25

Ein- und Ausgabe

Dieses Kapitel behandelt die folgenden Themen:

- Datei- und Verzeichnisoperationen
- Formatierte Daten schreiben
- Formatierte Daten lesen
- Array als Block lesen und schreiben

25.1 Datei- und Verzeichnisoperationen

Wie eine Datei kopiert werden kann, wird in Kapitel 2 beschrieben. Andere Dateioperationen, wie Löschen oder Umbenennen von Dateien und Verzeichnissen, das Anlegen eines Verzeichnisses und mehr, werden nicht vom C++-Standard unterstützt. Es gibt im Wesentlichen zwei Möglichkeiten der Realisierung:

- Nutzen der entsprechenden Funktionen der Programmiersprache C. C ist eine Untermenge von C++. Der Nachteil dieses Vorgehens ist die mangelnde Portabilität. So sind gelegentlich unter Windows andere Header-Dateien einzubinden als unter Linux, wie unten in Abschnitt 25.1.3 zu sehen.

- Die bereits erwähnte Boost-Library war schon oft Vorbild – große Teile sind in den C++-Standard eingeflossen. Boost bietet die gewünschten Funktionen mit dem Vorteil, dass die Quellprogramme ohne Änderung auf allen Betriebssystemen, auf denen Boost installiert ist, übersetzt werden können.

Die Bibliothek Boost.Filesystem ist in den Technical Report 2 (TR2) übernommen worden, der vermutlich in einigen Jahren Bestandteil des C++-Standards werden wird. In diesem Abschnitt werden die wichtigsten Dateioperationen am Beispiel gezeigt, wobei ein Teil C-basiert ist und ein anderer Teil Boost nutzt.

25.1.1 Datei oder Verzeichnis löschen

Das folgende Programm kann eine Datei oder ein leeres Verzeichnis löschen. Die C-Funktion `remove()` (Header `<cstdio>`) gibt bei Erfolg 0 zurück, ansonsten einen anderen Wert. Im C-Standard [ISOC] nicht gefordert, aber hilfreich, ist die Ablage eines Fehlercodes in der globalen Variablen `errno`. Die Funktion `strerror(int)` gibt einen zum Fehlercode passenden, implementationsabhängigen C-String zurück.

Listing 25.1: Datei oder leeres Verzeichnis löschen

```
// cppbuch/k25/files/loeschen.cpp
#include<cstdio>   // remove()
#include<cerrno>   // errno
#include<cstring>  // strerror(int)
#include<iostream>
using namespace std;

int main(int argc, char*argv[]) {
   if(argc != 2) {
      cout << "Datei oder leeres Verzeichnis löschen\n"
         "Gebrauch: loeschen.exe name" << endl;
   }
   else {
      if(remove(argv[1]) != 0) {
         cerr << "Löschen von " << argv[1]
              << " fehlgeschlagen: " << strerror(errno) << endl;
      }
   }
}
```

Das Programm kann kein gefülltes Verzeichnis löschen. Boost.Filesystem hat ebenfalls eine Funktion `remove()`, aber um die Funktionalität zu variieren, löscht das nächste Programm ein nicht-leeres Unterverzeichnis. Im Fehlerfall wird nicht `errno` gesetzt, sondern eine Exception geworfen, wie in der Boost-Library üblich. Im Programm lernen Sie auch die Funktionen `exists()` und `is_directory()` kennen.

