v.size(); ++i) {
      cout << *v[i] << " ";
   }
   cout << endl;
}

bool gleich(const string* p1, const string* p2) {
   return *p1 == *p2; // Vergleich der Werte, nicht der Zeiger
}

bool kleiner(const string* p1, const string* p2) {
   return *p1 < *p2; // Vergleich der Werte, nicht der Zeiger
}

int main() {
                    // automatische Typumwandlung in string
   string strarr[] = {"string", "array", "mit", "mit", "dubletten",
                      "dubletten", "dubletten"};
   size_t anzahl = sizeof(strarr)/sizeof(strarr[0]);
   vector<string*> v;
   for(size_t i = 0; i < anzahl; ++i) {
      v.push_back(&strarr[i]);
   }
   cout << "Original:\n";
   anzeige(v);
   // Voraussetzung für unique(): Container ist sortiert:
   sort(v.begin(), v.end(), kleiner);
   cout << "sortiert:\n";
   anzeige(v);
   v.erase(unique(v.begin(), v.end(), gleich), v.end());
   cout << "nach erase:\n";
   anzeige(v);
}
```

### 24.3.14 Reihenfolge umdrehen

`reverse()` dreht die Reihenfolge der Elemente einer Sequenz um: Die ersten werden die letzten sein – und umgekehrt. Weil das erste Element mit dem letzten vertauscht wird, das zweite mit dem zweitletzten usw., ist ein bidirektionaler Iterator erforderlich, der die Sequenz beidseitig beginnend bearbeiten kann.

```
template<class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);
```

```
template<class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first,
                            BidirectionalIterator last,
                            OutputIterator result);
```

Im folgenden Beispiel wird `reverse()` jeweils auf ein `char`-Array und einen String angewendet. `reverse_copy()` arbeitet mit einem Vektor von Strings, das Ergebnis wird im Vektor `kopie` abgelegt.

**Listing 24.38:** Reihenfolge umdrehen

```
// cppbuch/k24/folgen/reverse.cpp
#include<algorithm>
#include<string>
#include<cstring>
#include<vector>
#include<showSequence.h>
using namespace std;

int main() {
   char s0[] = "Madam";
   reverse(s0, s0 + strlen(s0));
   cout << s0 << endl;    // madaM
   string s1("ABCDEFGH");
   reverse(s1.begin(), s1.end());
   cout << s1 << endl;    // HGFEDCBA
   vector<string> vs;
   vs.push_back("eins");
   vs.push_back("zwei");
   vs.push_back("drei");
   showSequence(vs);      // eins zwei drei
   vector<string> kopie(vs.size());
   reverse_copy(vs.begin(), vs.end(), kopie.begin());
   showSequence(kopie);   // drei zwei eins
}
```

### 24.3.15 Anzahl der Elemente, die einer Bedingung genügen

Dieser Algorithmus gibt die Anzahl zurück, wie viele Elemente gleich einem bestimmten Wert `value` sind bzw. wie viele Elemente ein bestimmtes Prädikat erfüllen. Die Prototypen sind:

```
template<class InputIterator, class T>
iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last, const T& value);
```

```
template<class InputIterator, class Predicate>
iterator_traits<InputIterator>::difference_type
count_if(InputIterator first, InputIterator last, Predicate pred);
```

Das Programm zeigt beide Varianten. Die zweite mit `count_if()` benutzt ein unäres Prädikat `ungerade`, das als Funktion formuliert ist.

**Listing 24.39:** Anzahl bestimmter Elemente

```
// cppbuch/k24/folgen/count.cpp
#include<algorithm>
#include<vector>
#include<showSequence.h>
#include<Random.h>
using namespace std;

bool ungerade(int zahl) {
   return zahl % 2 != 0;
}

int main() {
   vector<int> v;
   Random zufall;
   for(size_t i = 0; i < 20; ++i) {
      v.push_back(zufall(1000));
   }
   showSequence(v);
   int gesucht = 277;
   cout << "Es sind "
        << count(v.begin(), v.end(), gesucht)
        << " Elemente mit dem Wert " << gesucht << " vorhanden." << endl;

   cout << "Es sind "
        << count_if(v.begin(), v.end(), ungerade)
        << " ungerade Elemente vorhanden." << endl;
}
```

### 24.3.16 Gilt X für alle, keins oder wenigstens ein Element einer Folge?

X steht dabei für ein Prädikat. Die zugehörigen Algorithmen sind neu in den Standard aufgenommen worden.

```
template<class Iterator, class Predicate>
bool all_of(Iterator first, Iterator last, Predicate pred);
```

gibt `true` zurück, wenn für *alle* Iteratoren `i` zwischen `first` und `last` (ausschließlich) `(pred(*i) == true)` gilt.

```
template<class Iterator, class Predicate>
bool none_of(Iterator first, Iterator last, Predicate pred);
```

gibt `true` zurück, wenn für *keinen* Iterator `i` zwischen `first` und `last` (ausschließlich) `(pred(*i) == true)` gilt.

```
template<class Iterator, class Predicate>
bool any_of(Iterator first, Iterator last, Predicate pred);
```

gibt `true` zurück, wenn für *wenigstens einen* Iterator `i` zwischen `first` und `last` (ausschließlich) `(pred(*i) == true)` gilt.

```
// Auszug aus cppbuch/k24/folgen/all_any_none.cpp
struct istPositiv { // zu prüfendes Prädikat
   bool operator()(int x) const {
      return x >= 0;
   }
};

int main() {
   vector<int> folge(12);
   for(size_t i = 0; i < folge.size(); ++i) {
      folge[i] = -i-1; // ggf. je nach Fall verändern
   }
   cout << "Folge = ";
   showSequence(folge);
   if(all_of(folge.begin(), folge.end(), istPositiv())) {
      cout << "Bedingung >=0 gilt für alle Elemente" << endl;
   }
   else {
      cout << "Bedingung >=0 gilt nicht für alle Elemente" << endl;
   }
   if(none_of(folge.begin(), folge.end(), istPositiv())) {
      cout << "Bedingung >= 0 gilt für kein Element" << endl;
   }
   else {
      cout << "Bedingung >=0 gilt für mindestens ein Element" << endl;
   }
   if(any_of(folge.begin(), folge.end(), istPositiv())) {
      cout << "Bedingung >= 0 gilt für wenigstens ein Element" << endl;
   }
   else {
      cout << "Bedingung >=0 gilt für kein Element" << endl;
   }
}
```

### 24.3.17 Permutationen

Eine Permutation entsteht aus einer Sequenz durch Vertauschung zweier Elemente. (0, 2, 1) ist eine Permutation, die aus (0, 1, 2) entstanden ist. Für eine Sequenz mit $N$ Elementen gibt es $N! = N(N-1)(N-2)...2 \cdot 1$ Permutationen, das heißt $3 \cdot 2 \cdot 1 = 6$ im obigen Beispiel:

(0, 1, 2), (0, 2, 1), (1, 0, 2), (1, 2, 0), (2, 0, 1), (2, 1, 0)

Man kann sich die Menge aller $N!$ Permutationen einer Sequenz wie oben geordnet vorstellen, sei es, dass die Ordnung mit dem <-Operator oder mit einem Vergleichsobjekt `comp` hergestellt wurde. Aus der Ordnung ergibt sich eine eindeutige Reihenfolge, sodass die nächste oder die vorhergehende Permutation eindeutig bestimmt ist. Dabei wird die Folge zyklisch betrachtet, das heißt, die auf (2, 1, 0) folgende Permutation ist (0, 1, 2). Die Algorithmen `prev_permutation()` und `next_permutation()` verwandeln eine Sequenz in die jeweils vorhergehende bzw. nächste Permutation:

```
template<class BidirectionalIterator>
bool prev_permutation(BidirectionalIterator first, BidirectionalIterator last);
```

```
template<class BidirectionalIterator, class Compare>
bool prev_permutation(BidirectionalIterator first,
                      BidirectionalIterator last, Compare comp);

template<class BidirectionalIterator>
bool next_permutation(BidirectionalIterator first, BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
bool next_permutation(BidirectionalIterator first, BidirectionalIterator last
                      Compare comp);
```

**Listing 24.40:** Permutationen

```
// cppbuch/k24/folgen/permute.cpp
#include<algorithm>
#include<showSequence.h>
#include<vector>
#include<numeric>
using namespace std;

long fakultaet(unsigned n) { // Fakultät n! berechnen
   long fac = 1;
   while(n > 1) {
      fac *= n--;
   }
   return fac;
}

int main() {
   vector<int> v(4);
   iota(v.begin(), v.end(), 0);     // 0 1 2 3
   long anzahl = fakultaet(v.size()); // Anzahl der Permutationen
   for(int i = 0; i < anzahl; ++i) {
      if(!prev_permutation(v.begin(), v.end())) {
         cout << "Zyklusbeginn:\n";       // siehe Text
      }
      showSequence(v);
   }
}
```

Wenn eine Permutation gefunden wird, ist der Rückgabewert `true`. Andernfalls handelt es sich um das Ende eines Zyklus. Dann wird `false` zurückgegeben und die Sequenz in die kleinstmögliche (bei `next_permutation()`) beziehungsweise die größtmögliche (bei `prev_permutation()`) entsprechend dem Sortierkriterium verwandelt. Das Beispiel produziert zuerst die Meldung »Zyklusbeginn«, weil die Vorbesetzung des Vektors mit (0, 1, 2, 3) die Bestimmung einer *vorherigen* Permutation nicht ohne Zyklusüberschreitung erlaubt. Deswegen wird die nach der Sortierung größte Sequenz, nämlich (3, 2, 1, 0), als Nächstes gebildet. Die Meldung »Zyklusbeginn« entfiele, wenn im Beispiel `prev_permutation()` durch `next_permutation()` ersetzt oder wenn alternativ ein Vergleichsobjekt `greater<int>()` als dritter Parameter übergeben würde.

### 24.3.18 Lexikografischer Vergleich

Der lexikografische[1] Vergleich dient zum Vergleich zweier Sequenzen, die durchaus verschiedene Längen haben können. Die Funktion gibt `true` zurück, wenn die erste Sequenz lexikografisch kleiner ist. Falls eine der beiden Sequenzen bereits vollständig durchsucht ist, ehe ein unterschiedliches Element gefunden wurde, gilt die kürzere Sequenz als kleiner. Die Prototypen sind:

```
template<class InputIterator1, class InputIterator2>
bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             Compare comp);
```

Damit können z.B. Zeichenketten alphabetisch sortiert werden. Ein Anwendungsbeispiel:

**Listing 24.41:** Lexikografischer Vergleich

```
// cppbuch/k24/folgen/lexicmp.cpp
#include<algorithm>
#include<iostream>
#include<functional>
using namespace std;

int main () {
  char text1[] = "Arthur";
  int length1 = sizeof(text1);
  char text2[] = "Vera";
  int length2 = sizeof(text2);
  if(lexicographical_compare(text1, text1 + length1, text2, text2 + length2)) {
    cout << text1 << " kommt vor " << text2 << endl;
  }
  else {
    cout << text2 << " kommt vor " << text1 << endl;
  }
  if(lexicographical_compare(text1, text1 + length1, text2, text2 + length2,
           greater<char>())) {   // umgekehrte Reihenfolge
    cout << text1 << " kommt nach " << text2 << endl;
  }
  else {
    cout << text2 << " kommt nach " << text1 << endl;
  }
}
```

[1] Siehe Glossar Seite 953

# 24.4 Sortieren und Verwandtes

## 24.4.1 Partitionieren

Eine Sequenz kann mit `partition()` so in zwei Bereiche zerlegt werden, dass alle Elemente, die einem bestimmten Kriterium `pred` genügen, anschließend vor allen anderen liegen. Es wird ein Iterator zurückgegeben, der auf den Anfang des zweiten Bereichs zeigt. Alle vor diesem Iterator liegenden Elemente genügen dem Prädikat. Eine typische Anwendung für eine derartige Zerlegung findet sich im bekannten Quicksort-Algorithmus.

Die zweite Variante `stable_partition()` garantiert darüber hinaus, dass die relative Ordnung der Elemente innerhalb eines Bereichs erhalten bleibt. Diese zweite Variante ist von der Funktion her ausreichend, sodass man die erste normalerweise nicht benötigt. Bei knappem Speicher benötigt die zweite Variante jedoch geringfügig mehr Laufzeit ($O(N \log N)$ statt $O(N)$, $N = last - first$), sodass es beide Varianten gibt. Die Prototypen sind:

```
template<class BidirectionalIterator, class Predicate>
BidirectionalIterator partition(BidirectionalIterator first,
                                BidirectionalIterator last,
                                Predicate pred);
```

```
template<class BidirectionalIterator, class Predicate>
BidirectionalIterator stable_partition(
                                BidirectionalIterator first,
                                BidirectionalIterator last,
                                Predicate pred);
```

Im folgenden Beispiel wird ein Vektor mit negativen und positiven Zahlen erzeugt, die zufällig durcheinandergewürfelt werden. Zwei Kopien des Vektors (`stable` und `unstable` genannt) werden partitioniert, wobei in den Partitionen des Vektors `stable` die ursprüngliche Reihenfolge der Elemente erhalten bleibt.

**Listing 24.42:** Folge partitionieren

```
// cppbuch/k24/sortieren/partition.cpp
#include<algorithm>
#include<vector>
#include<numeric>
#include<functional>
#include<showSequence.h>
using std::bind;
using namespace std::placeholders;
using namespace std;

int main() {
   vector<int> v(12);
   iota(v.begin(), v.end(), -6);
   random_shuffle(v.begin(), v.end());
   vector<int> unstable = v, stable = v;
   partition(unstable.begin(), unstable.end(), bind(less<int>(), _1, 0));
   stable_partition(stable.begin(), stable.end(), bind(less<int>(), _1, 0));
```

```
    cout << "In negative und positive Elemente zerlegen\n";
    cout << "Sequenz vor der Zerlegung :";
    showSequence(v);        // -5 -1 3 2 -3 5 -4 -6 4 0 1 -2
    cout << "stabile Partitionierung    :";
    showSequence(stable); // -5 -1 -3 -4 -6 -2 3 2 5 4 0 1
    cout << "unstabile Partitionierung :";
    // Die negativen Elemente sind nicht mehr in ihrer ursprünglichen Reihenfolge
    showSequence(unstable); // -5 -1 -2 -6 -3 -4 5 2 4 0 1 3
}
```

### 24.4.2 Sortieren

Der Algorithmus `sort()` sortiert zwischen den Iteratoren `first` und `last`. Er ist nur für Container mit Random-Access-Iteratoren geeignet, wie zum Beispiel `vector` oder `deque`. Ein wahlfreier Zugriff auf Elemente einer Liste ist nicht möglich, deshalb ist für eine Liste vom Typ `list` die dafür definierte Elementfunktion `list::sort()` zu nehmen.

```
template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);
```

```
template<class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

Die Sortierung ist nicht stabil, das heißt, dass verschiedene Elemente, die jedoch denselben Sortierschlüssel haben, in der sortierten Folge nicht unbedingt dieselbe Reihenfolge untereinander wie vorher in der unsortierten Folge haben. Der Aufwand ist im Mittel $O(N\log N)$ mit $N$ = `last - first`. Über das Verhalten im schlechtesten Fall (englisch *worst case*) wird keine Aufwandsschätzung gegeben. Falls das Worst-case-Verhalten wichtig ist, wird jedoch empfohlen, lieber `stable_sort()` zu verwenden. Ein Beispiel finden Sie im nächsten Abschnitt.

### 24.4.3 Stabiles Sortieren

Die Komplexität von `stable_sort()` ist auch im schlechtesten Fall $O(N\log N)$, falls genug Speicher zur Verfügung steht. Andernfalls ist der Aufwand höchstens $O(N(\log N)^2)$. Der Algorithmus basiert intern auf dem Sortieren durch Verschmelzen (*merge sort*, siehe mehr dazu auf Seite 672), das im Durchschnitt um einen konstanten Faktor von etwa 1,4 mehr Zeit als Quicksort benötigt. Dem zeitlichen Mehraufwand von 40 % stehen das sehr gute Verhalten im schlechtesten Fall und die Stabilität von `stable_sort()` gegenüber. Die Deklarationen von `stable_sort()` sind:

```
template<class RandomAccessIterator>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);
```

```
template<class RandomAccessIterator, class Compare>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
```

`sort()` und `stable_sort()` arbeiten mit dem <-Operator der Elemente oder einem Comparator-Objekt bzw. einer Funktion zum Vergleich. Im Beispiel werden beide Varianten gezeigt und der Unterschied zwischen stabiler und unstabiler Sortierung demonstriert.

**Listing 24.43:** Sortieren

```
// cppbuch/k24/sortieren/sort.cpp
#include<algorithm>
#include<vector>
#include<showSequence.h>
#include<Random.h> // von Seite 712
using namespace std;

bool intVergleich(double x, double y) {
   return long(x) < long(y);
}

int main() {
    vector<double> v(17);
    Random zufall;
    // Vektor mit Zufallswerten initialisieren. Dabei sollen einige Werte denselben
    // ganzzahligen Anteil haben, um den Unterschied von stabiler und nicht-stabiler
    // Sortierung sehen zu können.
    for(size_t i = 0; i < v.size(); ++i) {
       v[i] = zufall(v.size()/3)
              + double(zufall(1000)/1000.0);
    }
    vector<double> unstable = v,    // Hilfsvektoren
                   stable = v;
    cout << "Vektor           :\n";
    showSequence(v);

    // Sortierung mit < Operator:
    stable_sort(stable.begin(), stable.end());
    cout << "Kein Unterschied, weil double-Zahlen als Key genommen werden\n"
         "stabile  Sortierung    :\n";
    showSequence(stable);

    sort(unstable.begin(), unstable.end());
    cout << "unstabile Sortierung :\n";
    showSequence(unstable);

    // Sortierung mit Funktion statt <
    unstable = v;
    stable = v;
    cout << "Unterschied, weil nur der int-Teil Sortierkriterium  ist\n";

    stable_sort(stable.begin(), stable.end(),intVergleich);
    cout << "stabile Sortierung (int-Kriterium)  :\n";
    showSequence(stable);

    sort(unstable.begin(), unstable.end(), intVergleich);
    cout << "unstabile Sortierung (int-Kriterium)  :\n";
    showSequence(unstable);
}
```

### 24.4.4 Partielles Sortieren

Teilweises Sortieren bringt die *M* kleinsten Elemente nach vorn, der Rest bleibt unsortiert. Der Algorithmus verlangt jedoch nicht die Zahl *M*, sondern einen Iterator `middle` auf die entsprechende Position, sodass *M* = `middle - first` gilt. Die Prototypen sind:

```
template<class RandomAccessIterator>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                  RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                  RandomAccessIterator last, Compare comp);
```

Die Komplexität ist etwa $O(N \log M)$. Der Programmauszug für einen Vektor `v` zeigt die teilweise Sortierung. Im Ergebnis sind in der ersten Hälfte alle Elemente kleiner als in der zweiten. In der ersten Hälfte sind sie darüber hinaus sortiert, in der zweiten jedoch nicht.

```
   partial_sort(v.begin(), v.begin() + v.size()/2, v.end());
```

Beide Varianten gibt es auch in einer kopierenden Form, wobei `result_first` bzw. `result_last` sich auf den Zielcontainer beziehen. Die Anzahl der sortierten Elemente ergibt sich aus der kleineren der beiden Differenzen `result_last - result_first` bzw. `last - first`.

```
template<class InputIterator, class RandomAccessIterator>
RandomAccessIterator partial_sort_copy(
                  InputIterator first, InputIterator last,
                  RandomAccessIterator result_first,
                  RandomAccessIterator result_last);

template<class InputIterator, class RandomAccessIterator, class Compare>
RandomAccessIterator partial_sort_copy(
                  InputIterator first, InputIterator last,
                  RandomAccessIterator result_first,
                  RandomAccessIterator result_last,
                  Compare comp);
```

Der zurückgegebene Random-Access-Iterator zeigt auf das Ende des beschriebenen Bereichs, also auf `result_last` oder auf `result_first + (last - first)`, je nachdem, welcher Wert kleiner ist.

