16 Datenbankanbindung

Dieses Kapitel behandelt die folgenden Themen:

- Datenbankanbindung am Beispiel SQLite
- C++-Schnittstelle
- Anwendungsbeispiel

Viele Applikationen bauen auf der Nutzung einer Datenbank auf. Ein Datenbanksystem besteht aus der eigentlichen Datenbank und einer Software zur Verwaltung (DBMS = Datenbankmanagementsystem). Die meisten Datenbanksysteme sind sogenannte relationale Datenbanken (RDBMS), die intern als Sammlungen von Tabellen (Relationen) organisiert sind. Mittlerweile gibt es auch objektorientierte Datenbanken oder objektrelationale Mischformen, die nicht sehr verbreitet sind. Operationen auf diesen Tabellen werden mit Hilfe der Structured Query Language (SQL) bewerkstelligt. Es gibt zwei typische Szenarien:

- Die Datenbank ist als Server realisiert, gegebenfalls auf einem entfernten Rechner. Clienten greifen über ein Netzwerk auf die Datenbank zu. Weil die Clienten voneinander unabhängig sind, kann es viele gleichzeitige Zugriffe auf die Datenbank geben. Wegen der Trennung von den Applikationen ist dieses Modell geeignet, wenn sehr viele Daten verwaltet werden oder/und ein hoher Durchsatz erforderlich ist, wie er bei einer stark frequentierten Website auftritt. Typische Vertreter so einer Datenbank

sind Oracle oder DB2 oder im Open Source Bereich die verbreiteten Systeme MySQL und PostgreSQL.

- Die Datenbank ist an eine Applikation gebunden. Sie kann eine Menge von Dateien ersetzen, in denen vorher lokale Daten abgelegt wurden. So eine Datenbank eignet sich für Desktop-Anwendungen verschiedener Art und wegen ihrer geringen Größe für mobile Geräte. SQLite [SQLite] ist ein freies RDBMS, das dafür sehr gut einsetzbar ist und sehr wenig administrativen Aufwand erfordert.

Die C++-Anbindung an eine Datenbank ist in beiden Fällen ähnlich, weil entsprechende APIs zur Verfügung gestellt werden. Für dieses Kapitel habe ich mich entschieden, die Datenbankanbindung am Beispiel von SQLite zu zeigen, weil der Einsatz einer solchen Datenbank in vielen Fällen ausreichend ist. Alle zu einer Datenbank gehörenden Daten werden von SQLite in einer einzigen Datei abgelegt. SQLite ist portabel und wird millionenfach in bekannten Produkten eingesetzt (Mozilla Firefox, Google Gears, Adobe Photoshop Lightroom, Handys mit Symbian-Betriebssystem und andere) und ist integraler Bestandteil der Programmiersprachen PHP und Python.

Hinweis

Dieses Kapitel konzentriert sich auf die Ansprache von SQLite mit C++, um Ihnen einen leichten *Einstieg* in das Thema Datenbankanbindung zu ermöglichen. In den Beispielen wird auf SQL und auf die umfangreichen Funktionen von SQLite nur in aller Kürze eingegangen. Bei tiefergehendem Bedarf bitte ich Sie, ein Buch zu Datenbanken wie [Date] und die Webseite [SQLite] zu konsultieren.

SQLite ist leicht zu installieren, unter Linux am besten mit dem Werkzeug zur Systemverwaltung. Es muss nur darauf geachtet werden, die neuere Version *sqlite3* und auch das Package *sqlite3-devel* zu installieren. Unter Windows brauchen Sie nichts weiter zu tun, wenn Sie die Standardinstallation des Compilers und anderer Programme von der DVD zum Buch entsprechend der Anleitung vornehmen; SQLite ist dort bereits integriert.

16.1 C++-Interface

SQLite ist vollständig in C geschrieben und stellt an die 175 Funktionen zur Verfügung. Ergebnisse werden oft in einem Speicherbereich bereitgestellt, der intern mit der C-Speicherbeschaffungsfunktion `malloc` angelegt wird, für dessen Freigabe jedoch der Aufrufer der Funktion verantwortlich ist. Das kann auch mal vergessen werden, weswegen es guter Stil in C++ ist, die Freigabe in den Destruktor zu verlegen oder anderweitig automatisch sicherzustellen. Die Klasse `Datenbank` unten kapselt den Zugriff auf die Datenbank und garantiert die Freigabe, indem die ensprechenden `sqlite3_free...()`-Funktionen aufgerufen werden.