Listing 25.2: Verzeichnis löschen

```
// cppbuch/k25/files/boost/verzeichnisloeschen.cpp
#include<boost/filesystem/operations.hpp>
#include<iostream>
#include<string>
using namespace std;
```

```
namespace bf=boost::filesystem; // Abkürzung

int main(int argc, char*argv[]) {
   if(argc != 2) {
      cout << "Unterverzeichnis löschen\n"
         "Gebrauch: verzeichnisloeschen.exe name" << endl;
   } else if(argv[1][0] == '.') {
      cout << ". nicht erlaubt. Nur für Unterverzeichnis aufrufen!\n";
   }
   else {
      try {
         bf::path pfad(argv[1]);
         if(bf::exists(pfad) && bf::is_directory(pfad)) {
            bf::remove_all(pfad);
         }
         else {
            cout << "Unterverzeichnis " << argv[1] << " existiert nicht!\n";
         }
      }
      catch(const exception& e) {
         cerr << "Löschen des Verzeichnisses " << argv[1]
              << " fehlgeschlagen: " << e.what() << endl;
      }
   }
}
```

25.1.2 Datei oder Verzeichnis umbenennen

Die C-Funktion `rename()` benennt eine Datei oder ein Verzeichnis um. Wie oben dient `strerror()` zur Fehlerdokumentation.

Listing 25.3: Datei oder Verzeichnis umbenennen

```
// cppbuch/k25/files/umbenennen.cpp
#include<cstdio>   // rename()
#include<cerrno>   // errno
#include<cstring>  // strerror(int)
#include<iostream>
using namespace std;

int main(int argc, char*argv[]) {
   if(argc != 3) {
      cout << "Datei oder Verzeichnis umbenennen\n"
         "Gebrauch: umbenennen.exe altername neuername" << endl;
   }
   else {
      if(rename(argv[1], argv[2]) != 0) {
         cerr << "Umbenennen von " << argv[1]
              << " fehlgeschlagen: " << strerror(errno) << endl;
      }
   }
}
```

25.1.3 Verzeichnis anlegen

Das folgende Programm zum Anlegen eines Verzeichnisses zeigt betriebssystemabhängige Unterschiede. Zum einen sind die Header verschieden, zum anderen müssen unter Unix die Dateizugriffsrechte angegeben werden. Die Rechte sind schreiben (4), lesen (2), ausführen (1), wobei die Ziffern für jede Gruppe (Eigentümer, Gruppe, alle) addiert werden. Üblich ist die Darstellung als Oktalzahl. 0755 bedeutet: Der Eigentümer darf alles, die Gruppe und der Rest der Welt dürfen das Verzeichnis lesen und ausführen, aber nicht verändern (schreiben). »Ausführen« meint bei einem Verzeichnis, den Inhalt lesen zu können, wie etwa auch darunterhängende Verzeichnisse. Die Oktalzahl wird auch vom Unix-Befehl `chmod` verstanden. Windows hat eine andere Art der Rechtevergabe. Eine Angabe beim Anlegen des Verzeichnisses mit `mkdir()` ist nicht vorgesehen. Im Programm wird die unterschiedliche Behandlung durch das Makro `WIN32` gesteuert.

Listing 25.4: Verzeichnis anlegen mit cstdio

```
// cppbuch/k25/files/verzanlegen.cpp
#include<cstdio>   // mkdir()
#include<cerrno>   // errno
#include<cstring>  // strerror(int)
#include<iostream>
#ifdef WIN32
 #include<direct.h>
#else
 #include<sys/stat.h>
#endif
using namespace std;

int main(int argc, char*argv[]) {
   if(argc != 2) {
      cout << "Verzeichnis anlegen\n"
         "Gebrauch: verzanlegen.exe name" << endl;
   }
   else {
#ifdef WIN32
    int fehler = mkdir(argv[1]);
#else
    int fehler = mkdir(argv[1], 0755);
#endif
     if(fehler != 0) {
        cerr << "Anlegen des Verzeichnisses " << argv[1]
             << " fehlgeschlagen: " << strerror(errno) << endl;
     }
   }
}
```

Die umständliche zweifache Makro-Abfrage entfällt in dem folgenden Programm, das die Boost-Funktion `create_directory()` nutzt:

Listing 25.5: Verzeichnis anlegen mit Boost.Filesystem

```
// cppbuch/k25/files/boost/verzeichnisanlegen.cpp
#include<boost/filesystem/operations.hpp>
#include<iostream>
```

```
using namespace std;

int main(int argc, char*argv[]) {
   if(argc != 2) {
      cout << "Verzeichnis anlegen\n"
         "Gebrauch: verzeichnisanlegen.exe name" << endl;
   }
   else {
      try {
         boost::filesystem::create_directory(argv[1]);
      }
      catch(const exception& e) {
         cerr << "Anlegen des Verzeichnisses " << argv[1]
              << " fehlgeschlagen: " << e.what() << endl;
      }
   }
}
```

25.1.4 Verzeichnis anzeigen

Das folgende Programm zeigt ein Verzeichnis an, indem ein `directory_iterator` in STL-Manier über das Verzeichnis wandert. Neben dem Namen werden auch die Größe der Dateien und das letzte Modifikationsdatum ermittelt und angezeigt.

Listing 25.6: Verzeichnis anzeigen

```
// cppbuch/k25/files/boost/verzeichnisanzeigen.cpp
#include<boost/filesystem/operations.hpp>
#include<iostream>
#include<string>
using namespace std;
namespace bf=boost::filesystem;

int main(int argc, char*argv[]) {
   if(argc > 2) {
      cout << "Verzeichnis anzeigen\n"
         "Gebrauch: verzeichnisanzeigen.exe [name]" << endl;
   }
   else {
      string verz("."); // aktuelles Verzeichnis
      if(argc > 1) {
         verz = argv[1];
      }
      try {
         bf::path pfad(verz);
         bf::directory_iterator di(pfad), ende;
         while(di != ende) {
            bf::path p = di->path();
            bf::file_status status = di->status();
            cout << p.file_string() << '\t';
            if(bf::is_directory(status)) {
               cout << " (Verzeichnis)";
            }
```

```
            else {
               cout << bf::file_size(p) << " Bytes";
            }
            time_t t = bf::last_write_time(p);
            cout << '\t' << ctime(&t);
            ++di;
         }
      }
      catch(const exception& e) {
         cerr << "Anzeige des Verzeichnisses " << argv[1]
              << " fehlgeschlagen: " << e.what() << endl;
      }
   }
}
```

25.1.5 Verzeichnisbaum anzeigen

Mit den obigen Beispielen liegt alles vor, um auch einen Verzeichnisbaum bearbeiten zu können. Als Beispiel dient ein Programm zur Anzeige der Baumstruktur, ähnlich wie sie das Unix-Programm `tree` liefert (vergleiche Seite 605).

Listing 25.7: Verzeichnisbaum anzeigen

```
// cppbuch/k25/files/boost/tree/main.cpp
#include<iostream>
#include<string>
#include"tree.h"
using namespace std;

int main(int argc, char*argv[]) {
   if(argc > 2) {
      cout << "Verzeichnisbaum anzeigen\n Gebrauch: tree.exe [name]" << endl;
   }
   else {
      string verz("."); // aktuelles Verzeichnis
      if(argc > 1) {
         verz = argv[1];
      }
      try {
         baumAnzeigen(verz);
      }
      catch(const exception& e) {
         cerr << argv[1] << ": Fehler: " << e.what() << endl;
      }
   }
}
```

Die Ablauflogik liegt in der Funktion `baumAnzeigen(verzeichnis)`. Die Header-Datei ist trivial:

Listing 25.8: tree.h

```
// cppbuch/k25/files/boost/tree/tree.h
#ifndef TREE_H
```

```
#define TREE_H
#include<string>
void baumAnzeigen(const std::string& verz);
#endif
```

Die Funktion `baumAnzeigen()` ruft eine überladene Funktion gleichen Namens auf, die sich selbst aufruft, wobei der Zähler für die Einrückungsebene (`level`) erhöht wird. Der Verzeichnisbaum wird also rekursiv durchwandert. Die Rekursion bricht ab, wenn es auf einer Ebene keine Verzeichnisse mehr gibt. Damit die rekursive Funktion nicht direkt aufgerufen werden kann, fehlt sie in *tree.h* und ist in einem anonymen Namespace angelegt.