### 24.4.5 Das n.-größte oder n.-kleinste Element finden

Das *n.*-größte oder *n.*-kleinste Element einer Sequenz mit Random-Access-Iteratoren kann mit `nth_element()` gefunden werden.

```
template<class RandomAccessIterator>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                 RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                 RandomAccessIterator last, Compare comp);
```

Der Iterator `nth` wird auf die gewünschte Stelle gesetzt, zum Beispiel auf den Beginn des Containers. Nach Aufruf von `nth_element()` ist das kleinste Element an diese Stelle gerutscht. Die Reihenfolge der Elemente im Container wird also *geändert*. Falls `nth` vor Aufruf zum Beispiel auf die Position `v.begin() + 6` zeigt, steht dort anschließend das siebtkleinste Element.

Nach Aufruf des Algorithmus stehen links von `nth` nur Elemente, die kleiner oder gleich `(*nth)` und allen Elementen rechts davon sind. Der Aufwand des Algorithmus ist im Durchschnitt linear ($O(N)$). Der Aufwand in der vorliegenden Implementierung ist im schlechtesten, wenn auch seltenen Fall $O(N^2)$, weil ein Quicksort-ähnlicher Zerlegungsmechanismus verwendet wird. Das Beispiel ermittelt das kleinste und das größte Element sowie den Medianwert. Der Medianwert teilt eine Folge, sodass die Hälfte der Werte größer gleich und die Hälfte der Werte kleiner gleich dem Medianwert ist.

**Listing 24.44:** n.-größtes und n.-kleinstes Element finden

```
// cppbuch/k24/sortieren/nth.cpp
#include<algorithm>
#include<deque>
#include<functional>
#include<showSequence.h>
#include<Random.h> // von Seite 712
using namespace std;

int main() {
   deque<int> d;
   Random zufall;
   for(size_t i = 0; i < 15; ++i) {
      d.push_front(zufall(100));
   }
   showSequence(d); // 95 51 36 62 47 55 27 76 33 19 91 79 78 39 84
   deque<int>::iterator nth = d.begin();
   nth_element(d.begin(), nth, d.end());
   cout << "Kleinstes Element: " << (*nth) << endl;          // 19

   // Das Standard-Vergleichsobjekt greater dreht die Reihenfolge um.
   // In diesem Fall stejt das größte Objekt an der ersten Position.
   // Hier ist immer noch nth == d.begin().
   nth_element(d.begin(), nth, d.end(), greater<int>());
   cout << "Größtes Element : " << (*nth) << endl;          // 95

   // Mit dem < -Operator steht das größte Element am Ende:
   nth = d.end();
   --nth;              // zeigt nun auf das letzte Element
   nth_element(d.begin(), nth, d.end());
   cout << "Größtes Element : "  << (*nth) << endl;         // 95

   // Median
   nth = d.begin() + d.size()/2;
   nth_element(d.begin(), nth, d.end());
   cout << "Medianwert     :" << (*nth) << endl;           // 55
}
```

### 24.4.6 Verschmelzen (merge)

Verschmelzen, auch Mischen genannt, ist ein Verfahren, zwei sortierte Sequenzen zu einer zu vereinigen. Dabei werden schrittweise die jeweils ersten Elemente beider Sequenzen verglichen, und es wird das kleinere (oder größere, je nach Sortierkriterium) Element in die Ausgabesequenz gepackt. Die Prototypen sind:

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result, Compare comp);
```

`merge()` setzt eine vorhandene Ausgabesequenz voraus, die ausreichend Platz haben muss. Falls eine der beiden Eingangssequenzen erschöpft ist, wird der Rest der anderen in die Ausgabe kopiert. Ein kleines Programm soll dies zeigen:

**Listing 24.45:** Sortieren durch Verschmelzen

```
// cppbuch/k24/sortieren/merge0.cpp
#include<algorithm>
#include<numeric>
#include<vector>
#include<showSequence.h>
using namespace std;

int main() {
   vector<int> folge1(6);
   iota(folge1.begin(), folge1.end(), 0); // initialisieren
   showSequence(folge1);

   vector<int> folge2(10);
   iota(folge2.begin(), folge2.end(), 0); // initialisieren
   showSequence(folge2);

   // Verschmelzen zweier Folgen v1 und v2, Ablage in result
   vector<int> result(folge1.size()+folge2.size()); // Platz schaffen
   merge(folge1.begin(), folge1.end(),
         folge2.begin(), folge2.end(),
         result.begin());
   showSequence(result);
}
```

Das Ergebnis von *merge0.cpp*:

```
0 1 2 3 4 5
0 1 2 3 4 5 6 7 8 9
0 0 1 1 2 2 3 3 4 4 5 5 6 7 8 9
```

Vom Prinzip her erlaubt das Verschmelzen sehr schnelles Sortieren der Komplexität $O(N \log N)$ nach dem rekursiven Schema

1. Teile die Liste in zwei Hälften.
2. Falls die Hälften mehr als ein Element haben, sortiere beide Hälften mit *diesem Verfahren* (Rekursion).
3. Beide Hälften zur Ergebnisliste verschmelzen.

Eine nichtrekursive Variante ist natürlich möglich. Die Sortierung ist stabil. Der Nachteil besteht im notwendigen zusätzlichen Speicher für das Ergebnis. Zum Vergleich mit dem obigen Schema sei der Merge-Sort genannte Algorithmus mit den Mitteln der C++-Standardbibliothek formuliert (ohne dass er selbst dazugehört!):

**Listing 24.46:** Mergesort

```
// cppbuch/k24/sortieren/mergesort.t
#include<algorithm>

template<class ForwardIterator, class OutputIterator>
  void mergesort(ForwardIterator first, ForwardIterator last,
                 OutputIterator result) {
  // auto statt typename std::iterator_traits<ForwardIterator>::difference_type
  auto n   = std::distance(first, last);
  auto haelfte = n/2;
  ForwardIterator mitte = first;
  std::advance(mitte, haelfte);
  if(haelfte > 1) {       // linke Hälfte sortieren, falls notwendig
    mergesort(first, mitte, result); // Rekursion
  }

  if(n - haelfte > 1) {   // rechte Hälfte sortieren, falls notwendig
    OutputIterator ergebnis = result;
    std::advance(ergebnis, haelfte);
    mergesort(mitte, last, ergebnis); // Rekursion
  }

  // beide Hälften mischen und zurückkopieren
  OutputIterator end = std::merge(first, mitte, mitte, last, result);
  std::copy(result, end, first);
}
```

Die letzten beiden Anweisungen der Funktion können auf Kosten der Lesbarkeit zusammengefasst werden, wie es oft in der Implementierung der C++-Standardbibliothek zu finden ist:

```
// Beide Hälften verschmelzen und Ergebnis zurückkopieren
copy(result, merge(first, mitte, mitte, last, result), first);
```

Der Vorteil des hier beschriebenen Algorithmus gegenüber `stable_sort()` besteht darin, dass nicht nur Container, die mit Random-Access-Iteratoren zusammenarbeiten, sortiert werden können. Es genügen bidirektionale Iteratoren, sodass `v` im obigen Programm auch eine Liste sein kann. Sie kann mit `push_front()` gefüllt werden. Voraussetzung ist nur, dass eine Liste `puffer` vorhanden ist, die mindestens so viele Elemente wie `v` hat.

**Listing 24.47:** Mergesort-Anwendung

```
// cppbuch/k24/sortieren/mergesort_list.cpp
#include<algorithm>
#include<list>
#include<showSequence.h>
#include"mergesort.t"
#include<Random.h>
using namespace std;

int main() {    // mit Liste statt Vektor
   list<int> liste;
   Random zufall;
   for(size_t i = 0; i < 20; ++i) {
      liste.push_front(zufall(1000)); // pseudo-zufällige Zahlen
   }
   showSequence(liste);
   list<int> buffer = liste;
   mergesort(liste.begin(), liste.end(), buffer.begin());
   showSequence(liste);   // sortierte Zahlen
}
```

## Verschmelzen an Ort und Stelle (inplace_merge)

Wenn Sequenzen an Ort und Stelle gemischt werden sollen, muss der Weg über einen Pufferspeicher gehen. Die Funktion `inplace_merge()` mischt Sequenzen so, dass das Ergebnis an die Stelle der Eingangssequenzen tritt. Die Prototypen sind:

```
template<class BidirectionalIterator>
void inplace_merge(BidirectionalIterator first,
                   BidirectionalIterator middle,
                   BidirectionalIterator last);
```

```
template<class BidirectionalIterator, class Compare>
void inplace_merge(BidirectionalIterator first,
                   BidirectionalIterator middle,
                   BidirectionalIterator last,
                   Compare comp);
```

Der Pufferspeicher wird intern und implementationsabhängig bereitgestellt.

**Listing 24.48:** inplace_merge()

```
// cppbuch/k24/sortieren/merge1.cpp
#include<algorithm>
#include<vector>
#include<showSequence.h>
using namespace std;

int main() {
   vector<int> v(16);                  // gerade Zahl
   int middle = v.size()/2;
   for(int i = 0; i < middle; ++i) {
      v[i]       = 2*i;                // gerade
```

```
        v[middle + i] = 2*i + 1;      // ungerade
    }
    showSequence(v);
    inplace_merge(v.begin(), v.begin() + middle, v.end());
    showSequence(v);
}
```

Die erste Hälfte eines Vektors wird hier mit geraden Zahlen belegt, die zweite mit ungeraden. Nach dem Verschmelzen enthält derselbe Vektor alle Zahlen, ohne dass explizit ein Ergebnisbereich angegeben werden muss:

*0 2 4 6 8 10 12 14 1 3 5 7 9 11 13 15* vorher
*0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15* nachher

# 24.5 Suchen und Finden

## 24.5.1 Element finden

Der Algorithmus `find()` tritt in zwei Arten auf: ohne und mit erforderlichem Prädikat (als `find_if()`). Es wird die Position in einem Container gesucht, an der ein bestimmtes Element zu finden ist. Das Ergebnis ist ein Iterator, der auf die gefundene Stelle zeigt oder gleich dem Ende-Iterator `end()` ist. Die Prototypen sind:

```
template<class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value);

template<class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last, Predicate pred);
```

Im folgenden Beispiel wird nur die zweite Variante gezeigt. Es wird die erste ungerade Zahl im Vektor `v` gesucht:

**Listing 24.49:** Erste ungerade Zahl suchen

```
// cppbuch/k24/finden/find_if.cpp
#include<algorithm>
#include<vector>
#include<iostream>
#include<showSequence.h>
using namespace std;

class Ungerade {
public:
   bool operator()(int x) const {
      return x % 2 != 0;
   }
};

int main() {
```

```
  vector<int> v(8);
  for(size_t i = 0; i < v.size(); ++i) {
    v[i] = 2*i;                    // nur gerade Zahlen
  }
  v[5] = 99;                       // eine ungerade Zahl
  showSequence(v);
  // nach ungerader Zahl suchen
  auto iter = find_if(v.begin(), v.end(), Ungerade());
  if(iter != v.end()) {
    cout << "Die erste ungerade Zahl (" << *iter
         << ") wurde an Position " << (iter - v.begin())
         << " gefunden." << endl;
  }
  else {
    cout << "Keine ungerade Zahl gefunden." << endl;
  }
}
```

Man kann ohne die Funktorklasse `Ungerade` auskommen, wenn man eine der Standardfunktionen benutzt:

```
// Auszug aus cppbuch/k24/finden/find_if2.cpp
vector<int>::iterator iter
   = find_if(v.begin(), v.end(), bind(modulus<int>(), _1, 2));
```

### 24.5.2 Element einer Menge in der Folge finden

Der Algorithmus findet das erste Auftreten eines Elements einer Menge innerhalb einer Sequenz. Die Prototypen sind:

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_first_of(
                   ForwardIterator1 first1, ForwardIterator1 last1,
                   ForwardIterator2 first2, ForwardIterator2 last2);
```

```
template<class ForwardIterator1, ForwardIterator2, class BinaryPredicate>
ForwardIterator1 find_first_of(
                    ForwardIterator1 first1, ForwardIterator1 last1,
                    ForwardIterator2 first2, ForwardIterator2 last2,
                    BinaryPredicate pred);
```

Das Intervall `[first1, last1)` ist der zu durchsuchende Bereich, das Intervall `[first2, last2)` beschreibt einen Bereich mit zu suchenden Elementen. Zurückgegeben wird der erste Iterator `i` im zu durchsuchenden Bereich, der auf ein Element zeigt, das auch im zweiten Bereich vorhanden ist. Es sei angenommen, dass ein Iterator `j` auf das Element im zweiten Bereich zeigt. Dann gilt

`*i == *j` beziehungsweise

`pred(*i, *j) == true`.

Falls kein Element aus dem ersten Bereich im zweiten Bereich gefunden wird, gibt der Algorithmus `last1` zurück. Die Komplexität ist $O(N_1 * N_2)$, wenn $N_1$ und $N_2$ die Längen der Bereiche sind. Das nachfolgende Programm gibt Folgendes aus:

*Ist eines der Elemente der Menge (1 5 7 ) in der Folge*
*0 2 4 6 8 10 12 14 enthalten?*
*Nein.*
*Menge modifiziert:*
*Ist eines der Elemente der Menge (1 6 7 ) in der Folge*
*0 2 4 6 8 10 12 14 enthalten?*
*Ja. Element 6 ist in beiden Bereichen vorhanden. Sein erstes Vorkommen in der Folge ist Position 3.*

**Listing 24.50:** Erstes Auftreten finden

```
// Auszug aus cppbuch/k24/finden/find_first_of.cpp
#include<algorithm>
#include<vector>
#include<iostream>
#include<showSequence.h>
using namespace std;

int main() {
   vector<int> folge(8);
   vector<int> menge(3);
   // Folge und Menge initialisieren
   for(size_t i = 0; i < folge.size(); ++i) {
      folge[i] = 2*i;   // gerade Zahlen
   }
   menge[0] = 1;
   menge[1] = 5;
   menge[2] = 7;
   // Zwei Tests:
   for(int testNummer = 0; testNummer < 2; ++testNummer) {
      if(testNummer == 1) {
         menge[1] = 6;
         cout << endl << "Menge modifiziert:" << endl;
      }
      cout << "Ist eines der Elemente der Menge (";
      showSequence(menge, "");
      cout << ") in der Folge" << endl;
      showSequence(folge, "");
      cout << " enthalten?" << endl;
      // Suche nach Element, das auch in der Menge enthalten ist
      auto iter = find_first_of(folge.begin(), folge.end(),
                                menge.begin(), menge.end());
      if(iter != folge.end()) {
         cout << "Ja. Element " << *iter
              << " ist in beiden Bereichen vorhanden. Sein erstes Vorkommen"
            " in der Folge ist Position "
              << (iter - folge.begin()) << "." << endl;
      }
      else {
         cout << "Nein." << endl;
      }
   }
}
```

### 24.5.3 Teilfolge finden

Der Algorithmus `search()` durchsucht eine Sequenz, ob eine zweite Sequenz in ihr enthalten ist. Es wird ein Iterator auf die Position innerhalb der ersten Sequenz zurückgegeben, an der die zweite Sequenz beginnt, sofern sie in der ersten enthalten ist. Andernfalls wird ein Iterator auf die `last1`-Position der ersten Sequenz zurückgegeben. Die Prototypen sind:

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
                        ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
                        ForwardIterator2 first2, ForwardIterator2 last2,
                        BinaryPredicate binary_pred);
```

Geht es darum, das letzte Auftreten einer Teilfolge zu finden, bietet sich der Algorithmus `find_end()` an (siehe unten). Ein Beispiel für `search()`:

**Listing 24.51:** Teilfolge finden

```
// cppbuch/k24/finden/search.cpp
#include<algorithm>
#include<vector>
#include<iostream>
#include<cstdlib>
#include<showSequence.h>

class Absolutvergleich { // Vorzeichen ignorieren
    public:
      bool operator()(int x, int y) {
         return abs(x) == abs(y);
      }
};
using namespace std;

int main() {
    vector<int> folge(12);
    for(size_t i = 0; i < folge.size(); ++i) {
       folge[i] = i;      // 0 1 2 3 4 5 6 7 8 9 10 11
    }
    cout << "Folge = ";
    showSequence(folge);

    vector<int> teilfolge(4);
    for(size_t i = 0; i < teilfolge.size(); ++i)
       teilfolge[i] = i + 5;  // 5 6 7 8
    cout << "Teilfolge = ";
    showSequence(teilfolge);

    // Teilfolge in Folge suchen
    auto position = search(folge.begin(), folge.end(),
                           teilfolge.begin(), teilfolge.end());
```

```
    if(position != folge.end()) {
        cout << " Die Teilfolge ist in der Folge ab Position "
             << (position - folge.begin())
             << " enthalten." << endl;
    }
    else {
        cout << "Die Teilfolge ist nicht in der Folge enthalten." << endl;
    }
    // Fall2: binäres Prädikat. Dafür negative Zahlen setzen
    for(size_t i = 0; i < teilfolge.size(); ++i) {
        teilfolge[i] = -(i + 5); // -5 -6 -7 -8
    }
    cout << "Teilfolge = ";
    showSequence(teilfolge);

    // Teilfolge in Folge suchen, dabei Vorzeichen ignorieren
    position = search(folge.begin(), folge.end(),
                      teilfolge.begin(), teilfolge.end(),
                      Absolutvergleich());
    if(position != folge.end()) {
        cout << " Die Teilfolge ist in der Folge ab Position "
             << (position - folge.begin())
             << " enthalten. Vorzeichen wurden ignoriert" << endl;
    }
    else {
        cout << "Die Teilfolge ist nicht in der Folge enthalten." << endl;
    }
}
```

## Letztes Auftreten einer Teilfolge finden

Der Algorithmus findet das letzte Auftreten einer Teilfolge innerhalb einer Sequenz. Die Prototypen sind:

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1,
                          ForwardIterator2 first2, ForwardIterator2 last2);
```

```
template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1,
                          ForwardIterator2 first2, ForwardIterator2 last2,
                          BinaryPredicate pred);
```