Glücklicherweise kommt man in den meisten Fällen mit nur wenigen der vielen Funktionen aus, weil es darunter welche gibt, die sehr viel Funktionalität bieten. Die wichtigsten Funktionen sind:

- `int sqlite3_open(const char* dateiname, sqlite** ptr)` öffnet die in der Datei `dateiname` vorliegende Datenbank, wenn die Datei existiert. Wenn nicht, wird sie neu erzeugt. Falls der Dateiname `:memory` lautet, wird eine temporäre Datenbank im Speicher angelegt. Falls der Dateiname ein leerer C-String ist, wird eine Datenbank als temporäre Datei geschaffen. Diese temporäre Datei wird sofort nach Schließen der Datenbank gelöscht. `ptr` ist die Adresse eines Zeigers `db`, der nach dem Aufruf auf ein `sqlite3`-Objekt verweist. Mit Hilfe des Handles `db` sind Zugriffe auf die Daten möglich, wie im Folgenden zu sehen. Die Funktion gibt im Erfolgsfall `SQLITE_OK` zurück, andernfalls einen Fehlercode.
- `int sqlite3_close(sqlite* db)`: Verbindung zur Datenbank schließen. Die Funktion gibt im Erfolgsfall `SQLITE_OK` zurück, andernfalls einen Fehlercode.
- ```
  int sqlite3_get_table(
    sqlite3 *db,       // geöffnete Datenbank
    const char *sqlAnw, // SQL-Anweisung
    char*** pTabelle,  // Hierhin wird das Ergebnis der Anweisung geschrieben.
    int *pZ,           // Die Anzahl der Zeilen wird an pZ geschrieben.
    int *pS,           // Die Anzahl der Spalten wird an pS geschrieben.
    char **pfehlermsg // Hierhin wird ggf. eine Fehlermeldung geschrieben.
  )
  ```

  Falls die auszuwertende SQL-Anweisung kein Ergebnis liefert, etwa bei einer `insert`-Anweisung, bleibt die Tabelle leer. Die Funktion gibt im Erfolgsfall `SQLITE_OK` zurück, andernfalls einen Fehlercode.
- `void sqlite3_free(void* fehlermsg)`: gibt den für die Fehlermeldung angelegten Speicherplatz frei.
- `int sqlite3_free_table(char** tabelle)`: gibt den für die Tabelle angelegten Speicherplatz frei. `sqlite3_free()` genügt nicht!
- `const char* sqlite3_errmsg(sqlite* db)`: gibt die zum zuletzt aufgetretenen Fehler passende Fehlermeldung zurück.
- ```
  int sqlite3_exec(
    sqlite3 *db,       // geöffnete Datenbank
    const char *sqlAnw, // SQL-Anweisung
    int (*f)(void*, int, char**, char**), // Callback-Funktion
    void* arg,         // Erstes Argument der Callback-Funktion
    char **pfehlermsg // Hierhin wird ggf. eine Fehlermeldung geschrieben.
  )
  ```

  Ähnlich wie `sqlite3_get_table()` führt diese Funktion eine SQL-Anweisung aus. Der Unterschied besteht darin, dass keine Tabelle zurückgegeben, sondern die Funktion `f` für jede Zeile aufgerufen wird. Der Funktion `f` werden übergeben: `arg` (Parameter 1), die Anzahl der Spalten des aktuellen Ergebnisses (Parameter 2), ein Array von Zeigern auf C-Strings für jede Spalte (Parameter 3), ein Array von Zeigern auf C-Strings für jede Spaltenüberschrift (Parameter 4). Die selbst zu schreibende Funktion kann diese Parameter nach Wunsch auswerten.

Bis auf die letzte Funktion, für die es in [SQLite] ein Beispiel gibt, werden alle genannten Funktionen unten eingesetzt.