Listing 25.9: tree.cpp

```
// cppbuch/k25/files/boost/tree/tree.cpp
#include<iostream>
#include<stdexcept>
#include<boost/filesystem/operations.hpp>
#include"tree.h"
namespace bf = boost::filesystem;

namespace {
   void baumAnzeigen(const bf::path& p, int level) {
      bf::directory_iterator di(p), ende;
      while(di != ende) {
         for(int i = 0; i < level; ++i) {
            std::cout << "| ";
         }
         std::cout << "|-- " << di->filename() << std::endl;
         if(bf::is_directory(di->status())) {
            baumAnzeigen(di->path(), level+1);
         }
         ++di;
      }
   }
}  // anonymer Namespace

void baumAnzeigen(const std::string& verz) {
   bf::path pfad(verz);
   if(bf::is_directory(pfad)) {
      std::cout << pfad.filename() << std::endl;
      baumAnzeigen(pfad, 0);
   }
   else {
      throw std::runtime_error(" ist kein Verzeichnis!");
   }
}
```

Das Programm, ohne Argumente im Verzeichnis *cppbuch/k25/files/boost* aufgerufen, gibt aus (der Punkt am Anfang steht für das aktuelle Verzeichnis):

```
.
 |-- verzeichnisanzeigen.cpp
 |-- verzeichnisloeschen.cpp
 |-- README.txt
 |-- tree
 |   |-- main.o
 |   |-- tree.o
 |   |-- README.txt
 |   |-- tree.cpp
 |   |-- tree.exe
 |   |-- tree.h
 |   |-- makefile
 |   |-- main.cpp
 |-- verzeichnisanlegen.cpp
 |-- makefile
```

25.2 Tabelle formatiert ausgeben

Als Beispiel soll eine Tabelle von Sinus- und Kosinuswerten formatiert ausgegeben werden. Die benötigten Informationen finden Sie zum Nachlesen auf Seite 378. Durch das `fixed`-Bit wird die Darstellung mit Dezimalpunkt erreicht, und mit `precision(6)` wird die Anzahl der Nachkommastellen auf 6 festgelegt. Um ein gleichmäßiges Bild zu erzeugen, werden nachfolgende Nullen ausgegeben (`showpoint` wirkt nur in Verbindung mit `fixed`):

Listing 25.10: Formatierte Ausgabe

```
// cppbuch/k25/tabelle/tabelle.cpp
#include<iostream>
#include<cmath> // cos(), sin(), Konstante M_PI für π
// manche Compiler stellen Konstanten wie M_PI nicht zur Verfügung
#ifndef M_PI
 #define M_PI 3.14159265358979323846
#endif
using namespace std;

int main() {
   cout << "Grad     sin(x)       cos(x)\n";
   cout.setf(ios::showpoint|ios::fixed, ios::floatfield);
   cout.precision(6);
   for(int grad = 0; grad <= 90; grad += 10) {
        // Grad in Bogenmaß umwandeln
      const double rad = static_cast<double>(grad)/180.0*M_PI;
      cout.width(4);    cout << grad;
      cout.width(12);   cout << sin(rad);
      cout.width(12);   cout << cos(rad) << endl;
   }
}
```

Das Programm gibt aus:

Grad	*sin(x)*	*cos(x)*
0	*0.000000*	*1.000000*
10	*0.173648*	*0.984808*
20	*0.342020*	*0.939693*
...	usw.	

25.3 Formatierte Daten lesen

Die Art des Einlesens hängt von der Formatierung ab und auch von den Objekten, die die Daten aufnehmen sollen. Es bieten sich zwei verschiedene Möglichkeiten an, von denen die zweite vorgestellt wird:

- Einlesen einer Zeichenkette, die dann ausgewertet wird.
- Überladen des Eingabeoperators >> für benutzerdefinierte Datentypen.