Das Intervall `[first1, last1)` ist der zu durchsuchende Bereich, das Intervall `[first2, last2)` beschreibt die zu suchende Folge. Zurückgegeben wird der letzte Iterator im zu durchsuchenden Bereich, der auf den Beginn der Teilfolge zeigt. Falls die Teilfolge nicht gefunden wird, gibt der Algorithmus `last1` zurück. Falls der zurückgegebene Iterator mit `i` bezeichnet wird, gilt

`*(i+n) == *(first2+n)` beziehungsweise

`pred(*(i+n), *(first2+n)) == true`

für alle `n` im Bereich 0 bis (`last2-first2`). Die Komplexität ist $O(N_2(N_1 - N_2))$, wenn $N_1$ und $N_2$ die Länge des zu durchsuchenden Bereichs bzw. der zu suchenden Teilfolge sind. Die Benutzung entspricht der von `search()`, sodass hier nur ein Auszug eines Beispiels gezeigt wird:

```
// Auszug aus cppbuch/k24/finden/find_end.cpp
vector<int>::const_iterator iter
   = find_end(folge.begin(), folge.end(), teilfolge1.begin(), teilfolge1.end());
```

### 24.5.4 Bestimmte benachbarte Elemente finden

Zwei gleiche, direkt benachbarte (englisch *adjacent*) Elemente werden mit der Funktion `adjacent_find()` gefunden. Es gibt auch hier zwei überladene Varianten – eine ohne und eine mit binärem Prädikat. Die erste Variante vergleicht die Elemente mit dem Gleichheitsoperator `==`, die zweite benutzt das Prädikat. Die Prototypen sind:

```
template<class ForwardIterator>
ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last);
```

```
template<class ForwardIterator, class BinaryPredicate>
ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last,
                              BinaryPredicate binary_pred);
```

Der zurückgegebene Iterator zeigt auf das erste der beiden Elemente, sofern ein entsprechendes Paar gefunden wird. Beispiel:

**Listing 24.52:** Gleiche benachbarte Elemente finden

```
// cppbuch/k24/finden/adjacent_find.cpp
#include<algorithm>
#include<vector>
#include<iostream>
#include<showSequence.h>
using namespace std;

int main() {
   vector<int> v(8);
   for(size_t i = 0; i < v.size(); ++i) {
      v[i] = 2*i;              // gerade
   }
   v[5] = 99;                  // 2 gleiche benachbarte Elemente
   v[6] = 99;
   showSequence(v);
   auto iter = adjacent_find(v.begin(), v.end()); // finde gleiche Nachbarn
   if(iter != v.end()) {
      cout << "Die ersten gleichen benachbarten Zahlen ("
           << *iter << ") wurden an Position "
           << (iter - v.begin()) << " gefunden." << endl;
   }
   else {
      cout << "Keine gleichen Nachbarn gefunden." << endl;
   }
}
```

Wenn zum Beispiel nach einem doppelt so großen Nachfolger gesucht werden soll, kommt ein binäres Prädikat zum Einsatz, zum Beispiel

```
class IstDoppeltSoGross {
   public:
      bool operator()(int a, int b) const { return (b == 2*a);}
};
```

Der entsprechende Aufruf, um den doppelt so großen Nachfolger zu finden, lautet:

```
// Auszug aus cppbuch/k24/finden/adjacent_find_1.cpp
auto iter = adjacent_find(v.begin(), v.end(), IstDoppeltSoGross());
```

Die Auswertung kann wie oben mit der Iterator-Differenz `(iter - v.begin())` geschehen.

### 24.5.5 Bestimmte aufeinanderfolgende Werte finden

Der Algorithmus `search_n()` durchsucht eine Sequenz daraufhin, ob eine Folge von gleichen Werten in ihr enthalten ist. Die Prototypen sind:

```
template<class ForwardIterator, class Size, class T>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                         Size count, const T& value);
```

```
template<class ForwardIterator, class Size, class T, class BinaryPredicate>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                         Size count, const T& value,
                         BinaryPredicate binary_pred);
```

Zurückgegeben wird von der ersten Funktion der Iterator auf den Beginn der ersten Folge mit wenigstens `count` Werten, die gleich `value` sind. Falls eine derartige Folge nicht gefunden wird, gibt die Funktion `last` zurück. Die zweite Funktion prüft nicht auf Gleichheit, sondern wertet das binäre Prädikat aus. Im Erfolgsfall muss für wenigstens `count` aufeinanderfolgende Werte `X` das Prädikat `binary_pred(X, value)` gelten. Im Beispiel wird erst gesucht, ob der Wert 4 mindestens 3-mal nacheinander auftritt. Dann wird gesucht, ob es mindestens 3-mal nacheinander Werte gibt, die größer als 4 sind:

**Listing 24.53:** Aufeinanderfolgende Werte suchen, die einer Bedingung genügen

```
// Auszug aus cppbuch/k24/finden/search_n.cpp
#include<algorithm>
#include<vector>
#include<iostream>
#include<showSequence.h>
using namespace std;

int main() {
   vector<int> folge(12);
   for(size_t i = 0; i < folge.size(); ++i) {
      folge[i] = i;       // 0 1 2 3 4 5 6 7 8 9 10 11
   }
   folge[6] = folge[5] = folge[4]; // aufeinanderfolgende gleiche Werte
   cout << "Folge = ";
   showSequence(folge);
```

```
  int wert = 4;
  int wieoft = 3;
  auto position = search_n(folge.begin(), folge.end(), wieoft, wert);

  if(position != folge.end()) {
    cout << wert << " wurde " << wieoft << "–mal nacheinander ab Position "
         << (position – folge.begin()) << " gefunden." << endl;
  }
  else {
    cout << wert << " wurde nicht " << wieoft
         << "–mal nacheinander gefunden." << endl;
  }
  position = search_n(folge.begin(), folge.end(), wieoft, wert, greater<int>());
  if(position != folge.end()) {
    cout << "Ab Position " << (position – folge.begin())
         << " wurden " << wieoft << "–mal nacheinander Werte größer als "
         << wert << " gefunden." << endl;
  }
  else {
    cout << wieoft << "–mal nacheinander Werte größer als "
         << wert << " sind nicht vorhanden." << endl;
  }
}
```

### 24.5.6 Binäre Suche

Alle Algorithmen dieses Abschnitts sind Variationen der binären Suche. Wenn ein wahlfreier Zugriff mit einem Random-Access-Iterator auf eine sortierte Folge mit $n$ Elementen möglich ist, ist die binäre Suche sehr schnell. Es werden maximal $1+\log_2 n$ Zugriffe benötigt, um das Element zu finden, oder festzustellen, dass es nicht vorhanden ist. Falls ein wahlfreier Zugriff nicht möglich ist wie zum Beispiel bei einer Liste, in der man sich von Element zu Element hangeln muss, um ein bestimmtes Element zu finden, ist die Zugriffszeit von der Ordnung $O(n)$. Die C++-Standardbibliothek stellt vier Algorithmen bereit, die im Zusammenhang mit dem Suchen und Einfügen in sortierte Folgen sinnvoll sind und sich algorithmisch sehr ähneln:

#### binary_search

```
template<class ForwardIterator, class T>
bool binary_search(ForwardIterator first, ForwardIterator last,
                   const T& value);
```

```
template<class ForwardIterator, class T, class Compare>
bool binary_search(ForwardIterator first, ForwardIterator last,
                   const T& value, Compare comp);
```

Dies ist die eigentliche binäre Suche. Die Funktion gibt `true` zurück, falls der Wert `value` gefunden wird. In der ersten Variante wird der Operator < benutzt, indem die Beziehung

```
(!(*i < value) && !(value < *i))
```

betrachtet wird (Äquivalenz). `i` ist ein Iterator im Bereich `[first, last)`. Falls ein Bereich mit einem Vergleichsobjekt `comp` sortiert wurde, muss dies bei der Suche berücksichtigt werden (zweite Variante). Es wird entsprechend

```
(!comp(*i, value) && !comp(value, *i))
```

ausgewertet. Die Äquivalenzbeziehung entspricht nicht in jedem Fall der Gleichheit. Beim Vergleich ganzer Zahlen mit dem <-Operator fallen beide Begriffe zusammen. Aber: Im Duden werden Umlaute bei der Sortierung wie Selbstlaute behandelt. Die Wörter Mucke und Mücke stehen beide vor dem Wort mucken. Obwohl ungleich, sind sie doch äquivalent bezüglich der Sortierung. Ein Beispiel für `binary_search()` wird nach Vorstellung der nächsten drei Algorithmen gezeigt.

### lower_bound

Dieser Algorithmus findet die erste Stelle, an der ein Wert `value` eingefügt werden kann, ohne die Sortierung zu stören. Der zurückgegebene Iterator, er sei hier `i` genannt, zeigt auf diese Stelle, sodass ein Einfügen ohne weitere Suchvorgänge mit `insert(i, value)` möglich ist. Für alle Iteratoren `j` im Bereich `[first, i)` gilt, dass `*j < value` ist bzw. `comp(*j, value) == true`. Die Prototypen sind:

```
template<class ForwardIterator, class T>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                            const T& value);

template<class ForwardIterator, class T, class Compare>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                            const T& value, Compare comp);
```

### upper_bound

Dieser Algorithmus findet die *letzte* Stelle, an der ein Wert `value` eingefügt werden kann, ohne die Sortierung zu stören. Der zurückgegebene Iterator `i` zeigt auf diese Stelle, sodass ein schnelles Einfügen mit `insert(i, value)` möglich ist. Die Prototypen sind:

```
template<class ForwardIterator, class T>
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
                            const T& value);

template<class ForwardIterator, class T, class Compare>
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
                            const T& value, Compare comp);
```

### equal_range

Dieser Algorithmus ermittelt den größtmöglichen Bereich, innerhalb dessen an jeder beliebigen Stelle ein Wert `value` eingefügt werden kann, ohne die Sortierung zu stören. Bezüglich der Sortierung enthält dieser Bereich also äquivalente Werte. Die Elemente `p.first` und `p.second` des zurückgegebenen Iteratorpaars, hier `p` genannt, begrenzen den Bereich. Für jeden Iterator `k`, der die Bedingung `p.first` $\leq$ `k < p.second` erfüllt, ist schnelles Einfügen mit `insert(k, value)` möglich. Die Prototypen sind:

```
template<class ForwardIterator, class T>
pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first, ForwardIterator last,
          const T& value);
```

```
template<class ForwardIterator, class T, class Compare>
pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first, ForwardIterator last,
          const T& value, Compare comp);
```

Die beschriebenen Algorithmen werden anhand eines Beispielprogramms demonstriert, wobei `upper_bound()` wegen seiner Ähnlichkeit mit `lower_bound()` nicht aufgeführt ist. Die Sortierung des Containers muss gewährleistet sein, weil alle Algorithmen dieses Abschnitts dies voraussetzen.

**Listing 24.54:** binary_search() und Verwandte

```
// cppbuch/k24/finden/binarysearch.cpp
#include<algorithm>
#include<list>
#include<string>
#include<showSequence.h>
using namespace std;

int main() {
   list<string> staedte;
   staedte.push_front("Bremen");
   staedte.push_front("Paris");
   staedte.push_front("Mailand");
   staedte.push_front("Hamburg");
   staedte.sort();                  // Wichtige Vorbedingung!
   showSequence(staedte);
   string stadt;
   cout << "Welche Stadt eintragen/suchen? ";
   cin >> stadt;
   if(binary_search(staedte.begin(), staedte.end(), stadt)) {
      cout << stadt << " existiert.\n";
   }
   else {
      cout << stadt << " existiert noch nicht.\n";
   }
   // an der richtigen Stelle einfügen
   cout << stadt << " wird eingefügt:\n";
   auto i = lower_bound(staedte.begin(), staedte.end(), stadt);
   staedte.insert(i, stadt);
   showSequence(staedte);
   // Bereich gleicher Werte
   auto p = equal_range(staedte.begin(), staedte.end(), stadt);

   // Die zwei Iteratoren des Paars p begrenzen den Bereich, in dem stadt vorkommt.
   auto n = distance(p.first, p.second);
   cout << stadt << " ist " << n << " mal in der Liste enthalten\n";
}
```

# 24.6 Mengenoperationen auf sortierten Strukturen

Die folgenden Algorithmen beschreiben die grundlegenden Mengenoperationen wie Vereinigung, Durchschnitt usw. auf *sortierten* Strukturen. In der C++-Standardbibliothek basiert auch die Klasse `set` auf sortierten Strukturen (siehe Abschnitt 28.3.3). Die Komplexität der Algorithmen ist $O(N_1 + N_2)$, wobei $N_1$ und $N_2$ die jeweilige Anzahl der Elemente der beteiligten Mengen sind. Mengenoperationen auf sortierten Strukturen sind nur unter bestimmten Bedingungen sinnvoll, und es sind Randbedingungen zu beachten.

- Standard-Container aus Kapitel 28: `vector`, `list`, `deque`
  - Der Ergebniscontainer bietet ausreichend Platz. Nachteil: Nach dem Ende der Ergebnissequenz stehen noch alte Werte im Container, falls der Platz mehr als genau ausreichend ist.
  - Der Output-Iterator darf nicht identisch mit `v1.begin()` oder `v2.begin()` sein. `v1` und `v2` sind die zu verknüpfenden Mengen.
  - Der Ergebniscontainer ist leer. In diesem Fall ist ein Insert-Iterator als Output-Iterator zu nehmen.
- Assoziative Container: `set`, `map`
  Grundsätzlich ist ein Insert-Iterator zu nehmen. Der Inhalt eines Elements darf nicht direkt, das heißt über eine Referenz auf das Element, geändert werden. So würde sich ein nicht einfügender Output-Iterator verhalten, und die Sortierung innerhalb des Containers und damit seine Integrität würde verletzt.

## 24.6.1 Teilmengenrelation

Die Funktion `includes` gibt an, ob jedes Element einer zweiten sortierten Struktur $S_2$ in der ersten Struktur $S_1$ enthalten ist, also eine Teilmenge der ersten ist. Der Rückgabewert ist `true`, falls $S_2 \subseteq S_1$ gilt, ansonsten `false`. Die Prototypen sind:

```
template<class InputIterator1, class InputIterator2>
bool includes(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2);
```

```
template<class InputIterator1, class InputIterator2, class Compare>
bool includes(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2,
              Compare comp);
```

Das folgende Beispiel initialisiert einige `set`-Objekte als sortierte Strukturen. Man kann an deren Stelle natürlich auch schlichte Vektoren nehmen, vorausgesetzt, sie sind sortiert. Weil das Beispiel in den weiteren Abschnitten aufgegriffen wird, enthält es bereits hier mehr, als für `includes()` notwendig ist.

**Listing 24.55:** Operationen auf Mengen

```
// Auszug aus cppbuch/k24/mengen/set_algorithmen.cpp
#include<algorithm>
```

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

int main () {
    int v1[] = {1, 2, 3, 4};
    int v2[] = {0, 1, 2, 3, 4, 5, 7, 99, 13};
    int v3[] = {-2, 5, 12, 7, 33};
    // Weiter unten benötigte Sets mit den Vektorinhalten initialisieren.
    // Voreingestelltes Vergleichsobjekt: less<int>()
    // (implizite automatische Sortierung)
    // sizeof v/sizeof *v1 ist die Anzahl der Elemente in v
    set<int> s1(v1, v1 + sizeof v1/sizeof *v1);
    set<int> s2(v2, v2 + sizeof v2/sizeof *v2);
    set<int> s3(v3, v3 + sizeof v3/sizeof *v3);
    set<int> result;        // leere Menge (s1, s2, s3 wie oben)
    if(includes(s2.begin(), s2.end(), s1.begin(), s1.end())) {
        showSequence(s1);           // 1 2 3 4
        cout << " ist eine Teilmenge von ";
        showSequence(s2);           // 0 1 2 3 4 5 7 99
    }  // Fortsetzung nächster Abschnitt
```

### 24.6.2 Vereinigung

Die Funktion `set_union` bildet eine sortierte Struktur, in der alle Elemente enthalten sind, die in wenigstens einer von zwei anderen sortierten Strukturen $S_1$ und $S_2$ vorkommen. Es wird die Vereinigung beider Strukturen gebildet:

$$S = S_1 \cup S_2$$

Voraussetzung ist, dass die aufnehmende Struktur genügend Platz bietet oder dass sie leer ist und ein Insert-Iterator als Output-Iterator verwendet wird. Die Prototypen sind:

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result);
```

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result, Compare comp);
```

Die Ergebnismenge `result` (siehe unten) ist anfangs leer. Im nachfolgenden Beispiel muss der Output-Iterator ein Insert-Iterator sein. Dazu wird die Funktion `inserter()`, die auf Seite 813 beschrieben ist, in der Parameterliste aufgeführt. Sie gibt einen Insert-Iterator zurück. Nur `result.begin()` als Output-Iterator zu verwenden, führt zu Fehlern.

```
    set<int> result;    // leere Menge (s1, s2, s3 seien wie oben)
    set_union(s1.begin(), s1.end(), s3.begin(), s3.end(),
              inserter(result, result.begin()));
    showSequence(s1);             // 1 2 3 4
    cout << " vereinigt mit ";
    showSequence(s3);             // -2 5 7 12 33
```

```
cout << " ergibt ";
showSequence(result);          // -2 1 2 3 4 5 7 12 33
```

### 24.6.3 Schnittmenge

Die Funktion `set_intersection` bildet eine sortierte Struktur, in der alle Elemente enthalten sind, die sowohl in der einen als auch in der anderen von zwei sortierten Strukturen $S_1$ und $S_2$ vorkommen. Es wird die Schnittmenge oder der Durchschnitt beider Strukturen gebildet:

$S = S_1 \cap S_2$

Die Prototypen sind:

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1,
                                InputIterator2 first2, InputIterator2 last2,
                                OutputIterator result);
```

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1,
                                InputIterator2 first2, InputIterator2 last2,
                                OutputIterator result,
                                Compare comp);
```

Um die alten Ergebnisse zu löschen, wird `clear()` aufgerufen. Andernfalls würden sie mit ausgegeben.