Hinweis

Wenn Sie sich bei den Dateinamen und SQL-Anweisungen auf ASCII beschränken, sind Sie auf der sicheren Seite. Andernfalls wird vorausgesetzt, dass alle Zeichenketten UTF-8 codiert sind. UTF-16 ist möglich, setzt aber andere Funktionsaufrufe voraus [SQLite].

Die Funktion `execute(string sqlcmd)` der Klasse `Datenbank` führt eine beliebige, als String übergebene SQL-Anweisung aus. Die Anweisung wird an die Funktion `sqlite3_get_table()` weitergeleitet, die als Ergebnis eine Tabelle in Form eines C-String-Arrays zurückgibt. Um den Speicherplatz wieder freigeben zu können, wird die Tabelle in dem praktischen zweidimensionalen Feld `Array2d<std::string>` aus Abschnitt 5.7.3 gespeichert. Die Funktion `execute(string sqlcmd)` gibt dieses Array als Ergebnis zurück.

Listing 16.1: Klasse Datenbank: SQLite C++-Schnittstelle

```
// cppbuch/k16/Datenbank.h
#ifndef DATENBANK_H
#define DATENBANK_H
#include<string>
#include<stdexcept>
#include<sqlite3.h>
#include<array2d.h>

struct SQLError : public std::runtime_error {
   SQLError(const std::string& msg)
      : std::runtime_error (std::string("SQL-Fehler: ") + msg) {
   }
};

class Datenbank {
public:
   Datenbank(const char* dateiname)
      : db(0) {
      if(sqlite3_open(dateiname, &db) != SQLITE_OK) { // Fehler?
         const char* fehlermeldung = sqlite3_errmsg(db);
         sqlite3_close(db);
         throw SQLError(fehlermeldung);
      }
   }

   ~Datenbank() {
      sqlite3_close(db);
   }

   Array2d<std::string> execute(const std::string& sqlAnweisung) const {
      // 4 Variable zur Speicherung des Ergebnisses aus sqlite3_get_table()
      char* fehlermeldung = 0;
      int zeilen;
      int spalten;
      char** cstringarray = 0;
      int erg = sqlite3_get_table(db, sqlAnweisung.c_str(),
                                  &cstringarray,          // Ergebnistabelle
```

```
                              &zeilen, &spalten, &fehlermeldung);
    Array2d<std::string> ergebnis(1, 1, ""); // Platzhalter
    if(erg != SQLITE_OK) {               // Fehler?
       std::string msg(fehlermeldung);
       sqlite3_free(fehlermeldung);      // Freigaben nicht vergessen!
       sqlite3_free_table(cstringarray);
       throw SQLError(msg);
    }
    else {
       if(zeilen > 0) { // Zeilen einsammeln, eine mehr für die Überschrift
          ergebnis = Array2d<std::string>(++zeilen, spalten);
          for(int z = 0; z < zeilen; ++z) {
             for(int s = 0; s < spalten; ++s) {
                const char* str = cstringarray[z*spalten + s];
                ergebnis[z][s] = str ? str : ""; // NULL berücksichtigen
             }
          }
       }
       sqlite3_free_table(cstringarray); // Freigabe nicht vergessen!
    }
    return ergebnis;
  }
private:
  sqlite3 *db;
  Datenbank(const Datenbank&);         // kopieren verbieten
  Datenbank& operator=(const Datenbank&); // zuweisen verbieten
};
#endif
```

Das folgende kleine Programm bedient sich der obigen Schnittstelle. Mit ihm können interaktiv SQL-Anweisungen eingegeben und deren Ergebnisse angezeigt werden. Die globale Funktion `printArray()` gehört zur Klasse `Array2d`.