25.3.1 Eingabe benutzerdefinierter Typen

C++ ermöglicht die Eingabe benutzerdefinierter Datentypen, indem der Eingabeoperator >> überladen wird. Ebenso wie beim Überladen des Ausgabeoperators muss das erste Argument eine Referenz auf den Stream (hier also `istream`) sein, um eine Hintereinanderschaltung zu erlauben. Die Analogie zum auf Seite 322 behandelten Ausgabeoperator << liegt auf der Hand. An dieser Stelle wird ein Eingabeoperator für die uns schon bekannte Klasse `Datum` angegeben. Es soll möglich sein, ein Datum im Format Tag◇Monat◇Jahr einzugeben, wobei das Trennzeichen ◇ entweder ein Punkt (.) oder ein Schrägstrich (/) sein darf:

```
// Anwendung des Eingabe-Operators für Datumswerte:
Datum dat1, dat2;
cout << "Eingabe zweier Daten: ";
cin >> dat1 >> dat2;     // Verkettung von >>
cout << dat1 << endl     // Lösung der Aufgaben von Seite 337 vorausgesetzt
     << dat2 << endl;
cout << "Eingabe von Daten bis zum Fehler\n";
while(cin) {
  try {
     cout << "Datum ?";
     cin >> dat1;
     cout << dat1 << endl;
  } catch(const char* e) {
    cout << e << " Abbruch!" << endl;
  }
}
```

Um das Programm zu realisieren, wird in der aus Kapitel 9 bekannten Datei *datum.h* der >>-Operator als globale Funktion deklariert. Die unten gezeigte zugehörige Implementa-

tion muss in der Datei *datum.cpp* nachgetragen werden. Ferner sollte der Typumwandlungsoperator durch eine Methode `toString()` ersetzt werden, damit der Compiler nicht versucht, `operator>>(istream&, string&)` zu benutzen.

```
// Auszug aus cppbuch/k25/datum/datum.h
std::istream& operator>>(std::istream&, Datum&) throw(const char*);

// Implementierung, Auszug aus cppbuch/k25/datum/datum.cpp
std::istream& operator>>(std::istream &eingabe, Datum &d) {
// Einlese-Operator für ein Datum
// erlaubte Formate: Tag.Monat.Jahr oder Tag/Monat/Jahr
   char c = '\0';
   int tag, monat, jahr;
   eingabe >> tag >> c;          // Tag und 1. Trennzeichen
   if(c !='.' && c !='/') {
      eingabe.setstate(std::ios::failbit); // Status setzen
   }
   else {
       eingabe >> monat >> c;    // Monat und 2. Trennzeichen
       if(c !='.' && c !='/') {
          eingabe.setstate(std::ios::failbit); // Status setzen
       }
       else {
          eingabe >> jahr;
       }
       if(jahr < 100) {
         jahr += 2000;
       }
       // Datum gültig?
       if(istGueltigesDatum(tag, monat, jahr)) { // nur dann
          d.set(tag, monat, jahr);
       }
       else {
          eingabe.setstate(std::ios::failbit);
       }
   }
   if(!eingabe.good()) {
      throw "kein gültiges Datum!";
   }
   return eingabe;
}
```

Die Variable `c` wird mit 0 initialisiert, damit sie nicht zufällig einen der erlaubten Werte annimmt, wenn die Eingabe von `tag` fehlschlagen sollte. Ein Eingabefehler beim Einlesen von `tag`, `monat` oder `jahr` bewegt das C++-Laufzeitsystem zum Setzen des `failbit`, weil es sich um `int`-Werte handelt. Ein anderer Eingabefehler führt im überladenen Operator ebenfalls zu einen `failbit`-Fehler, sodass die obige Abfrage `while(cin)` bei Fehlern zum Schleifenabbruch führt.