```
// Fortsetzung des Beispiels von oben
// Zuerst Menge leeren:
result.clear();
// Durchschnitt bilden:
set_intersection(s2.begin(), s2.end(),
                 s3.begin(), s3.end(),
                 inserter(result, result.begin()));
showSequence(s2);                 // 0 1 2 3 4 5 7 99
cout << " geschnitten mit ";
showSequence(s3);                 // -2 5 7 12 33
cout << " ergibt ";
showSequence(result);             // 5 7
```

### 24.6.4 Differenz

Die Funktion `set_difference` bildet eine sortierte Struktur, in der alle Elemente enthalten sind, die in der ersten Struktur $S_1$, aber nicht in einer zweiten sortierten Struktur $S_2$ vorkommen. Es wird die Differenz $S_1 - S_2$ beider Strukturen gebildet, auch als $S_1 \backslash S_2$ geschrieben. Die Prototypen sind:

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1,
                              InputIterator2 first2, InputIterator2 last2,
                              OutputIterator result);
```

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1,
                              InputIterator2 first2, InputIterator2 last2,
                              OutputIterator result,
                              Compare comp);
```

Das Beispiel folgt dem obigen Muster:

```
result.clear();
set_difference(s2.begin(), s2.end(),
               s1.begin(), s1.end(),
               inserter(result, result.begin()));
showSequence(s2);                // 0 1 2 3 4 5 7 99
cout << " minus ";
showSequence(s1);                // 1 2 3 4
cout << " ergibt ";
showSequence(result);            // 0 5 7 99
```

### 24.6.5 Symmetrische Differenz

Die Funktion `set_symmetric_difference` bildet eine sortierte Struktur, in der alle Elemente enthalten sind, die entweder in der ersten Struktur $S_1$ oder in einer zweiten sortierten Struktur $S_2$ vorkommen, aber nicht in beiden. Es wird die symmetrische Differenz beider Strukturen gebildet, auch als Exklusiv-Oder bezeichnet. Mit den vorangegangenen Operationen kann die symmetrische Differenz ausgedrückt werden:

$$S = (S_1 - S_2) \cup (S_2 - S_1)$$

oder

$$S = (S_1 \cup S_2) - (S_2 \cap S_1)$$

Die Prototypen sind:

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_symmetric_difference(
                    InputIterator1 first1, InputIterator1 last1,
                    InputIterator2 first2, InputIterator2 last2,
                    OutputIterator result);
```

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_symmetric_difference(
                    InputIterator1 first1, InputIterator1 last1,
                    InputIterator2 first2, InputIterator2 last2,
                    OutputIterator result, Compare comp);
```

Das letzte Beispiel dieser Art zeigt die symmetrische Differenz:

```
result.clear();
set_symmetric_difference(s2.begin(), s2.end(),
                         s3.begin(), s3.end(),
                         inserter(result, result.begin()));
showSequence(s2);                // 0 1 2 3 4 5 7 99
cout << " exklusiv oder ";
```

```
    showSequence(s3);                // -2 5 7 12 33
    cout << " ergibt ";
    showSequence(result);            // -2 0 1 2 3 4 12 33 99
} // Ende von cppbuch/k24/mengen/set_algorithmen.cpp
```



## 24.7 Heap-Algorithmen

Die in Abschnitt 28.2.7 beschriebene Priority-Queue basiert auf einem binären Heap (englisch für Haufen oder Halde). Vor der Beschreibung der Heap-Algorithmen seien kurz die wichtigsten Eigenschaften der Datenstruktur Heap (nicht zu verwechseln mit dem ebenfalls Heap genannten Freispeicher) charakterisiert:

- Die $N$ Elemente eines Heaps liegen in einem kontinuierlichen Array auf den Positionen 0 bis $N - 1$. Es wird vorausgesetzt, dass ein wahlfreier Zugriff möglich ist (Random-Access-Iterator).
- Die Art der Anordnung der Elemente im Array entspricht einem vollständigen binären Baum, bei dem alle Ebenen mit Elementen besetzt sind. Die einzig mögliche Ausnahme bildet die unterste Ebene, in der alle Elemente auf der linken Seite erscheinen. Abbildung 24.1 zeigt die Array-Repräsentation eines Heaps $H$ mit 14 Elementen, wobei die Zahlen in den Kreisen die Array-Indizes darstellen (*nicht* die Elementwerte). Das Element $H[0]$ ist also stets die Wurzel, und jedes Element $H[j]; (j > 0)$ hat einen Elternknoten $H[(j - 1)/2]$.

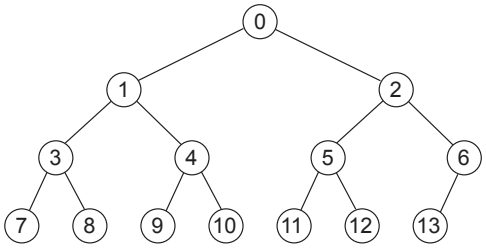


**Abbildung 24.1:** Array-Repräsentation eines Heaps (Zahl = Array-Index)

- Jedem Element $H[j]$ ist eine Priorität zugeordnet, die größer oder gleich der Priorität der Kindknoten $H[2j + 1]$ und $H[2j + 2]$ ist. Hier und im Folgenden sei zur Vereinfachung angenommen, dass große Zahlen hohe Prioritäten bedeuten. Im Allgemeinen kann es auch umgekehrt sein oder es können gänzlich andere Kriterien die Priorität bestimmen. Abbildung 24.2 zeigt beispielhafte *Elementwerte* eines Heaps: $H[0]$ ist gleich 98 usw.

Beachten Sie, dass der Heap nicht vollständig sortiert ist, sondern dass es nur auf die Prioritätsrelation zwischen Eltern- und zugehörigen Kindknoten ankommt.

Ein Array $H$ mit $N$ Elementen ist genau dann ein Heap, wenn $H[(j - 1)/2] \geq H[j]$ für $1 \leq j < N$ gilt. Daraus folgt automatisch, dass $H[0]$ das größte Element ist. Eine Priority-

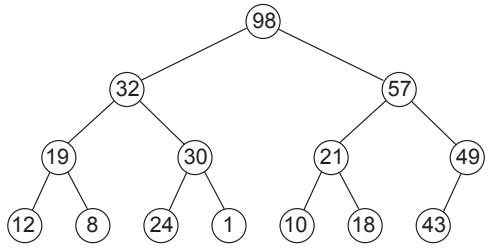


**Abbildung 24.2:** Array-Repräsentation eines Heaps (Zahl = Elementwert)

Queue entnimmt einfach immer das oberste Element eines Heaps. Anschließend wird er restrukturiert, das heißt, das nächstgrößte Element wandert an die Spitze. Bezogen auf die Abbildungen 24.1 und 24.2 wäre dies das Element Nr. 2 mit dem Wert 57.

Die C++-Standardbibliothek bietet vier Heap-Algorithmen an, die auf alle Container, auf die mit Random-Access-Iteratoren zugegriffen werden kann, anwendbar sind.

- `pop_heap()` entfernt das Element mit der höchsten Priorität.
- `push_heap()` fügt ein Element einem vorhandenen Heap hinzu.
- `make_heap()` arrangiert alle Elemente innerhalb eines Bereichs, sodass dieser Bereich einen Heap darstellt.
- `sort_heap()` verwandelt einen Heap in eine sortierte Folge.

Diese Algorithmen müssen keine Einzelheiten über die Container wissen. Ihnen werden lediglich zwei Iteratoren übergeben, die den zu bearbeitenden Bereich markieren. Zwar ist `less<T>` als Prioritätskriterium vorgegeben, aber vielleicht wird ein anderes Kriterium gewünscht. Daher gibt es für jeden Algorithmus eine überladene Variante, welche die Übergabe eines Vergleichsobjekts erlaubt.

### 24.7.1 pop_heap

Die Funktion `pop_heap()` entnimmt ein Element aus einem Heap. Der Bereich `[first, last)` sei dabei ein gültiger Heap. Die Prototypen sind:

```
template<class RandomAccessIterator>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last);
```

```
template<class RandomAccessIterator, class Compare>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
              Compare comp);
```

Die Entnahme besteht nur darin, dass der Wert mit der höchsten Priorität, der an der Stelle `first` steht, mit dem Wert an der Stelle (`last -1`) vertauscht wird. Anschließend wird der Bereich `[first, last-1)` in einen Heap verwandelt. Die Komplexität von `pop_heap()` ist *O*(log(*last* – *first*)). Anwendung für einen Vektor `v`:

**Listing 24.56:** Heap-Operationen

```
// Auszug aus cppbuch/k24/heap/heap.cpp
// gültigen Heap erzeugen
make_heap(v.begin(), v.end()); // Seite 691
// Die beiden Zahlen mit der höchsten Priorität anzeigen und entnehmen:
```

```
vector<int>::iterator last = v.end();
cout << *v.begin() << endl;
pop_heap(v.begin(), last--);
cout << *v.begin() << endl;
pop_heap(v.begin(), last--);
```

**Hinweis**

Bitte beachten Sie, dass nicht mehr `v.end()` das Heap-Ende anzeigt, sondern der Iterator `last`. Der Bereich dazwischen ist bezüglich der Heap-Eigenschaften von `v` *undefiniert*.

## 24.7.2 push_heap

Die Funktion `push_heap()` fügt ein Element einem vorhandenen Heap hinzu. Wie die Prototypen zeigen, werden der Funktion nur zwei Iteratoren und gegebenenfalls ein Vergleichsobjekt übergeben. Das einzufügende Element tritt hier nicht auf:

```
template<class RandomAccessIterator>
void push_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void push_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
```

Es muss die Vorbedingung gelten, dass der Bereich `[first, last-1)` ein gültiger Heap ist. `push_heap()` kümmert sich nicht selbst um den einzutragenden Wert. An die Stelle (`last`) wird deshalb *vorher* der auf den Heap abzulegende Wert eingetragen. Der anschließende Aufruf von `push_heap(first, ++last)` sorgt dafür, dass nach dem Aufruf der Bereich `[first, last)` ein Heap ist. Die Funktion ist etwas umständlich zu bedienen, aber sie ist auch nur als Hilfsfunktion gedacht und sehr schnell. Die Komplexität von `push_heap()` ist $O(\log(last - first))$. In den Beispiel-Heap werden nun zwei Zahlen wie beschrieben eingefügt (das vorhergehende Beispiel wird fortgesetzt):

```
// eine »wichtige Zahl« (99) eintragen
*last = 99;
push_heap(v.begin(), ++last);
// eine »unwichtige Zahl« (-1) eintragen
*last = -1;
push_heap(v.begin(), ++last);
```

Beim Einfügen muss beachtet werden, dass `last` nicht über `v.end()` hinausläuft. Durch die Tatsache, dass bei der Entnahme immer der Wert mit der höchsten Priorität an die Spitze gesetzt wird, ist die Ausgabe sortiert:

```
// Ausgabe und Entfernen aller Zahlen der Priorität nach:
while(last != v.begin()) {
   cout << *v.begin() << ' ';
   pop_heap(v.begin(), last--);
}
```

### 24.7.3 make_heap

`make_heap()` sorgt dafür, dass die Heap-Bedingung für alle Elemente innerhalb eines Bereichs gilt. Die Prototypen sind:

```
template<class RandomAccessIterator>
void make_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void make_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
```

Die Komplexität ist proportional zur Anzahl der Elemente zwischen `first` und `last`. Das Beispiel von oben zeigt die Anwendung auf einem Vektor als Container:

```
make_heap(v.begin(), v.end());
```

### 24.7.4 sort_heap

`sort_heap()` verwandelt einen Heap in eine sortierte Sequenz. Die Sortierung ist nicht stabil, die Komplexität ist $O(N \log N)$, wenn $N$ die Anzahl der zu sortierenden Elemente ist. Die Prototypen sind:

```
template<class RandomAccessIterator>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
```

Die Sequenz ist *aufsteigend* sortiert. Damit ist gemeint, dass die Elemente hoher Priorität *an das Ende* der Sequenz kommen:

```
// neuen gültigen Heap aus allen Elementen erzeugen
make_heap(v.begin(), v.end());
// und sortieren
sort_heap(v.begin(), v.end());
```

**Übung**

**24.2** Gegeben sei die Template-Klasse `Heap` mit den folgenden Schnittstellen:

```
template<typename T, class Compare = std::less<T> >
class Heap {
 public:
   Heap(const Compare& cmp = Compare());
   void push(const T& t);
   void pop();
   const T& top() const;
   bool empty() const;
   vector<T> toSortedVector() const;
};
```

Die letzte Methode gibt den Heap-Inhalt als sortierten Vektor zurück. Implementieren Sie die Klasse unter Verwendung einiger Algorithmen des Abschnitts 24.7. Testen Sie die

Klasse, indem Sie das `priority_queue`-Objekt in der Lösung von Aufgabe 28.2 durch ein gleichnamiges `Heap`-Objekt ersetzen.

### 24.7.5 is_heap

`is_heap()` gibt zurück, ob ein Bereich den Heap-Kriterien genügt. Gegebenenfalls kann ein Vergleichsobjekt übergeben werden.

```
template<class RandomAccessIterator>
bool is_heap(RandomAccessIterator first,
             RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
bool is_heap(RandomAccessIterator first,
             RandomAccessIterator last,
             Compare comp);
```

Ergänzend dazu gibt es den Algorithmus `is_heap_until(first, last)`, der den letzten Iterator im Bereich `[first, last)` zurückgibt, bis zu dem der Bereich als Heap angesehen werden kann.

## 24.8 Vergleich von Containern auch ungleichen Typs

### 24.8.1 Unterschiedliche Elemente finden

`mismatch()` überprüft zwei Container auf Übereinstimmung ihres Inhalts, wobei eine Variante ein binäres Prädikat benutzt. Die Prototypen sind:

```
template<class InputIterator1, class InputIterator2>
pair<InputIterator1, InputIterator2> mismatch(
              InputIterator1 first1,
              InputIterator1 last1,
              InputIterator2 first2);

template<class InputIterator1, class InputIterator2, class BinaryPredicate>
pair<InputIterator1, InputIterator2> mismatch(
              InputIterator1 first1,
              InputIterator1 last1,
              InputIterator2 first2,
              BinaryPredicate binary_pred);
```

Der Algorithmus gibt ein Paar von Iteratoren zurück, die auf die erste Stelle der Nichtübereinstimmung in den jeweiligen korrespondierenden Containern zeigen. Falls beide Container übereinstimmen, ist der erste Iterator des zurückgegebenen Paares gleich `last1`. Die Container müssen nicht vom selben Typ sein. In dem Beispiel wird ein Vektor `v` mit einem Set `s` verglichen:

**Listing 24.57:** Prüfung auf (Nicht-)Übereinstimmung

```
// cppbuch/k24/vergleich/mismatch.cpp
#include<algorithm>
#include<vector>
#include<set>
#include<showSequence.h>
using namespace std;

int main() {
   vector<int> v(8);
   for(size_t i = 0; i < v.size(); ++i) {
       v[i] = 2*i;                  // sortierte Folge
   }
   set<int> s(v.begin(), v.end()); // Set mit v initialisieren
   v[3] = 7;                       // Nichtübereinstimmung erzeugen
   showSequence(v);                // Anzeige
   showSequence(s);
   // Prüfung mit Iterator-Paar 'wo'
   // auto = pair<vector<int>::iterator, set<int>::iterator>
   auto wo = mismatch(v.begin(), v.end(), s.begin());
   if(wo.first == v.end()) {
      cout << "Inhalt der Container stimmt überein." << endl;
   }
   else {
     cout << "Der erste Unterschied (" << *wo.first << " != "
          << *wo.second          << ") wurde an Position "
          << (wo.first - v.begin()) << " gefunden." << endl;
   }
}
```

Eine indexartige Position ist in einem Set nicht definiert, deswegen ist ein Ausdruck der Art `(wo.second - s.begin())` ungültig. Zwar zeigt `wo.second` auf die Stelle der Nichtübereinstimmung in `s`, aber die Arithmetik ist nicht erlaubt. Wenn man die relative Nummer bezüglich des ersten Elements in `s` unbedingt benötigen sollte, kann man `distance()` verwenden.

Die Variante mit dem binären Prädikat ist geeignet, wenn es um nicht-exakte Übereinstimmung geht. Wenn etwa berechnete Resultate, die mit Rundungsfehlern behaftet sind, verglichen werden sollen, genügt es, wenn die Ergebnisse innerhalb einer gewissen Genauigkeit übereinstimmen. Im folgenden Beispiel werden zwei nicht genau gleiche Container als übereinstimmend betrachtet. Wenn der Schwellenwert 0.01 auf z.B. 0.005 abgesenkt wird, werden Unterschiede angezeigt.

**Listing 24.58:** Prüfung mit binärem Prädikat

```
// cppbuch/k24/vergleich/mismatch_b.cpp
#include<algorithm>
#include<vector>
#include<cmath> // fabs
#include<showSequence.h>
#include<Random.h>
using namespace std;
```

```
class VergleichMitToleranz {
public:
   VergleichMitToleranz(double e)
      : eps(e) {
   }
   bool operator()(double x, double y) {
      return fabs(x-y) < eps;
   }
private:
   double eps;
};

int main() {
   vector<double> v1(8), v2(8);
   Random zufall;
    for(size_t i = 0; i < v1.size(); ++i) {
       v1[i] = i + zufall(100)/10000.0;
       v2[i] = i + zufall(100)/10000.0;
    }
    showSequence(v1);              // Anzeige
    showSequence(v2);

    // Prüfung mit Iterator-Paar 'wo'
    // auto = pair<vector<double>::iterator, vector<double>::iterator>
    auto wo = mismatch(v1.begin(), v1.end(), v2.begin(),
                       VergleichMitToleranz(0.01));
    if(wo.first == v1.end()) {
       cout << "Inhalt der Container stimmt innerhalb der Toleranz überein."
            << endl;
    }
    else {
      cout << "Der erste Unterschied (" << *wo.first << " != "
           << *wo.second         << ") wurde an Position "
           << (wo.first - v1.begin()) << " gefunden." << endl;
    }
}
```

### 24.8.2 Prüfung auf gleiche Inhalte

`equal()` überprüft zwei Container auf Übereinstimmung ihres Inhalts, wobei eine Variante ein binäres Prädikat benutzt. Im Unterschied zu `mismatch()` wird jedoch kein Hinweis auf die Position gegeben. Wie am Rückgabetyp `bool` erkennbar, wird nur festgestellt, ob die Übereinstimmung besteht oder nicht. Die Prototypen sind:

```
template<class InputIterator1, class InputIterator2>
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2);
```

```
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2,
           BinaryPredicate binary_pred);
```

# 24.9 Rechnen mit komplexen Zahlen: Der C++-Standardtyp complex

Komplexe Zahlen werden in den Natur- und Ingenieur-Wissenschaften viel verwendet. Sie bestehen aus dem Real- und dem Imaginärteil, die beide von einem der möglichen Typen für reelle Zahlen sein können (`float`, `double`, `long double`). Der Standarddatentyp für komplexe Zahlen ist deshalb kein Grunddatentyp, sondern zusammengesetzt. Weil die Bestandteile eines `complex`-Objekts, also Real- und Imaginärteil, von einem der Typen `float`, `double` oder `long double` sein können, sind komplexe Zahlen als Klassen-Template realisiert. Die für den Compiler nötigen Informationen enthält der Header `<complex>`. Die Tabelle 24.1 enthält die Funktionen, die mit komplexen Zahlen verwendet werden können. Einige Beispiele zeigen, wie mit komplexen Zahlen in C++ gerechnet wird. Eine andere Möglichkeit zur Ermittlung von $\pi$ ist die Benutzung der Konstanten `M_PI` bzw. `M_PI_4` aus `<cmath>` (wird nicht immer zur Verfügung gestellt, weil nicht Teil des C-Standards[ISOC]).

**Listing 24.59:** Anwendung komplexer Zahlen