Listing 16.2: Programm zur interaktiven Eingabe von SQL-Anweisungen

```
// cppbuch/k16/abfrage.cpp
#include<iostream>
#include"Datenbank.h"

int main(int argc, char **argv){
  std::string zeile;
  std::cout << "Datenbankname: ";
  std::getline(std::cin, zeile);
  try {
    Datenbank db(zeile.c_str());
    do {
      std::cout << "SQL-Anweisung: ";
      std::getline(std::cin, zeile);
      if(zeile.length() > 0) {
        printArray(db.execute(zeile.c_str()));
      }
    } while(zeile.length() > 0);
  } catch(const SQLError& e) {
    std::cout << e.what() << std::endl;
```

```
  }
}
```

Zuerst wird der Name der Datenbankdatei abgefragt, dann die SQL-Anweisungen. Eine leere Anweisung bewirkt den Abbruch des Programms. Dies Struktur erlaubt die Abarbeitung von Skripten, zum Beispiel `abfrage.exe < skript.txt`, wobei das Skript etwa wie folgt aussehen könnte:

```
shopdb.db
select * from Kunde;
select * from Rechnung;
```

16.2 Anwendungsbeispiel

Um eine konkrete Anwendung zu zeigen, wird zunächst eine kleine Datenbank definiert. Sie enthält Kunden, Adressen, Artikel, Rechnungen und Rechnungspositionen. Die Aufgabe besteht darin, per Programm Rechnungen zu schreiben. Das Datenbankschema ist durch die folgenden erzeugenden SQL-Anweisungen gegeben (Erläuterungen folgen):

Listing 16.3: Datenbankschema

```
-- cppbuch/k16/shopdb.txt
create table Adresse (
  Id integer primary key autoincrement,
  Ort text,
  Plz text,
  StrNr text);

create table Kunde (
  Id integer primary key autoincrement,
  Name text,
  AdressId integer not null,
  foreign key(AdressId) references Adresse(Id));

create table Artikel (
  Nr integer primary key,
  Bezeichnung text,
  Preis numeric(8,2) not null);

create table Rechnung (
  Id integer primary key autoincrement,
  Datum date,
-- Kundennr darf NULL sein (Barverkauf)
  Kundennr integer,
  foreign key(Kundennr) references Kunde(Id));

create table Position (
  Rechnungsnr integer not null,
```

```
  Artikelnr integer not null,
  Menge integer,
  check (Menge > 0),
  foreign key(Rechnungsnr) references Rechnung(Id),
  foreign key(Artikelnr) references Artikel(Nr));

create trigger datumtrigger after insert on Rechnung
begin
  update Rechnung set Datum = DATE('NOW') where rowid = new.rowid;
end;

create view positionen as
   select Rechnungsnr, Artikelnr,Menge,Bezeichnung, Preis, Preis*Menge
   from (Position p join Rechnung r on Rechnungsnr=r.Id)
   left outer join Artikel art on Artikelnr=art.Nr;

-- alle Rechnungsinfos
create view alles as
   select Rechnungsnr,Datum,Name,Plz,Ort,StrNr,Menge,Artikelnr,
          Bezeichnung, Preis, Preis*Menge
   from (((Position p join Rechnung r on Rechnungsnr=r.Id)
       left outer join Kunde k on Kundennr=k.Id)
       left outer join Artikel art on Artikelnr=art.Nr)
       left outer join Adresse a on AdressId=a.Id;
```

Die Datenbank *shopdb.db* wird neu erzeugt, indem eine möglicherweise vorher existierende Datei gleichen Namens gelöscht und dann

```
sqlite3 shopdb.db < shopdb.txt
```

aufgerufen wird. Einem Kunden ist eine Adresse nicht direkt, sondern als Verweis (Fremdschlüssel) zugeordnet, weil erstens ein Kunde umziehen kann und zweitens mehrere Kunden dieselbe Adresse haben können (Mehrfamilienhaus). Damit werden die Informationen entkoppelt. `primary key` besagt, dass der Eintrag zur eindeutigen Identifizierung genommen werden kann. `not null` bedeutet, dass der Eintrag nicht leer sein darf.

Eine Rechnung enthält das Datum und die Kundennummer. Das Datum wird automatisch beim Anlegen einer neuen Rechnung eingetragen, bewirkt durch den Trigger `datumtrigger`. Eine Position verweist auf die zugehörige Rechnung und den Artikel. Die Abbildung 16.1 zeigt die Struktur in grafischer Form.