25.4 Array als Block lesen oder schreiben

Das Beispiel auf Seite 221 zeigt bereits das Schreiben eines Arrays als Block. Es fehlt jedoch das Lesen, und die Fehlerbehandlung besteht nur im Programmabbruch. Im Folgenden werden Lesen und Schreiben zur einfacheren Verwendung in jeweils eine einfach aufzurufende Funktion gepackt. Der Template-Parameter `T` steht für den Typ eines Array-Elements. Im Fehlerfall wird eine Exception geworfen:

Listing 25.11: Funktionen zur blockweisen binären Ein- und Ausgabe

```
// cppbuch/k25/binaer/binaerIO.t
#ifndef BINAERIO_T
#define BINAERIO_T
#include<ios>
#include<fstream>
#include"IOException.h" // siehe Text

template<typename T>
void schreibeBinaer(const std::string& dateiname, const T* daten,
                    size_t anzBytes) {
   std::ofstream ziel;
   ziel.open(dateiname.c_str(), std::ios::binary|std::ios::out);
   if (!ziel) {
      throw IOException(dateiname);
   }
   ziel.write(reinterpret_cast<const char*>(daten), anzBytes);
   ziel.close();
}

template<typename T>
void liesBinaer(const std::string& dateiname, T* daten, size_t anzBytes) {
   std::ifstream quelle;
   quelle.open(dateiname.c_str(), std::ios::binary|std::ios::in);
   if (!quelle) {
      throw IOException(dateiname);
   }
   quelle.read(reinterpret_cast<char*>(daten), anzBytes);
   quelle.close();
}
#endif
```

Hinweis

Die beiden Funktionen sind nur für Arrays geeignet, deren Elemente keine Zeiger enthalten. Von den Zeigern referenzierte Objekte würden nicht mitkopiert werden.

Die Klasse `IOException` sorgt dafür, dass der Dateiname im Fehlerfall an den Aufrufer übermittelt wird. Wie das im Einzelnen vonstatten geht, lesen Sie in der Beschreibung der Klasse `IOException` auf Seite 574 sowie in der nachfolgenden Beispielanwendung.

Listing 25.12: Anwendung: zweidimensionale Matrix schreiben und lesen

```
// cppbuch/k25/binaer/binaerIO.cpp
#include<cstdlib>
#include<iostream>
#include<algorithm>    // equal()
#include"binaerIO.t"
using namespace std;

int main() {
   const string dateiname("binaerdaten.bin");
   const int ZEILEN = 10;
   const int SPALTEN = 8;
   double matrix[ZEILEN][SPALTEN];
   // Matrix mit (hier beliebigen) Werten füllen und anzeigen:
   cout << "Matrix:\n";
   for(int i = 0; i < ZEILEN; ++i) {
      for(int j = 0; j < SPALTEN; ++j) {
         matrix[i][j] = i*SPALTEN + j;
         cout << matrix[i][j] << '\t';
      }
      cout << endl;
   }
   cout << endl;
   // Matrix schreiben
   size_t anzahlBytes = ZEILEN*SPALTEN*sizeof(matrix[0][0]);
   try {
      schreibeBinaer(dateiname, matrix, anzahlBytes);
   }
   catch(const IOException& e) {
      cout << e.what() << endl;
      return 1;
   }
   // Kopie anlegen und mit 0 vorbesetzen
   double kopie[ZEILEN][SPALTEN] = {{0}, {0}};
   // Kopie von der Datei einlesen
   try {
      liesBinaer(dateiname, kopie, anzahlBytes);
   }
   catch(const IOException& e) {
      cout << e.what() << endl;
      return 2;
   }
   // Werte vergleichen zum Nachweis des korrekten Schreibens/Lesens
   if(equal(matrix[0], matrix[0] + ZEILEN*SPALTEN, kopie[0]))
      cout << "Geschriebene und gelesene Daten sind gleich." << endl;
   else
      cout << "Geschriebene und gelesene Daten sind ungleich!" << endl;
}
```