```
// cppbuch/k24/complex/complex.cpp
#include<iostream>
#include<complex>              // Header für komplexe Zahlen
#include<cmath>                // atan()
using namespace std;
// Anstatt double sind auch float und long double möglich.
int main() {
  complex<double> c1;          // komplexe Zahl 0.0 + 0.0i erzeugen
  complex<double> c2(1.2, 3.4); // (1.2 + 3.4i) erzeugen
  cout << c2 << endl;          // Standard-Ausgabeformat: (1.2,3.4)
  c1 += c2 + c1;               // beispielhafte Rechenoperationen
  c1 = c2 * 5.0;
  double re = c1.real();       // Realteil ermitteln
  cout << re << endl;          // und ausgeben
  cout << c1.imag() << endl;   // Imaginärteil direkt ausgeben
  // Beispiele mit Hilfsfunktionen
  complex<double> c3 = {1.0, 2.0}; // (1.0 + 2.0i) erzeugen, alternative Schreibweise
  c1 = conj(c3);               // konjugiert komplex: (1.0 - 2.0i)
  // Umrechnung aus Polarkoordinaten
  const double PI = 4.0 * atan(1.0); // π berechnen
  double betrag = 100.0;
  double phase = PI/4.0;           // π/4 = 45°
  c1 = polar(betrag, phase);

  // Umrechnung in Polarkoordinaten
  double rho  = abs(c1);           // Betrag ρ
  double theta = arg(c1);          // Winkel θ
  double nrm = norm(c1);           // Betragsquadrat
  cout << "Betrag = " << betrag << endl;
  cout << "rho    = " << rho   << endl;
```

**Tabelle 24.1:** Mathematische Funktionen für `complex`-Zahlen

| Schnittstelle | mathematische Entsprechung |
|---|---|
| `T real(const C& x)` | Realteil von $x$ (= `x.real()`) |
| `T imag(const C& x)` | Imaginärteil von $x$ (= `x.imag()`) |
| `T abs(const C& x)` | Betrag von $x$ |
| `C fabs(const C& x)` | Betrag von $x$ als komplexe Zahl |
| `T arg(const C& x)` | Phasenwinkel von $x$ in rad |
| `T norm(const C& x)` | Betragsquadrat von $x$ |
| `C conj(const C& x)` | zu $x$ konjugiert-komplexe Zahl |
| `C polar(const T& rho, const T& theta)` | Zahl entsprechend Betrag `rho` und Phase `theta` |
| `C acos(const C& x)` | acos $x$ |
| `C acosh(const C& x)` | acosh $x$ |
| `C asin(const C& x)` | asin $x$ |
| `C asinh(const C& x)` | asinh $x$ |
| `C atan(const C& x)` | atan $x$ |
| `C atanh(const C& x)` | atanh $x$ |
| `C cos(const C& x)` | cos $x$ |
| `C cosh(const C& x)` | cosh $x$ |
| `C exp(const C& x)` | $e^x$ |
| `C log(const C& x)` | ln $x$ |
| `C log10(const C& x)` | $\log_{10} x$ |
| `C pow(const C& x, int y)` | $x^y$ |
| `C pow(const C& x, const T& y)` | $x^y$ |
| `C pow(const T& x, const C& y)` | $x^y$ |
| `C pow(const C& x, const C& y)` | $x^y$ |
| `C sin(const C& x)` | sin $x$ |
| `C sinh(const C& x)` | sinh $x$ |
| `C sqrt(const C& x)` | $+\sqrt[2]{x}$ |
| `C tan(const C& x)` | tan $x$ |
| `C tanh(const C& x)` | tanh $x$ |

Abkürzungen:
`T` = einer der Typen `float`, `double` oder `long double`
`C` = `complex<T>`

```
  // Fortsetzung
  cout << "Norm  = " << nrm   << endl;
  cout << "Phase = " << phase << endl;
  cout << "theta  = " << theta
       << " = " << theta/PI*180. << " Grad" << endl;
  cout << "Komplexe Zahl eingeben. Erlaubte Formate z.B.:"
          "\n (1.78, -98.2)\n (1.78)\n 1.78\n:";
  cin >> c1;
  cout << "komplexe Zahl = " << c1 << endl;
}
```

Mit komplexen Zahlen kann wie mit reellen Zahlen gerechnet werden. Ausgenommen sind nur die Operatoren `++`, `-` und `%`. Auch die Prüfung auf Gleichheit (`==`) oder Ungleich-

heit (!=) ist möglich. Die anderen Vergleichsoperatoren ergeben bei komplexen Zahlen keinen Sinn.

**Übung**

**24.3** Schreiben Sie ein Programm zur Lösung der quadratischen Gleichung $x^2 + px + q = 0$, wobei Sie komplexe Zahlen für das Ergebnis verwenden. Die Lösungsformeln: $D = p^2/4 - q$, $x_1 = -p/2 + \sqrt{D}$, $x_2 = -p/2 - \sqrt{D}$. Eine komplexe Lösung ergibt sich, wenn $D < 0$ ist.

## 24.10 Schnelle zweidimensionale Matrix

In Abschnitt 5.7.3 wird eine Klasse für ein dynamisches 2-dimensionales Array beschrieben. Die folgende Matrix-Klasse ist eine Erweiterung mit überladenen Operatoren. Dabei ist wie bei der Standard-Vector-Klasse der Zugriff auf ein Matrix-Element über den Indexoperator `[][]` ungeprüft, der Zugriff über `at(i, j)` geprüft. Bei der Zuweisung müssen die Dimensionen links und rechts übereinstimmen.

Der Index-Operator `[z]` gibt einen Zeiger auf die Zeile `z` zurück, sodass `[z][s]` das Element `s` dieser Zeile zurückgibt. Das folgende Listing zeigt die Klasse. Im danach anschließenden Abschnitt geht es um die Optimierung mathematischer Operationen.

**Listing 24.60:** Dynamisches 2-dimensionales Array

```
// cppbuch/k24/array2d/array2d.h
// 2-dim. Array-Klasse mit zusammenhängendem (contiguous) Memory-Bereich
#ifndef ARRAY2D_H
#define ARRAY2D_H
#include<stdexcept>
#include<utility>
#include<algorithm>
#include<iostream>

template<typename T>
class Array2d {
 public:
   // Typedefs wegen STL-Konformität
   typedef T value_typ;
   typedef T& reference;
   typedef T* iterator;
   typedef const T* const_iterator;
   typedef T* pointer;
   typedef const T* const_pointer;
   typedef int difference_type;
   typedef unsigned int size_type;
```

```
   // Konstruktoren, Destruktor, Zuweisung
   Array2d(int zeilen, int spalten);
   Array2d(int zeilen, int spalten, const T& init);
   Array2d(const Array2d& a);
   Array2d(Array2d&& a);
   ~Array2d();
   Array2d& operator=(const Array2d& a);
   Array2d& operator=(Array2d&& a);

   // Elementfunktionen
   int getZeilen() const;
   int getSpalten() const;
   void init(const T& wert); // Alle Elemente mit wert initialisieren
   const T& at(int z, int s) const;
   T& at(int z, int s);
   const T* operator[](int z) const;
   T* operator[](int z);
   const T* data() const;

   // STL-entsprechende Funktionen
   size_t size() const { return zeilen*spalten;}
   iterator begin() { return arr;}
   iterator end() { return arr + zeilen*spalten;}
   const_iterator begin() const { return arr;}
   const_iterator end() const { return arr + size();}
   const_iterator cbegin() const { return arr;}
   const_iterator cend() const { return arr + size();}
   void swap(Array2d& a);

 private:
   void checkIndizes(int z, int s) const;
   void checkDimension(const Array2d<T>& a) const;
   int zeilen;
   int spalten;
   T* arr;
};

// relationale Operatoren
template<typename T>
bool operator==(const Array2d<T>& a, const Array2d<T>& b) {
   if(a.getZeilen() != b.getZeilen()) {
      return false;
   }
   if(a.getSpalten() != b.getSpalten()) {
      return false;
   }
   return std::equal(a.begin(), a.end(), b.begin());
}

template<typename T>
bool operator<(const Array2d<T>& a, const Array2d<T>& b) {
   return std::lexicographical_compare(a.begin(), a.end(), b.begin(), b.end());
}
```

```
// um aus == und < alle anderen relationalen Operatoren bei Bedarf automatisch zu bilden:
using namespace std::rel_ops;
// Implementierung, soweit nicht oben schon vorhanden:

template<typename T>
Array2d<T>::Array2d(int z, int s)
   : zeilen(z), spalten (s), arr(new T[z*s]) {
}

template<typename T>
Array2d<T>::Array2d(int z, int s, const T& wert)
   : zeilen(z), spalten (s), arr(new T[z*s]) {
   init(wert);
}

template<typename T>
Array2d<T>::Array2d(const Array2d<T>& a)
   : zeilen(a.getZeilen()), spalten (a.getSpalten()),
     arr(new T[zeilen*spalten]) {
   for(size_t i = 0; i < size(); ++i) {
      arr[i] = a.arr[i];
   }
}

template<typename T>
Array2d<T>::Array2d(Array2d<T>&& a) // bewegender (moving) Kopierkonstruktor
   : zeilen(0), spalten(0), arr(0) {
   swap(a);
}

template<typename T>
void Array2d<T>::swap(Array2d<T>& a) {
   std::swap(a.zeilen, zeilen);
   std::swap(a.spalten, spalten);
   std::swap(a.arr, arr);
}

template<typename T>
Array2d<T>::~Array2d() {
   delete [] arr;
}

template<typename T>
Array2d<T>& Array2d<T>::operator=(const Array2d<T>& a) {
   checkDimension(a);
   for(size_t i = 0; i < size(); ++i) {
      arr[i] = a.arr[i];
   }
   return *this;
}

template<typename T>
Array2d<T>& Array2d<T>::operator=(Array2d<T>&& a) {
```

```
   swap(a);
   return *this;
}

template<typename T>
inline int Array2d<T>::getZeilen() const {
   return zeilen;
}

template<typename T>
inline int Array2d<T>::getSpalten() const {
   return spalten;
}

template<typename T>
void Array2d<T>::init(const T& wert) {
   for(size_t i = 0; i < size(); ++i) {
      arr[i] = wert;
   }
}

template<typename T>
inline const T* Array2d<T>::operator[](int z) const {
   return &arr[z*spalten];
}

template<typename T>
inline T* Array2d<T>::operator[](int z) {
   return &arr[z*spalten];
}

template<typename T>
inline void Array2d<T>::checkIndizes(int z, int s) const {
   if(z < 0 || z >= zeilen)
      throw std::range_error(
               "Array2d: Zeile ausserhalb des erlaubten Bereichs");
   if(s < 0 || s >= spalten)
      throw std::range_error(
               "Array2d: Spalte ausserhalb des erlaubten Bereichs");
}

template<typename T>
inline void Array2d<T>::checkDimension(const Array2d<T>& a) const {
   if(zeilen != a.getZeilen() || spalten != a.getSpalten())
      throw std::range_error("Array2d: ungleiche Zeilen-/Spaltenanzahl");
}

template<typename T>
inline const T& Array2d<T>::at(int z, int s) const {
   checkIndizes(z, s);
   return arr[z*spalten + s];
}
```

```
template<typename T>
inline T& Array2d<T>::at(int z, int s) {
   checkIndizes(z, s);
   return arr[z*spalten + s];
}

template<typename T>
inline const T* Array2d<T>::data() const {
   return arr;
}
#endif
```

Durch die Verwendung von inline-Methoden ist die Klasse genauso schnell wie ein dynamisches C-Array. Sie ist aber leichter zu benutzen; insbesondere muss man sich keine Gedanken über das Zusammenspiel von `new` und `delete` machen und hat auf Wunsch einen indexgeprüften Zugriff. Durch den bewegenden Kopierkonstruktor und den bewegenden Zuweisungsoperator werden temporäre Objekte vermieden.

Mehr zu bewegenden Kopierkonstruktoren usw. lesen Sie auf den Seiten 591 ff.

Durch die Einbindung des Namensraums `std::rel_ops` wird erreicht, dass die relationalen Operatoren `!= > >= <=` bei Bedarf aus den Operatoren `==` und `<` erzeugt werden.

Mehr zu `rel_ops` lesen Sie auf Seite 747.

### 24.10.1 Optimierung mathematischer Array-Operationen

**Hinweis**

Das Verständnis dieses Abschnitts setzt die Kenntnis der folgenden Kapitel voraus:
Abschnitt 6.5: Templates mit variabler Parameterzahl
Kapitel 22: Performance, Wert- und Referenzsemantik
Abschnitt 27.4: Tupel

Die Klasse `Array2d` kann mit mathematischen Operationen versehen werden. Wenn dies mit überladenen Operatoren geschieht, sind Anweisungen wie die folgenden möglich:

```
// Addition
Array2d<int> a1(2, 3);
Array2d<int> a2(2, 3);
Array2d<int> a3(2, 3);
Array2d<int> ergebnis(2, 3);
// Berechnung der Array-Inhalte weggelassen
ergebnis = a1 + a2 + a3;           // operator+()
```

Das Problem aus Performance-Sicht ist die Erzeugung temporärer Objekte, wie schon in Kapitel 22 erläutert. Eine naiver `operator+()` würde auf der rechten Seite der Zuweisung ein temporäres Objekt erzeugen. Die Summation geschieht mit einer Schleife über alle Matrix-Elemente. Dasselbe geschieht im zweiten Schritt. Anschließend wird der Inhalt des Ergebnisses der Variablen `ergebnis` wieder mit einer Schleife über alle Elemente zuge-

wiesen. Zuletzt wird das temporäre Objekt vernichtet. Abschnitt 22.2 (Optimierung durch Referenzsemantik für R-Werte) zeigt, wie einige der temporären Objekte vermieden werden können. Noch besser wäre es jedoch, wenn das Entstehen aller temporären Objekte verhindert werden könnte und nur eine einzige Schleife notwendig wäre, etwa

```
for(size_t i = 0; i < ergebnis.getZeilen(); ++i) {
  for(size_t j = 0; j < ergebnis.getSpalten(); ++j) {
     ergebnis[i][j] = a1[i][j] + a2[i][j] + a3[i][j];
  }
}
```

Sinn des Überladens ist es, solche Schleifen zugunsten einfacher Anweisungen wie `ergebnis = a1 + a2 + a3;` zu vermeiden, wobei die Anzahl der Operanden auf der rechten Seite beliebig sein kann. Am Ende des Kapitels 22 wird auf die Möglichkeit hingewiesen, das Problem mit Expression Templates zu lösen. Eine andere Möglichkeit bieten Tupel in der Kombination mit Templates variabler Parameterzahl (*variadic templates*). Dieser Ansatz wird im Folgenden erläutert. Es zeigt sich, dass er genauso gute Ergebnisse wie Expression Templates liefert. Die Grundidee ist:

- Es gibt eine Template-Klasse `Summand`. In einem Objekt dieser Klasse ist nur ein Verweis auf ein Array und ein möglicherweise vorhandener multiplikativer Faktor gespeichert.

**Listing 24.61:** Klasse Summand

```
// Auszug aus cppbuch/k24/array2dmath/array2d.h
template<typename T>
struct Summand {
   Summand(const T& faktor, const Array2d<T>& arr)
      : f(faktor), a(arr) {
   }

   Summand(const Array2d<T>& arr)
      : f(T(1)), a(arr) {
   }

   T getWert(size_t index) const {
       return *(a.data()+index) * f;
   }

   size_t getZeilen() const {
      return a.getZeilen();
   }

   size_t getSpalten() const {
      return a.getSpalten();
   }

   Summand<T>& mult(const T& faktor) {
      f *= faktor;
      return *this;
   }
```

```
  T f;
  const Array2d<T>& a;
};
```

- `operator+(const Array2d&, const Array2d&)` berechnet nichts, sondern erzeugt zwei Verweisobjekte des Typs `Summand` und gibt ein Tupel, das diese zwei Objekte enthält, zurück:

```
// Auszug aus cppbuch/k24/array2dmath/array2d.h
template<typename T>
tuple<Summand<T>, Summand<T> >
inline operator+(const Array2d<T>& x, const Array2d<T>& y) {
  return tuple<Summand<T>, Summand<T>>(Summand<T>(x), Summand<T>(y));
}
```

- Bei mehr als zwei Summanden wird dem zurückgegebenen Tupel ein `Summand`-Objekt mit einem Verweis auf das nächste zu addierende `Array2d` hinzugefügt.

```
ergebnis = Tupel(a1, a2) + Summand(a3); // Pseudocode!
```

Der zugehörige Plus-Operator erweitert einfach das Tupel um `a3`. Da dieses Schema für beliebig viele Summanden gilt, kommen hier die *variadic templates* ins Spiel:

```
template<typename T, typename... Args>
tuple<Summand<T>, Args...>
inline operator+(const tuple<Args...>& t, const Array2d<T>& y) {
  return t + Summand<T>(y);
}
```

Die letzte Zeile ruft den folgenden Operator auf, der mit der Funktion `tuple_cat()` das Tupel verlängert:

```
template<typename T, typename... Args>
tuple<Summand<T>, Args...>
inline operator+(const tuple<Args...>& t, const Summand<T>& y) {
  return tuple_cat(tuple<Summand<T>>(y), t);
}
```

- Bei den vorangehenden Schritten ist entscheidend, dass keinerlei `new`-Operation und keinerlei Schleife vorkommen. Der letzte Schritt ist die Zuweisung des letzten Tupels, das Verweise auf alle beteiligten Arrays enthält:

```
ergebnis = Tupel(a1, a2, a3, ...); // Pseudocode!
```

Dieser Zuweisungsoperator erledigt die eigentlichen Additionen in einer einzigen Schleife, indem er auf ein Array-Element die Summe der entsprechenden Array-Elemente des Tupels addiert:

```
Array2d& operator=(const tuple<Args...>& t) {
     for(size_t i = 0; i < size(); ++i) {
        arr[i] = Summe<sizeof...(Args), T, Args...>::ergebnis(i, t);
     }
  return *this;
}
```

Die Klasse `Summe` unten wertet die entsprechenden Array-Elemente *zur Compilierzeit* aus. Das bedeutet, dass es keine `for`-Schleife dafür geben kann, die zur Laufzeit ausgeführt wird, sondern dass der Compiler die Addition rekursiv bewerkstelligt, ähnlich wie in Abschnitt 6.5 beschrieben.