Am Ende der Datei sind zwei Sichten (englisch *view*) definiert. Die Sicht `positionen` selektiert die Positionen. Die Abfrage `select * from positionen where Rechnungsnr = 1;` gibt alle zur Rechnung 1 gehörenden Positionen als Tabelle zurück, wobei eine Zeile aus den Informationen Rechnungsnr, Artikelnr, Menge, Bezeichnung, Preis und Preis*Menge besteht. Wie Sie sehen, können auch arithmetische Operationen vorgenommen werden. Die Sicht `alles` liefert sämtliche in der Datenbank enthaltenen Informationen, die mit `where` natürlich wie oben gefiltert werden können. Um die Datenbank zu füllen, wird

```
sqlite3 shopdb.db < shopdbfuellen.txt
```

aufgerufen. Die Datei *shopdbfuellen.txt* enthält SQL-Anweisungen, um Daten in die verschiedenen Tabellen einzutragen. Dabei werden teilweise die Fremdschlüssel der Kürze halber direkt angegeben, teilweise mit `select` ermittelt.

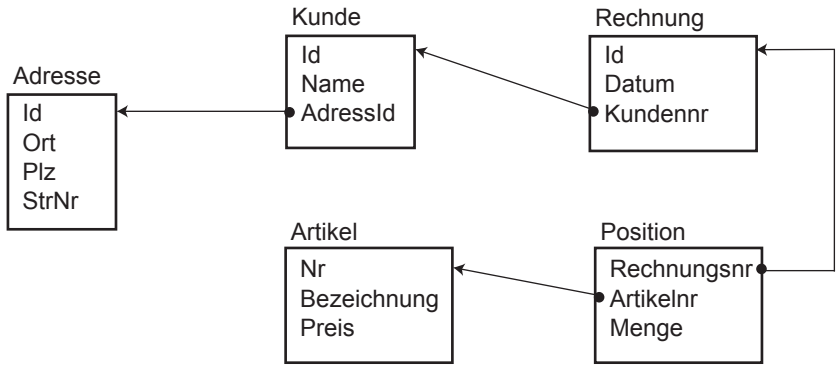

Abbildung 16.1: Datenbankschema

Listing 16.4: Datenbankinhalt für das Beispiel

```
-- cppbuch/k16/shopdbfuellen.txt
insert into Adresse (Ort, Plz, StrNr) values(
                                'Entenhausen', '12345', 'Erpelweg 7');
insert into Adresse (Ort, Plz, StrNr) values(
                                'Muenchen', '81679', 'Kolbergerstr. 22');
insert into Adresse (Ort, Plz, StrNr) values(
                                'Barcelona', '08037', 'Roger De Lluria 21');

-- Zahl = Adress-Index
insert into Kunde (Name, AdressId) values('Donald Duck', 1);
insert into Kunde (Name, AdressId) values('Addy Woollen', 3);
insert into Kunde (Name, AdressId) values('Carl Hanser', 2);
-- alternativ: mit select ermitteln:
insert into Kunde (Name, AdressId) values('Daisy Duck',
       (select Id from Adresse where Ort='Entenhausen'));

insert into Artikel values(4727, 'Schokolinsen', 1.10);
insert into Artikel values(123, 'Champagner', 39.99);
insert into Artikel values(9282, 'Kaffeemaschine', 21.95);
insert into Artikel values(1912, 'Rezeptbuch', 9.00);

-- Daisy kaufte 20 Schokolinsen, 1 Rezeptbuch, 1 Champagner
insert into Rechnung (Kundennr) values((select Id from Kunde where
                                                    Name='Daisy Duck'));
-- Rechnungsnr, Artikelnr, Stueckzahl
insert into Position values(1, 4727, 20);
insert into Position values(1, 1912, 1);
-- oder mit select:
insert into Position values((select Id from Rechnung where Rechnung.Kundennr=
     (select Id from Kunde where Name == 'Daisy Duck')), 123, 1);

-- Barverkauf, keine Kundennr
insert into Rechnung (Kundennr) values(NULL);
insert into Position values(2, 9282, 2);
```

```
-- Rechnung 3 fuer Kunde Nr. 3 (Carl Hanser)
insert into Rechnung (Kundennr) values(3);
insert into Position values(3, 123, 10);
insert into Position values(3, 1912, 1);

-- Rechnung 4 fuer Kunde Nr. 2 (Addy Woollen)
insert into Rechnung (Kundennr) values(2);
insert into Position values(4, 123, 50);
```