```
// Berechnung über alle Summanden
template<size_t N, typename T, typename... Args>
struct Summe {
   static T ergebnis(size_t index, const tuple<Args...>& t) {
      return get<N-1>(t).getWert(index)
           + Summe<N-1, T, Args...>::ergebnis(index, t);
   }
};

// partielle Spezialisierung zum Rekursionsabbruch
template<typename T, typename... Args>
struct Summe<1, T, Args...> {
   static T ergebnis(size_t index, const tuple<Args...>& t) {
      return get<0>(t).getWert(index);
   }
};
```

`get<M>(t)` gibt das Tupel-Element Nr. `M` zurück, also ein `Summand`-Objekt. `getWert(size_t index)` ist eine Methode der Klasse `Summand`, die das Array-Element an der Stelle `index` zurückgibt. Die Funktion `ergebnis<N, T, Args>()` gibt also das entsprechende Array-Element plus die Summe der restlichen Elemente zurück. Falls `N = 0` ist, gibt es keine restlichen Elemente mehr. Der Compiler instanziiert die Funktion `ergebnis()` für alle rekursiv ermittelten Typen.

Der Vorteil dieses Ansatzes ist derselbe wie der Vorteil von Expression Templates, dass nämlich die eigentliche Berechnung möglichst spät geschieht, und dass der Compiler bei der Auswertung eines Ausdrucks nur Stack-Objekte erzeugt. Durch Inlining und Optimierung wird der Code dann auf das Wesentliche reduziert.

Die Datei *cppbuch/k24/array2dmath/array2d.h* wurde noch um die Multiplikation mit Faktoren und die Matrixmultiplikation erweitert, sodass auch Ausdrücke wie `ergebnis = -a1 + 2.0 * a2 + 3.0 * a3*a4;` möglich sind. Alle Operanden sind 2-dimensionale Matrizen. Die Datei wird aus Platzgründen hier nicht abgedruckt. Abschließend folgt ein Vergleich mit anderen Matrix-Klassen. Es wurden 500x500-Matrizen addiert. Jede Matrix hatte somit 250.000 Elemente. Die Compilation wurde mit dem Schalter `-O3` für maximale Optimierung durchgeführt.

**Tabelle 24.2:** Testergebnisse für die Addition von 500x500-Matrizen

| Bibliothek | Klasse | Anweisung | Dauer [ms] | Hinweis |
|---|---|---|---|---|
| | `Array2d<double>` | `aerg = -a1 + 2.0 *a2 + a3;` | 1,5 | 1. |
| Standard-C++ | `valarray<double>` | `verg = -v1 + 2.0 *v2 + v3;` | 1,5 | 2. |
| Boost | `matrix<double>` | `aerg = -a1 + 2.0 *a2 + a3;` | 2,5 | 3. |
| Blitz | `Array<double,2>` | `aerg = -a1 + 2.0 *a2 + a3;` | 1,5 | 4. |

Die Testergebnisse beziehen sich auf meinen PC; auf Ihrem Rechner können die Zahlen andere sein. Es ist zu sehen, dass die Boost-Matrix nicht ganz so gut optimiert ist. Den

gesamten Test finden Sie im Verzeichnis *cppbuch/k24/array2dmath/performancetest*. Bitte lesen Sie zuerst die dortige Datei *README.txt*, wenn Sie die Tests durchführen wollen. Die Hinweise zu der Tabelle 24.2:

1. Keine Bibliothek, weil es sich um die oben vorgestellte Klasse handelt.
2. Ein `valarray` ist keine Matrix. Um dennoch einen Vergleich durchzuführen, wurden `valarray`-Objekte mit je 250.000 Elementen verwendet.
3. Die Matrix ist in der Datei *boost/numeric/ublas/matrix.hpp* definiert. Die zugrundeliegende Boost-Version ist 1.45.
4. Mit der Blitz-Bibliothek (Quelle: *http://oonumerics.org/blitz/*) sind multi-dimensionale Arrays möglich. Die Zahl 2 im Array-Typ gibt die Dimension an. Die zugrundeliegende Blitz-Version ist 0.9 mit einem Patch, der die Compilation mit G++ ab Version 4.3 ermöglicht (siehe *cppbuch/k24/array2dmath/performancetest/blitz/readme.txt*).

**Tabelle 24.3:** Testergebnisse für die Multiplikation von 500x500-Matrizen

| Bibliothek | Klasse | Anweisung | Dauer [s] | Hinweis |
|---|---|---|---|---|
| | `Array2d<double>` | `aerg = a1 * a2;` | 0,65 | 1. |
| Boost | `matrix<double>` | `aerg = prod(a2, a1);;` | 0,65 | 2. |
| Blitz | `Array<double,2>` | `aerg = sum(a1(i,k) * a2(k,j),k);` | 0,65 | 3. |

Die Tabelle 24.3 zeigt die Ergebnisse für die Matrix-Multiplikation. Alle Verfahren sind offensichtlich gleich gut optimiert. Zu den Hinweisnummern der letzten Spalte:

1. Nur `Array2d` hat überladene Operatoren, und dies ohne Geschwindigkeitsverlust.
2. Das Überladen von Operatoren ist nicht realisiert, sodass ein Funktionsaufruf verwendet wird.
3. Eine Matrix-Multiplikation wird mit Blitz *nicht* mit `a1 * a2`, sondern durch die o.a. Anweisung erreicht. Dabei sind `i`, `j` und `k` mithilfe vorgegebener Typen (`firstIndex` usw.) definiert, wie in *cppbuch/k24/array2dmath/performancetest/blitz/multblitz.cpp* und unten zu sehen. Näheres bitte ich der Blitz-Dokumentation zu entnehmen.

```
firstIndex i;    // Platzhalter für den ersten Index
secondIndex j;   // Platzhalter für den zweiten Index
thirdIndex k;    // Platzhalter für den dritten Index
```

Die Klasse `Array2d` demonstriert gut die Wirksamkeit des Konzepts »Variadic Templates«. Die Klasse implementiert noch jedoch nicht alle typischen mathematischen Operationen wie etwa Klammerrechnung (`2.0*(a1+a2)`) oder die Division von Matrizen.

## 24.11 Singleton

Gelegentlich kommt es vor, dass man nur genau *eine* Instanz einer Klasse braucht, zum Beispiel zum Verwalten von Ressourcen, die definitiv nur einmal vorkommen. Das kann ein Objekt zur Ansprache mehrerer Drucker sein oder eine Log-Datei, die an vielen Stellen eines Programms benutzt wird. Das Singleton-Entwurfsmuster [Gamma] ist dafür

geeignet. Es stellt sicher, dass es exakt eine Instanz gibt, auf die über eine eindeutige Schnittstelle zugegriffen wird. Wegen der einfachen Anforderungen, die aber nicht ganz einfach zu implementieren sind, zählt dieses Entwurfsmuster zu den am häufigsten diskutierten.

### 24.11.1 Implementierung mit einem Zeiger

Eine einfache Implementierung mit einem Zeiger könnte so aussehen:

```
class Singleton {
public:
  static Singleton* getInstance() {
     if(!pInstance) { // statisches Attribut
        pInstance = new Singleton;
     }
     return pInstance;
  }
  // Restliche Methoden weggelassen
private:
  static Singleton* pInstance;
  Singleton();                          // direkte Konstruktion verbieten
  Singleton(const Singleton&);          // Kopie verbieten
  Singleton& operator=(const Singleton&); // sinnlose Zuweisung verbieten
};
```

In einer zugehörigen Implementationsdatei *Singleton.cpp* wird der Zeiger mit `0` initialisiert: `Singleton* Singleton::pInstance = 0;` Es ist sichergestellt, dass die Erzeugung mit `new` exakt einmal geschieht. Konstruktor und Kopierkonstruktor sind `private`, damit eine Objekterzeugung unter Umgehung von `getInstance()` nicht möglich ist. Weil es nur genau eine einzige Instanz gibt, ist der Gebrauch des Zuweisungsoperators sinnlos. Er ist deshalb ebenfalls als `private` deklariert.

#### Nachteile

Ein Problem ist die Entsorgung so eines Singleton-Objekts. Natürlich wird der Speicher ganz am Ende des Programms vom Betriebssystem wieder freigegeben. Der Destruktor wird aber nicht automatisch aufgerufen, sodass zwar kein Speicherleck entsteht, aber es kann sein, dass Ressourcen wie Dateien oder Netzwerkverbindungen, die vom Singleton vielleicht genutzt wurden, nicht freigegeben werden.

Abhilfe könnte man schaffen, indem explizit vor Ende des `main`-Programms `delete` auf den Zeiger angewendet wird – ein Weg, der dem C++-Prinzip RAII (siehe Glossar) widerspricht.

Ein weiterer Nachteil besteht darin, dass ein `delete` an einer beliebigen anderen Stelle nicht verhindert werden kann. Oder man macht den Destruktor privat – dann kann das Singleton-Objekt gar nicht gelöscht werden.

### 24.11.2 Implementierung mit einer Referenz

Das Problem der Ressourcenfreigabe ist besser lösbar, wenn ein ein statisches Attribut angelegt wird. Der Destruktor würde automatisch nach dem Ende von `main()` ausgeführt

und die Ressourcen freigeben. Von der Funktion `getInstance()` würde einfach eine Referenz auf das Attribut zurückgegeben:

```
class Singleton {
public:
   static Singleton& getInstance() {
      return instance;
   }
// Rest weggelassen
private:
  static Singleton instance;
  // Rest weggelassen
};
```

#### Nachteil

Ein Problem könnte entstehen, wenn in einer beliebigen anderen Übersetzungseinheit auf das Singleton zugegriffen würde, etwa

```
// irgendeineDatei.cpp
int ergebnis = Singleton::getInstance().eineFunktion();
```

Zwar werden globale und statische Objekte initialisiert, ehe die erste Zeile von `main()` ausgeführt wird. Weil C++ aber keine Reihenfolge bei der Initialisierung dieser Objekte festlegt, wenn diese in verschiedenen Übersetzungseinheiten vorliegen, könnte es sein, dass in der obigen Anweisung das `Singleton`-Objekt noch nicht existiert.

### 24.11.3 Meyers' Singleton

Meyers [ScM] hat die bisher beschriebenen Probleme auf elegante Art gelöst, indem er eine *funktionslokale* statische Variable nutzt. Ein funktionslokales statisches Objekt, das keine zur Compilationszeit bekannte Konstante ist, wird erst dann initialisiert, wenn die Funktion zum ersten Mal aufgerufen wird. Lokale statische Variable haben Sie schon in Abschnitt 3.1.3 (Seite 105) bei der »Funktion mit Gedächtnis« kennengelernt. Das Prinzip:

```
static Singleton& getInstance() {
   static Singleton instance;
   return instance;
}
```

Natürlich müssen auch hier Konstruktor, Kopierkonstruktor, Zuweisungsoperator und Destruktor `private` sein. Nun kann es sein, dass man dem Singleton bei der Initialisierung Parameter übergeben möchte, zum Beispiel den Namen einer Log-Datei – als Alternative zur Abfrage einer Umgebungsvariablen oder Konfigurationsdatei. Der Funktion `getInstance()` jedesmal den Parameter mitzugeben, ist unsicher. Eine `setParameter()`-Funktion würde nicht garantieren, dass die Initialisierung nur einmal erfolgt.

#### Erweiterung um Initialisierung mit Parametern

Aus diesem Grund wird im Folgenden ein Beispiel auf der Basis von [ScM] gezeigt, das die Initialisierung des Singletons vor der Verwendung garantiert, und auch, dass eine mehrfach versuchte Initialisierung nicht möglich ist. Die Initialisierungsfunktion muss

nur einmalig vor der sonstigen Verwendung aufgerufen werden. Das Singleton ist ein Objekt zum Schreiben in eine Log-Datei, deren Name `main()` übergeben wird. Zuerst sei eine mögliche Anwendung gezeigt:

**Listing 24.62:** Singleton-Anwendung

```
// cppbuch/k24/singleton/main.cpp
#include<iostream>
#include"LogSingleton.h"     // siehe unten
using namespace std;

int main(int argc, char* argv[]) {
   if(argc == 2) {
      try {
         // 1. initialisieren
         LogSingleton::initialize(argv[1]); // nicht Thread-sicher, siehe unten
         // 2. Instanz holen und verwenden:
         LogSingleton& s1 = LogSingleton::getInstance();
         cout << "s1.getLogfilename() = " << s1.getLogfilename() << endl;
         // Dieselbe Instanz unter anderem Namen:
         LogSingleton& s2 = LogSingleton::getInstance();
         cout << "Identische Objekte. Die Adressen sind dieselben:" << endl
              << &s1 << endl
              << &s2 << endl;
         // Verwendung des Singletons
         s1.log("Die erste Nachricht!");
         s2.log("Die zweite Nachricht!");
      }
      catch (const logic_error& e) { // Erklärung siehe LogSingleton.h unten
         cout << "Programmierfehler: " << e.what() << endl;
      }
      catch (const runtime_error& e) { // Erklärung siehe LogSingleton.h unten
         cout << "Runtime-Fehler: " << e.what() << endl;
      }
   }
   else {
      cout << "Gebrauch: " << argv[0] << " Log-Dateiname" << endl;
   }
}  // Hier wird ggf. der LogSingleton-Destruktor aufgerufen.
```

Es folgt die `LogSingleton`-Klasse. In der Funktion `getInstance()` wird geprüft, ob es sich um den ersten Aufruf handelt. Es kein Problem, wenn das Singleton dann noch nicht initialisiert ist – die Funktion `initialize()` benötigt einmalig die Instanz, um die Attribute wie gewünscht zu belegen. Anschließend wird das Flag `initialisiert` gesetzt, sodass weitere Aufrufe von `getInstance()` möglich sind.

**Listing 24.63:** Logging-Klasse als Singleton

```
// cppbuch/k24/singleton/LogSingleton.h
#ifndef LOGSINGLETON_H
#define LOGSINGLETON_H
#include<iostream>
#include<fstream>
#include<stdexcept>
```

```
class LogSingleton {
public:
    static LogSingleton& getInstance() { // liefert stets Referenz auf dasselbe Objekt
        static LogSingleton single;
        static bool ersterAufruf = true;
        if (!ersterAufruf && !single.initialisiert) {
            throw std::logic_error("LogSingleton ist nicht initialisiert!");
        }
        ersterAufruf = false;
        return single;
    }

    // initialize() muss exakt einmal vor getInstance() aufgerufen werden.
    static void initialize(const std::string& lfn) {
        LogSingleton& s = getInstance();
        if (s.initialisiert) {
            throw std::logic_error("mehrfache Initialisierung ist "
                    "nicht möglich!");
        }
        s.logfilename = lfn; // Initialisierung
        s.datei.open(lfn);
        if (!s.datei.good()) {
            throw std::runtime_error(lfn + " kann nicht geöffnet werden.");
        }
        s.initialisiert = true;
    }

    void log(const std::string& message) { // normale Benutzung
        datei << message << std::endl;
    }

    const std::string& getLogfilename() const {
        return logfilename;
    }
private:
    bool initialisiert;
    std::string logfilename;
    std::ofstream datei;
    LogSingleton() : initialisiert(false) { // Konstruktor. Externen Aufruf verbieten
    }
    LogSingleton(const LogSingleton&);  // Kopierkonstruktor verbieten
    void operator=(const LogSingleton&); // sinnlose Zuweisung verbieten
    virtual ~LogSingleton() {           // externen Aufruf verbieten
       // Ausgabe nur zur Demonstration
       std::cout << "LogSingleton-Destruktor aufgerufen" << std::endl;
       // Ressourcen freigeben. Dies ist hier nicht unbedingt notwendig, weil
       datei.close(); // der ofstream-Destruktor dies automatisch erledigt hätte.
    }
};
#endif
```

**Hinweis**

Das vorgestellte Singleton ist nicht Thread-safe. Das heißt, ein gleichzeitiger Zugriff von verschiedenen Threads aus kann zu Inkonsistenzen führen. Auch gibt es je nach Anwendungsfall verschiedene Singleton-Konzepte. Wenn Sie über darüber mehr wissen möchten, empfehle ich Ihnen [Alex].

# 24.12 Vermischtes

## 24.12.1 Erkennung eines Datums

Mit Hilfe eines regulären Ausdrucks soll überprüft werden, ob ein Datum der Form tt.mm.jjjj gültig ist. Dabei soll es auch möglich sein, nur eine Ziffer für den Tag bzw. den Monat einzugeben. Da hier nur die Gültigkeit interessiert und nicht, wo sich ein gültiges Datum innerhalb einer Zeichenkette befindet, genügt die Abfrage mit `regex_match()`. Ein passender regulärer Ausdruck (von vielen möglichen) ist `\d\d?\.\d\d?\.\d\d\d\d`.

**Listing 24.64:** Syntaktische Gültigkeit eines Datums

```
// cppbuch/k24/vermischtes/regex/datumsyntaxpruefen.cpp
#include <iostream>
#include<boost/regex.hpp>
#include<string>

bool datumok(const std::string& eingabe) {
   boost::regex datumregex("\\d\\d?\\.\\d\\d?\\.\\d\\d\\d\\d");
   return boost::regex_match(eingabe, datumregex);
}
using namespace std;

int main(int argc, char* argv[]) {
   if(2 != argc) {
      cout << "Gebrauch: datumpruefen.exe tt.mm.jjjj" << endl;
   }
   else {
      try {
         if(datumok(argv[1])) {
            cout << argv[1] << " ist ein gültiges Datum." << endl;
         }
         else {
            cout << argv[1] << " ist kein gültiges Datum." << endl;
         }
      } catch(boost::regex_error& re) {
         cerr << "Fehler: " << re.what() << endl;
      }
   }
}
```

### Syntax oder Semantik?

Am obigen Beispiel wird sofort ein Problem klar: 00.0.0000 ist ein ungültiges Datum, wird aber vom Programm als gültiges Datum interpretiert. Der Grund liegt darin, dass das Programm nur die vorgegebene Syntax prüft, aber die Semantik (Bedeutung) des Inhalts ignoriert. Was tun? Es gibt verschiedene Abstufungen der Prüfung:

1. Die einfachste Syntaxprüfung ist die oben dargestellte. Um die Gültigkeit zu prüfen, müssen Tag, Monat und Jahr aus dem String extrahiert werden, um dann die Gültigkeit zum Beispiel mit der Funktion `bool istGueltigesDatum(int t, int m, int j)` von Seite 336 zu prüfen.
2. Man kann aber auch in Erwägung ziehen, bereits bei der syntaktischen Analyse die Gültigkeit der Daten zu überprüfen. Für den Tag gibt es folgende Möglichkeiten:
   - Wenn der Tag einstellig ist, liegt die Ziffer im Bereich `[1-9]`.
   - Ist die erste Ziffer eine 0, liegt die zweite im Bereich `[1-9]`.
   - Ist die erste Ziffer eine 1 oder eine 2, liegt die zweite im Bereich `[0-9]`.
   - Ist die erste Ziffer eine 3, liegt die zweite im Bereich `[0-1]`.

   Der erste und der zweite Punkt werden mit dem regulären Ausdruck `0?[1-9]` realisiert, der dritte mit `(1|2)\d`, der vierte mit `3[01]`. Der reguläre Ausdruck für den Tag einschließlich des Punktes kann damit als `(0?[1-9])|((1|2)\d)|(3[01])\.` formuliert werden. Für den Monat gilt:
   - Wenn der Monat einstellig ist oder mit einer 0 beginnt, liegt die Ziffer vor dem Punkt im Bereich `[1-9]`.
   - Wenn der Monat zweistellig ist, kann die erste Ziffer nur eine 1 sein und die zweite liegt im Bereich `[0-2]`.

   Der reguläre Ausdruck für den Monat einschließlich des Punktes kann damit als `(0?[1-9])|(1[0-2])\.` formuliert werden.
3. Mit der vorstehenden Lösung ist aber noch nicht alles erreicht, denn manche Monate haben nur 30 Tage, und der Februar sogar nur 28 bzw. 29 in einem Schaltjahr. 30 oder 28 Tage lassen sich mit einigem Aufwand in der Syntax berücksichtigen, aber das Schaltjahr nur noch mit sehr großem, unverhältnismäßigem Aufwand.

Man sieht, dass der Aufwand recht hoch getrieben werden kann. Bei der Überprüfung von Dialogeingaben oder von einer Datei eingelesenen Informationen ist daher abzuwägen, wie der Gesamtaufwand aus Syntaxprüfung und inhaltlicher Prüfung minimiert werden kann.

**Empfehlung**

Einfache reguläre Ausdrücke bevorzugen, ergänzt durch eine inhaltliche Prüfung.

Diese Empfehlung entspricht dem obigen Punkt 1. Sie erspart Planungs- und Rechenzeitaufwand. Die Funktion `datumok()` des obigen Listings wird dazu ersetzt durch

**Listing 24.65:** Gültigkeit eines Datums

```
// Auszug aus cppbuch/k24/vermischtes/regex/datumpruefen.cpp

bool datumok(const std::string& eingabe) {
  boost::regex datumregex("\\d\\d?\\.\\d\\d?\\.\\d\\d\\d\\d");
```

```
   bool ergebnis = boost::regex_match(eingabe, datumregex);
   if(ergebnis) { // Syntax ok, Inhalt prüfen
      std::vector<std::string> v;
      boost::split(v, eingabe, boost::is_any_of("."));
      ergebnis = istGueltigesDatum(boost::lexical_cast<int>(v[0]),
                                   boost::lexical_cast<int>(v[1]),
                                   boost::lexical_cast<int>(v[2]));
   }
   return ergebnis;
}
```

Die Funktion `istGueltigesDatum()` wird von Seite 336 übernommen.

Mehr über `split()` und `lexical_cast` lesen Sie in den Abschnitten 24.1.1 und 24.1.2.

### 24.12.2 Erkennung einer IP-Adresse

Die Lösung dieses Problem lässt sich auf das vorhergehende zurückführen. Eine IP-Adresse hat vier durch einen Punkt getrennte Felder, wobei jedes Feld aus einer vorzeichenlosen Zahl mit maximal drei Ziffern besteht. Ein möglicher regulärer Ausdruck ist `\d\d?\d?\.\d\d?\d?\.\d\d?\d?\.\d\d?\d?` oder kürzer `(?:\d\d?\d?\.){3}\d\d?\d?`. Es gilt die Bedingung, dass jede Zahl höchstens den Wert 255 annehmen kann. Im Folgenden ist nur die überprüfende Funktion gezeigt:

**Listing 24.66:** Gültigkeit einer IP-Adresse

```
// Auszug aus cppbuch/k24/vermischtes/regex/IPadressepruefen.cpp
bool ipok(const std::string& eingabe) {
   boost::regex datumregex("(?:\\d\\d?\\d?\\.){3}\\d\\d?\\d?");
   bool ergebnis = boost::regex_match(eingabe, datumregex);
   if(ergebnis) { // Syntax ok, Inhalt prüfen
      std::vector<std::string> v;
      boost::split(v, eingabe, boost::is_any_of("."));
      for(size_t i = 0; i < v.size(); ++i) {
         ergebnis = ergebnis && boost::lexical_cast<int>(v[i]) < 256;
      }
   }
   return ergebnis;
}
```

### 24.12.3 Erzeugen von Zufallszahlen

[ISOC++] enthält einen großen Abschnitt über die Erzeugung von (Pseudo-)Zufallszahlen. Der Abschnitt ist für dieses Buch zu umfangreich und eher aus mathematischer Sicht als aus programmiersprachlicher Sicht interessant. Die Gleichverteilung und die Normalverteilung werden am häufigsten benutzt. Deshalb beschränke ich mich auf die folgenden drei Möglichkeiten:

1. Ein einfacher Zufallszahlengenerator `Random`, der auf den C-Funktionen `rand()` und `srand()` beruht.
2. Eine C++-Standardfunktion zur Erzeugung gleichverteilter Zufallszahlen.

3. Eine C++-Standardfunktion zur Erzeugung normalverteilter Zufallszahlen.

Die letzten beiden Möglichkeiten wie auch alle anderen neu in C++ aufgenommenen Zufallsverteilungen werden noch nicht von allen Compilern unterstützt. Der GNU C++ Compiler kennt sie erst ab Version 4.5. Allerdings gibt es eine funktionierende Boost-Library. Deswegen empfehle ich Ihnen bei Bedarf, die Library Boost.Random zu verwenden und die dazugehörige Dokumentation zu Rate zu ziehen.

### Reproduzierbarkeit

Für alle gezeigten Verfahren gilt, dass die berechneten Zufallswerte reproduzierbar, also pseudo-zufällig sind. Erneutes Starten eines Programms liefert dieselbe Zahlenfolge. Entscheidend dabei ist die Initialisierung des Zufallszahlengenerators mit den Funktionen `srand(unsigned seed)` im ersten und `seed(unsigned seed)` in den letzten beiden Fällen. Die Reproduzierbarkeit erleichtert das Testen. Um bei jedem Programmstart eine *andere* Zahlenfolge zu erreichen, kann man der Seed-Funktion die aktuelle Systemzeit mit `time(NULL)` oder einen davon abgeleiteten Wert übergeben.

### Zufallszahlen auf der Basis von rand()

`rand()` gibt eine etwa gleichverteilte Pseudozufallszahl zwischen 0 und `RAND_MAX` zurück, `srand(unsigned seed)` initialisiert den Zufallszahlengenerator. Im Beispiel wird die Einbindung als Funktionsobjekt gezeigt.

**Listing 24.67:** Zufallszahlengenerator

```
// cppbuch/include/Random.h
#ifndef RANDOM_H
#define RANDOM_H
#include<cstdlib> // rand(), srand() und RAND_MAX
class Random {
 public:
   Random(size_t g = 100)
      : grenze(g) {
   }
   void setSeed(size_t seed) {
      srand(seed);
   }
   // gibt eine Pseudo-Zufallszahl zwischen 0 und range-1 zurück
   size_t operator()(size_t range) {
       return (size_t)((double)rand()*range/(RAND_MAX+1.0));
   }
   // gibt eine Pseudo-Zufallszahl zwischen 0 und grenze-1 zurück
   size_t operator()() {
       return (size_t)((double)rand()*grenze/(RAND_MAX+1.0));
   }
 private:
   size_t grenze;
};#endif
```

Ein `Random`-Objekt wird mit dem Grenzwert initialisiert. Gegebenenfalls kann mit `setSeed()` ein Anfangswert für eine andere Zufallsfolge eingestellt werden. Die Typumwandlung in `double` bei der Berechnung vermeidet Überlaufprobleme. Durch Überladen des `()`-

Operators kann das Objekt wie eine Funktion benutzt werden: Falls die zweite, parameterlose Form des Funktionsoperators `operator()()` gewünscht wird, kann der Bereich im Konstruktor eingestellt werden.

**Listing 24.68:** Anwendung des Zufallszahlengenerators

```
// cppbuch/k24/zufallszahlen/random.cpp
#include<iostream>
#include"Random.h"
using namespace std;

int main() {
   Random zufall_1;
   for(int i=0; i < 5; ++i) { // 5 Zufallszahlen zwischen 0 und 999 ausgeben
      cout << zufall_1(1000) << endl;
    }

   Random zufall_2(1000);     // andere Möglichkeit
   for(int i=0; i < 5; ++i) {
      cout << zufall_2() << endl;
   }
}
```

Die zweite Form wird von manchen Algoritmen der STL benutzt, siehe zum Beispiel `generate_n()` in Abschnitt 24.3.3.

**Hinweis**

Weil `Random` in den Beispielen häufig benutzt wird, wurde die Datei *Random.h* in das *include*-Verzeichnis der Beispiele aufgenommen.

## Gleichverteilung

Die Erzeugung der Zufallszahlen wird von einem Generator übernommen. Es gibt nach [ISOC++] mehrere Generatoren; der gewählte ist `default_random_engine`. Der Gleichverteilung `uniform_int_distribution` wird der gewünschte Bereich übergeben. Die Zufallszahl wird durch Aufruf des Verteilungsobjekts erzeugt, wobei dem Funktionsobjekt der Generator als Parameter übergeben wird. Im folgenden Beispiel werden 100000 Zufallszahlen zwischen 10 und 20 erzeugt. Zur Kontrolle werden die Häufigkeiten für jeden Wert und der berechnete Mittelwert ausgegeben

**Listing 24.69:** Erzeugung gleichverteilter Zufallszahlen

```
// cppbuch/k24/zufallszahlen/verteilungen/gleichverteilung.cpp
#include <iostream>
#include <ctime>
#include <random>
#include <vector>
using namespace std;

int main() {
   const int MIN = 10;
```

```
  const int MAX = 20;
  static_assert(MAX > MIN, "MAX muss > MIN sein!");
  const size_t ITERATIONEN = 100000;
  vector<size_t> haeufigkeit(MAX-MIN+1, 0);
  uniform_int_distribution<> verteilung(MIN, MAX); // <>: Vorgabe ist int
  default_random_engine generator;
  //generator.seed(time(NULL)); // // ohne '//': keine Reproduzierbarkeit
  // Zufallszahl erzeugen, dabei die Häufigkeit hochzählen:
  for(size_t i = 0; i < ITERATIONEN; ++i) {
    ++haeufigkeit[verteilung(generator)-MIN];
  }
  // Häufigkeiten und Mittelwert ausgeben:
  int summe = 0.0;
  for(size_t i = 0; i < haeufigkeit.size(); ++i) {
    int wert = (int)i+MIN;
    summe += wert*haeufigkeit[i];
    cout << wert << ": " << haeufigkeit[i] << endl;
  }
  cout << "Mittelwert: " << (double)summe/ITERATIONEN << endl;
}
```

## Normalverteilung

Neben der Gleichverteilung ist die Normalverteilung am bekanntesten, charakterisiert durch die Gaußsche Glockenkurve. Sie wird insbesondere in der Messtechnik verwendet. So gehorchen die Abweichungen vom Sollmaß bei der industriellen Herstellung von Werkstücken der Normalverteilung. Das folgende Beispiel berechnet 1000000 normalverteilte Zufallszahlen, wobei zur Veranschaulichung das Ergebnis auf der Konsole ausgegeben wird (Abbildung 24.3).

**Listing 24.70:** Erzeugung normalverteilter Zufallszahlen

```
// cppbuch/k24/zufallszahlen/verteilungen/normalverteilung.cpp
#include <iostream>
#include <ctime>
#include <random>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
  const double MIN = -3.0; // darzustellender Bereich
  const double MAX = 3.0;
  const size_t INTERVALLE = 80; // Auflösung der Ausgabe (Spalten)
  const size_t ZEILEN = 20;  // Auflösung der Ausgabe (Zeilen)
  const double INTERVALLBREITE = (MAX-MIN)/INTERVALLE;
  const size_t ITERATIONEN = 1000000;

  vector<size_t> haeufigkeit(INTERVALLE, 0);
  normal_distribution<> gauss(0.0, 1.0); // Mittelwert, Standardabweichung
  default_random_engine generator;
  //generator.seed(time(NULL)); // ohne '//': keine Reproduzierbarkeit
  // Zufallszahl erzeugen, dabei die Häufigkeit hochzählen
```

```
  for(size_t i = 0; i < ITERATIONEN; ++i) {
     double wert = gauss(generator);
     if(wert > MIN && wert < MAX) {
        ++haeufigkeit[ (size_t)((wert - MIN)/INTERVALLBREITE)];
     } // else.. Außenbereiche ignorieren
  }
  // Darstellung der Glockenkurve auf der Konsole
  size_t max = *max_element(haeufigkeit.begin(), haeufigkeit.end());
  for(size_t i = 0; i < ZEILEN; ++i) {
     for(size_t j = 1; j < INTERVALLE; ++j) {
        double grenze = (double)haeufigkeit[j]*ZEILEN/max + 0.5;
        cout << (grenze <= (ZEILEN-i) ? ' ' : '*');
     }
     cout << endl;
  }
}
```

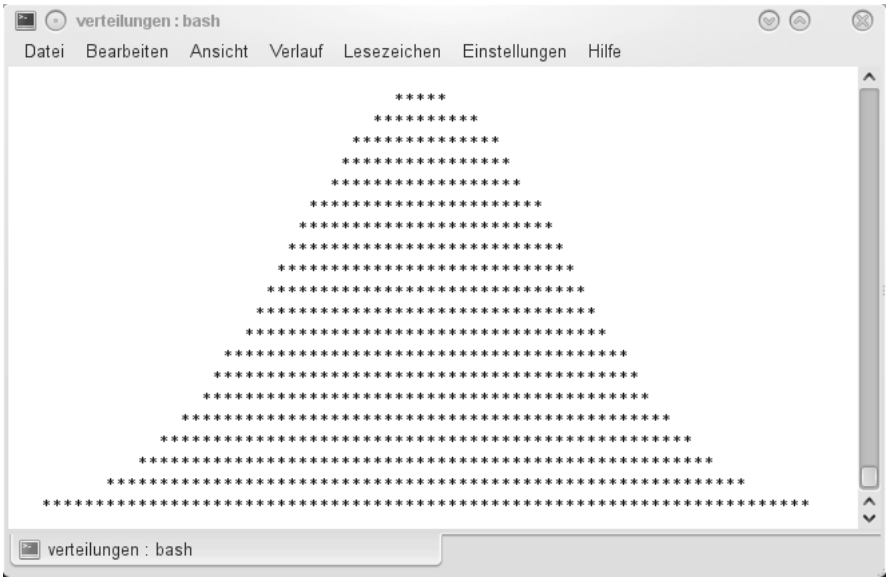


**Abbildung 24.3:** Glockenkurve auf der Konsole

## 24.12.4 for_each – Auf jedem Element eine Funktion ausführen

Der Algorithmus `for_each` bewirkt, dass auf jedem Element eines Containers eine Funktion ausgeführt wird. Die Deklaration ist:

```
template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f);
```

`f` kann sowohl eine Funktion (aber keine Methode einer Klasse) als auch ein Funktionsobjekt sein und wird nach Gebrauch zurückgegeben. Beispiel:

```
template<typename T>
void anzeige(const T& x) {
  std::cout << x << std::endl;
}
// Anzeige aller Elemente eines Containers c. Für die Elemente muss operator<<() definiert sein.
for_each(c.begin(), c.end(), anzeige);
```

### 24.12.5 Verschiedene Möglichkeiten, Container-Bereiche zu kopieren

Der Algorithmus `copy()` kopiert die Elemente eines Quellbereichs in den Zielbereich, wobei das Kopieren am Anfang oder am Ende der Bereiche (mit `copy_backward()`) beginnen kann. Falls der Zielbereich nicht überschrieben, sondern in ihn eingefügt werden soll, ist als Output-Iterator ein Iterator zum Einfügen (Insert-Iterator) zu nehmen. Der Algorithmus `copy()` ist immer dann zu nehmen, wenn Ziel- und Quellbereich sich nicht oder so überlappen, dass der Anfang des Quellbereichs im Zielbereich liegt. `result` muss anfangs auf den Anfang des Zielbereichs zeigen. Zur Verdeutlichung der Wirkungsweise sind hier ausnahmsweise nicht die Prototypen, sondern die vollständigen Definitionen gezeigt:

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
                    OutputIterator result) {
   while (first != last) {
      *result++ = *first++;
   }
   return result;
}
```

Der Algorithmus `copy_backward()` ist immer dann zu nehmen, wenn Ziel- und Quellbereich sich so überlappen, dass der Anfang des Zielbereichs im Quellbereich liegt. `result` muss anfangs auf das Ende des Zielbereichs zeigen.

```
template<class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2 copy_backward(BidirectionalIterator1 first,
                                     BidirectionalIterator1 last,
                                     BidirectionalIterator2 result) {
   while (first != last) {
      *--result = *--last;
   }
   return result;
}
```

Auch hier gilt wie allgemein in der C++-Standardbibliothek, dass `last` nicht die Position des letzten Elements bezeichnet, sondern die Position nach dem letzten Element. `result` darf niemals zwischen `first` und `last` liegen. Wie Abbildung 24.4 zeigt, sind drei Fälle zu berücksichtigen:

a) Die Bereiche sind voneinander vollständig getrennt. Die Bereiche können in demselben oder in verschiedenen Containern liegen. `result` zeigt auf den Beginn des Zielbereichs. `copy()` kopiert den Quellbereich beginnend mit `*first`. Zurückgegeben wird `result + (last - first)`, also die Position nach dem letzten Element des Zielbereichs.

b) Die Bereiche überlappen sich so, dass der Zielbereich *vor* dem Quellbereich beginnt. `result` zeigt auf den Beginn des Zielbereichs. `copy()` kopiert den Quellbereich begin-

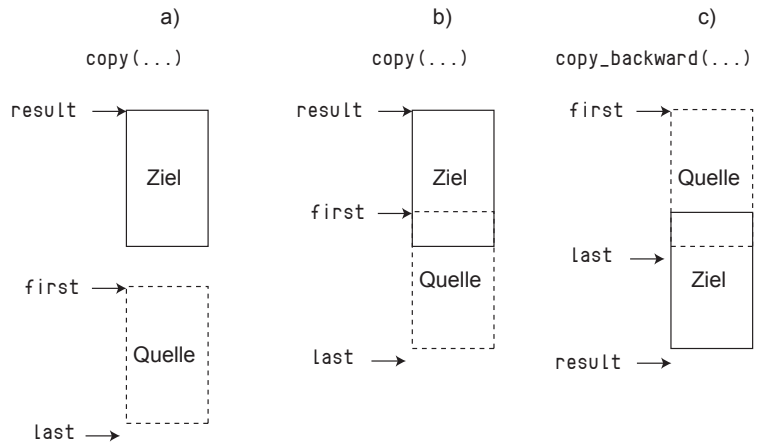


**Abbildung 24.4:** Kopieren ohne und mit Bereichsüberlappung

nend mit `*first`. Wie bei a) wird die Position nach dem letzten Element des Zielbereichs zurückgegeben.

c) Die Bereiche überlappen sich so, dass der Zielbereich *mitten im* Quellbereich beginnt. Um die Daten nicht zu zerstören, muss vom Ende her beginnend kopiert werden. `result` zeigt auf die Position direkt nach dem *Ende* des Zielbereichs. `copy_backward()` kopiert den Quellbereich, indem zuerst `*(--last)` an die Stelle `--result` kopiert wird. Hier wird `result - (last - first)` zurückgegeben, also die Position des zuletzt kopierten Elements im Zielbereich.

```
// Beispiele:
// v1 nach v2 kopieren
copy(v1.begin(), v1.end(), v2.begin());

// v1 nach v2 kopieren, dabei am Ende beginnen
copy_backward(v1.begin(), v1.end(), v2.end());

// v1 nach cout kopieren, Separator *
ostream_iterator<int> ausgabe(cout, "*");
copy(v1.begin(), v1.end(), ausgabe);
```

## Kopieren mit Bedingung

In Analogie zu anderen Algorithmen mit der Endung `_if` gibt es auch den Algorithmus `copy_if()`, der nur Elemente kopiert, die einer bestimmten Bedingung genügen. Der Prototyp ist

```
template<class InputIterator, class OutputIterator, class Predicate>
OutputIterator copy_if(InputIterator first, InputIterator last,
                       OutputIterator result, Predicate pred)
```

Im folgenden Beispiel werden alle Elemente des `container1` am Ende des `container2` eingefügt, sofern sie größer als 10 sind.

**Listing 24.71:** copy_if

```
// Auszug aus cppbuch/k24/vermischtes/copy/copy_if.cpp
int main() {
   typedef vector<int> Container;
   Container container1(20);
   iota(container1.begin(), container1.end(), 1);
   showSequence(container1);

   Container container2;  // leeren Container anlegen
   // alle Elemente > 10 am Ende einfügen:
   copy_if(container1.begin(), container1.end(),
          back_inserter(container2),
          bind(greater<int>(), _1, 10));
   showSequence(container2);
}
```

### Kopieren einer bestimmten Zahl von Elementen

Der Algorithmus `copy_n()` kopiert eine bestimmte Anzahl von Elementen. Der Prototyp ist

```
template<class InputIterator, class Size, class OutputIterator>
OutputIterator copy_n(InputIterator first, Size n, OutputIterator result);
```

**Listing 24.72:** copy_n

```
// Auszug aus cppbuch/k24/vermischtes/copy/copy_n.cpp
   vector<int> v1(20);
   iota(v1.begin(), v1.end(), 1);
   vector<int> v2(20, 0);
   // die ersten 10 Elemente kopieren
   copy_n(v1.begin(), 10, v2.begin());
}
```

## 24.12.6 Vertauschen von Elementen, Bereichen und Containern

Der Algorithmus `swap()` vertauscht Elemente von Containern oder Container selbst. Er tritt in vier Varianten auf:

- `swap()` vertauscht zwei einzelne Elemente. Die beiden Elemente können in verschiedenen, in demselben oder in keinem Container sein.

  ```
  template<typename T>
  void swap(T& a, T& b);
  ```

- `iter_swap()` nimmt zwei Iteratoren und vertauscht die dazugehörenden Elemente. Die beiden Iteratoren können zu verschiedenen oder zu demselben Container gehören.

  ```
  template<class ForwardIterator1,
           class ForwardIterator2>
  void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
  ```

```
// erstes und letztes Element per Iterator vertauschen:
vector<int>::iterator first = v.begin(),
                      last = v.end();
--last;
iter_swap(first, last);          // Tausch
```

- `swap_ranges()` vertauscht zwei Bereiche.

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 swap_ranges(ForwardIterator1 first1,
                             ForwardIterator1 last1,
                             ForwardIterator2 first2);
```

`first1` zeigt auf den Anfang des ersten Bereichs, `last1` auf die Position nach dem letzten Element des ersten Bereichs. Der Anfang des zweiten Bereichs wird durch `first2` gegeben. Die Anzahl der auszutauschenden Elemente wird durch die Größe des ersten Bereichs gegeben. Die Bereiche können in demselben Container sein, dürfen sich jedoch nicht überlappen. `swap_ranges()` gibt einen Iterator auf das Ende des zweiten Bereichs zurück.

```
// Vertauschen der beiden Hälften eines Vektors v
// mit einer geradzahligen Anzahl von Elementen
vector<double>::iterator Mitte = v.begin()+v.size()/2;
swap_ranges(v.begin(), Mitte, Mitte);
```

- `swap()` ist für diejenigen Container spezialisiert, die eine Methode `swap()` zum Vertauschen bereitstellen, also `deque`, `list`, `vector`, `set`, `map`, `multiset` und `multimap`. Diese Methoden sind sehr schnell ($O(1)$), weil nur Verwaltungsinformationen ausgetauscht werden[2]. `swap()` ruft intern die Methoden der Container auf.

### 24.12.7 Elemente transformieren

Wenn es darum geht, nicht nur etwas zu kopieren, sondern dabei gleich umzuwandeln, ist `transform()` der richtige Algorithmus. Die Umwandlung kann sich auf nur ein Element oder auf zwei Elemente gleichzeitig beziehen. Dementsprechend gibt es zwei überladene Formen:

```
template<class InputIterator, class OutputIterator, class UnaryOperation>
OutputIterator transform(InputIterator first, InputIterator last,
                         OutputIterator result, UnaryOperation op);
```

Hier wird auf jedes Element des Bereichs von `first` bis ausschließlich `last` die Operation `op` angewendet und das Ergebnis in den mit `result` beginnenden Bereich kopiert. `result` darf identisch mit `first` sein, wobei dann die Elemente durch die transformierten ersetzt werden. Der Rückgabewert ist ein Iterator auf die Position nach dem Ende des Zielbereichs. Im Beispiel

```
string s("ABC123");
transform(s.begin(), s.end(), s.begin(), tolower);
```

werden alle Großbuchstaben eines C-Strings in Kleinbuchstaben umgewandelt. Die Funktion `tolower()` ist im Header `<cctype>` deklariert.

[2] Ausnahme: `array`, vgl. Seite 765

```
template<class InputIterator1, class InputIterator2, class OutputIterator,
         class BinaryOperation>
OutputIterator transform(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, OutputIterator result,
                         BinaryOperation bin_op);
```

In der zweiten Form werden zwei Bereiche betrachtet. Der erste ist das Intervall `[first1, last1)`, der zweite das Intervall `[first2, first2 + last1 - first1)`, das heißt, der zweite Bereich ist genauso groß wie der erste. Die Operation `bin_op` nimmt jeweils ein Element aus jedem der zwei Bereiche und legt ihr Ergebnis in `result` ab. `result` darf identisch mit `first1` oder `first2` sein, wobei dann die Elemente durch die transformierten ersetzt werden. Der Rückgabewert ist ein Iterator auf die Position nach dem Ende des Zielbereichs. Damit sind komplexere Operationen möglich.

Im Beispiel werden jeweils zwei Strings verkettet. Dabei wird sowohl eine unäre Operation als Funktion eingesetzt wie auch eine binäre Operation als Funktor. Erstere wandelt alle Buchstaben eines C++-Strings in Großbuchstaben um und gibt den veränderten String zurück. Der Funktor verkettet zwei String-Objekte miteinander und fügt dabei das Wort »und« ein.

**Listing 24.73:** transform()

```
// cppbuch/k24/vermischtes/transform.cpp
#include<algorithm>
#include<cstddef>
#include<cctype>
#include<locale>
#include<string>
#include<vector>
#include<showSequence.h>

std::string upper_case(std::string s) { // unäre Operation als Funktion
   for(size_t i = 0; i < s.length(); ++i)
      s[i] = toupper(s[i]);
   return s;
}

class Verketten {            // binäre Operation als Funktor
public:
   std::string operator()(const std::string& a, const std::string& b) const {
      return a + " und " + b;
   }
};

using namespace std;

int main() {
   locale::global(locale("de_DE")); // falls Umlaute vorkommen
   const size_t ANZAHL = 3;
   vector<string> maedels(ANZAHL), jungs(ANZAHL), paare(ANZAHL);
   maedels[0] = "Annabella";
   maedels[1] = "Scheherazade";
   maedels[2] = "Julia";
```

```
   jungs[0] = "Nikolaus";
   jungs[1] = "Amadeus";
   jungs[2] = "Romeo";
   transform(jungs.begin(), jungs.end(),
             jungs.begin(),  // Ziel == Quelle
             upper_case);    // in Großbuchstaben wandeln
   transform(maedels.begin(), maedels.end(),
             jungs.begin(), paare.begin(), Verketten());
   showSequence(paare, "", "\n"); // gebildete Paare ausgeben
}
```

### 24.12.8 Ersetzen und Varianten

Der Algorithmus `replace()` ersetzt in einer Sequenz jeden vorkommenden Wert `old_value` durch `new_value`. Alternativ ist mit `replace_if()` eine bedingungsgesteuerte Ersetzung mit einem unären Prädikat möglich:

```
template<class ForwardIterator, class T>
void replace(ForwardIterator first, ForwardIterator last,
             const T& old_value, const T& new_value);
```

```
template<class ForwardIterator, class Predicate, class T>
void replace_if(ForwardIterator first, ForwardIterator last,
                Predicate pred, const T& new_value);
```

Erstmalig treten nun auch kopierende Varianten von Algorithmen auf, die sich im Namen durch ein hinzugefügtes `_copy` unterscheiden:

```
template<class InputIterator, class OutputIterator, class T>
OutputIterator replace_copy(InputIterator first, InputIterator last,
                            OutputIterator result, const T& old_value,
                            const T& new_value);
```

```
template<class InputIterator, class OutputIterator, class Predicate, class T>
OutputIterator replace_copy_if(Iterator first, Iterator last,
                               OutputIterator result, Predicate pred,
                               const T& new_value);
```

Das Beispiel zeigt alle vier Varianten:

**Listing 24.74:** replace() mit Varianten

```
// cppbuch/k24/vermischtes/replace.cpp
#include<algorithm>
#include<string>
#include<vector>
#include<showSequence.h>

// Unäres Prädikat als Funktor
class Zitrusfrucht {
   public:
      bool operator()(const std::string& a) const {
          return a == "Zitrone" || a == "Apfelsine" || a == "Limone";
      }
```

```
};
using namespace std;

int main() {
   vector<string> obstkorb(3), kiste(3);
   obstkorb[0] = "Apfel";
   obstkorb[1] = "Apfelsine";
   obstkorb[2] = "Zitrone";
   showSequence(obstkorb); // Apfel Apfelsine Zitrone
   cout << "replace: Apfel durch Quitte ersetzen:\n";
   replace(obstkorb.begin(), obstkorb.end(),
                    string("Apfel"), string("Quitte"));
   showSequence(obstkorb); // Quitte Apfelsine Zitrone
   cout << "replace_if: Zitrusfrüchte durch Pflaumen ersetzen:\n";
   replace_if(obstkorb.begin(), obstkorb.end(),
                    Zitrusfrucht(), string("Pflaume"));
   showSequence(obstkorb); // Quitte Pflaume Pflaume
   cout << "replace_copy: kopieren + ersetzen der Pflaumen durch Limonen:\n";
   replace_copy(obstkorb.begin(), obstkorb.end(),
          kiste.begin(), string("Pflaume"), string("Limone"));
   showSequence(kiste);     // Quitte Limone Limone
   cout << "replace_copy_if: kopieren und ersetzen "
           "der Zitrusfrüchte durch Tomaten:\n";
   replace_copy_if(kiste.begin(), kiste.end(),
          obstkorb.begin(), Zitrusfrucht(), string("Tomate"));
   showSequence(obstkorb); // Quitte Tomate Tomate
}
```

### 24.12.9 Elemente herausfiltern

Der Algorithmus entfernt alle Elemente aus einer Sequenz, die gleich einem Wert `value` sind beziehungsweise einem Prädikat `pred` genügen. Hier sind die Prototypen einschließlich der kopierenden Varianten aufgeführt:

```
template<class ForwardIterator, class T>
ForwardIterator remove(ForwardIterator first,
                   ForwardIterator last,
                   const T& value);
```

```
template<class ForwardIterator, class Predicate>
ForwardIterator remove_if(ForwardIterator first,
                      ForwardIterator last,
                      Predicate pred);
```

```
template<class InputIterator, class OutputIterator, class T>
OutputIterator remove_copy(InputIterator first,
                       InputIterator last,
                       OutputIterator result,
                       const T& value);
```

```
template<class InputIterator, class OutputIterator, class Predicate>
OutputIterator remove_copy_if(InputIterator first,
```

```
                          InputIterator last,
                          OutputIterator result,
                          Predicate pred);
```

»Entfernen eines Elements« bedeutet in Wirklichkeit, dass alle nachfolgenden Elemente um eine Position nach links rücken. Das letzte Element wird bei Entfernen eines einzigen Elements verdoppelt, weil eine Kopie davon dem vorhergehenden Platz zugewiesen wird. `remove()` gibt einen Iterator auf das nunmehr verkürzte Ende der Sequenz zurück.

Dabei ist zu beachten, dass die gesamte Länge der Sequenz sich nicht ändert! Es wird keine Neuordnung des Speicherplatzes vorgenommen. Falls nicht kopiert wird, enthält der Bereich zwischen dem zurückgegebenen Iterator und `last` nur noch bedeutungslos gewordene Elemente. Das Beispiel zeigt nur die nicht-kopierenden Varianten:

**Listing 24.75:** remove() mit Varianten

```
// cppbuch/k24/vermischtes/remove.cpp
#include<iostream>
#include<algorithm>
#include<vector>
#include<iterator>
#include<cstring>
#include<numeric>
#include<showSequence.h>

// Umlaute werden nicht berücksichtigt
bool istVokal(char c) {
   return std::strchr("aeiouyAEIOUY", c) != 0;
}
using namespace std;

int main() {
    vector<char> v(26);
    // Alphabet mit Kleinbuchstaben erzeugen
    iota(v.begin(), v.end(), 'a');
    showSequence(v, "\n", "");
    cout << "remove 't': ";
    vector<char>::iterator last = remove(v.begin(), v.end(), 't');
    // last = neues Ende nach der Verschiebung
    // v.end() bleibt unverändert!
    // Die Sequenz wird hier nicht mit showSequence() angezeigt,
    // weil nur die Elemente von begin() bis last signifikant sind
    ostream_iterator<char> ausgabe(cout, "");
    copy(v.begin(), last, ausgabe); // abcdefghijklmnopqrsuvwxyz (t fehlt)
    cout << endl;
    last = remove_if(v.begin(), last, istVokal);
    cout << "nur noch Konsonanten übrig: ";
    copy(v.begin(), last, ausgabe); // bcdfghjklmnpqrsvwxz
    cout << endl;
    cout << "Vollständige Sequenze bis end() einschließlich "
            " bedeutungsloser Elemente am Ende: ";
    showSequence(v, "\n", "");
}
```

### 24.12.10 Grenzwerte von Zahltypen

Im Header `<limits>` wird die Template-Klasse `numeric_limits` definiert. Sie hat Spezialisierungen für die ganzzahligen Grunddatentypen `bool`, `char`, `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, sowie für die Gleitkommazahltypen `float`, `double` und `long double`. Für diese Grunddatentypen beschreiben die Spezialisierungen verschiedene implementationsabhängige Funktionen und Eigenschaften, die alle `public` sind und von denen die wichtigsten in der Tabelle 24.4 aufgelistet sind. Eine Anwendung wird in Abschnitt 1.6.3 (Seite 47) gezeigt.

**Tabelle 24.4:** `<limits>`: Attribute und Funktionen (Auszug). NaN = not a number, `T` = Typ der Zahl

| Schnittstelle | Bedeutung |
|---|---|
| `bool is_specialized` | `true` nur für Grunddatentypen, für die eine Spezialisierung vorliegt, `false` für alle anderen |
| `T min()` | minimal möglicher Wert |
| `T max()` | maximal möglicher Wert |
| `int radix` | Zahlenbasis, normal 2 (Ausnahme z.B. BCD-Zahlen) |
| `int digits` | Ganzzahlen:<br>Anzahl der Bits (ohne Vorzeichen-Bit). Gleitkommazahlen: Anzahl der Bits in der Mantisse. Annahme: `radix` == 2 |
| `int digits10` | Anzahl signifikanter Dezimalziffern bei Gleitkommazahlen, zum Beispiel 6 bei `float`, 10 bei `double` |
| `bool is_signed` | `true` bei vorzeichenbehafteten Zahlen |
| `bool is_integer` | `true` bei Ganzzahltypen |
| `bool is_exact` | `true` bei exakten Zahlen, z.B. ganze oder rationale Zahlen – nicht aber Gleitkommazahlen |
| `T epsilon()` | kleinster positiver Wert $x$, für den die Maschine die Differenz zwischen 1.0 und (1.0 + $x$) noch unterscheidet |
| `T round_error()` | maximaler Rundungsfehler |
| `int min_exponent` | kleinster negativer Exponent für Gleitkommazahlen |
| `int min_exponent10` | kleinster negativer 10er-Exponent für Gleitkommazahlen ($\leq -37$) |
| `int max_exponent` | größtmöglicher Exponent für Gleitkommazahlen |
| `int max_exponent10` | größtmöglicher 10er-Exponent für Gleitkommazahlen ($\geq +37$) |
| `T infinity()` | Repräsentation von $+\infty$, falls vorhanden |
| `bool has_infinity` | `true`, falls der Zahltyp eine Repräsentation für $+\infty$ hat |
| `bool is_iec559` | `true`, falls der Zahltyp dem IEC 559(= IEEE 754)-Standard genügt. |
| `bool is_bounded` | `true` für alle Grunddatentypen, `false` wenn die Menge der darstellbaren Werte unbegrenzt ist, z.B. bei Typen mit beliebiger Genauigkeit. |
| `bool is_modulo` | `true`, falls bereichsüberschreitende Operationen wieder eine gültige Zahl ergeben. Z.B. ergibt die Addition einer Zahl auf die größtmögliche Integerzahl bei den meisten Maschinen wieder eine Integerzahl, die kleiner als die größtmögliche ist. Dies gilt im Allgemeinen nicht für Gleitkommazahlen. |
| `round_style` | Art der Rundung<br>Ganzzahlen: `round_toward_zero` (=0)<br>Gleitkommazahlen: `round_to_nearest` (=1) |

### 24.12.11 Minimum und Maximum

Die inline-Templates `min()` und `max()` geben jeweils das kleinere (bzw. das größere) von zwei Elementen zurück. Bei Gleichheit wird das erste Element zurückgegeben. Die Prototypen sind:

```
template<typename T>
const T& min(const T& a, const T& b);

template<typename T, class Compare>
const T& min(const T& a, const T& b, Compare comp);

template<typename T>
const T& max(const T& a, const T& b);

template<typename T, class Compare>
const T& max(const T& a, const T& b, Compare comp);
```

`minmax()` gibt ein `pair`-Objekt zurück, das den kleineren Wert an der Stelle `first` und den größeren an der Stelle `second` enthält:

```
template<typename T> pair<const T&, const T&>
minmax(const T& a, const T& b);

template<typename T, class Compare>
pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
```