Rechnungen schreiben

Die obige Datenbank soll so ausgewertet werden, dass alle Rechnungen, denen ein Kunde zugeordnet ist (kein Barverkauf), zum Versand ausgedruckt werden sollen. Der Einfachheit halber wird auf das Währungssymbol und die Mehrwertsteuerberechnung verzichtet; beide sind bei Bedarf leicht nachzurüsten. Die Aufgabe wird auf zwei Arten gelöst:

- Die Anfrage `select * from alles;` wird ausgewertet.
- Im zweiten Fall wird gezeigt, wie die benötigten Informationen gezielt mit einzelnen `select`-Anweisungen ermittelt werden.

Listing 16.5: Rechnungen schreiben mit `select * from alles;`

```
// cppbuch/k16/rechnung.cpp
#include<iostream>
#include"Datenbank.h"

namespace {
   // SQL-Zahlen auf 2 dargestellte Nachkommazahlen bringen
   std::string formatPreis(const std::string& s) {
      size_t punkt = s.find('.');
      if(punkt == std::string::npos) {
         return s + ".00";
      } else if(s.length() - punkt == 2) {
         return s + "0";
      }
      return s;
   }
}

int main(int argc, char **argv){
   enum Struktur {NR, DATUM, NAME, PLZ, ORT, STRNR, MENGE, ARTNR, BEZ,
                  EPREIS, GPREIS};
   try {
      Datenbank db("shopdb.db");
      Array2d<std::string> rechnungen = db.execute("select * from alles");
      std::string rechnNr("");
      int z = 1;
      bool rechnungskopfGedruckt = false;
      while(z < rechnungen.getZeilen()){
         if(rechnNr != rechnungen[z][NR]) { // neue Rechnung beginnt
            rechnNr = rechnungen[z][NR];
            rechnungskopfGedruckt = false;
         }
```

```
        if(rechnungen[z][NAME] != "") { // Kunde existiert, kein Barverkauf
           int position = 0;
           if(!rechnungskopfGedruckt) {
              std::cout
                 << "Herrn/Frau/Firma " << rechnungen[z][NAME] << std::endl
                 << rechnungen[z][STRNR] << std::endl
                 << rechnungen[z][PLZ] << " " << rechnungen[z][ORT]
                 << "\n\n\nRechnung Nr. " << rechnNr
                 << "  vom " << rechnungen[z][DATUM] << std::endl
                 << "Pos. Menge  Artikelnr.        Bezeichnung "
                 "Einzelpreis  Gesamtpreis" << std::endl;
              rechnungskopfGedruckt = true;
              position = 1;
           }
           // Positionen ausgeben
           std::cout.width(4);
           std::cout << position++;
           std::cout.width(6);
           std::cout << rechnungen[z][MENGE];
           std::cout.width(12);
           std::cout << rechnungen[z][ARTNR];
           std::cout.width(20);
           std::cout << rechnungen[z][BEZ];
           std::cout.width(12);
           std::cout << formatPreis(rechnungen[z][EPREIS]);
           std::cout.width(12);
           std::cout << formatPreis(rechnungen[z][GPREIS]) << std::endl;
        }
        ++z;
        if(rechnungskopfGedruckt &&
           (z == rechnungen.getZeilen()
            || (z < rechnungen.getZeilen()
                && rechnNr != rechnungen[z][NR]))) {
           Array2d<std::string> summe
              = db.execute(
                   "select sum(Preis*Menge) from alles where Rechnungsnr="
                   + rechnNr + ";");
           std::cout.width(66);
           std::cout << "------------" << std::endl;
           std::cout.width(66);
           std::cout << formatPreis(summe[1][0]) << std::endl
                     << '\f' << std::endl;              // Seitenvorschub
        }
     }
  } catch(const SQLError& e) {
     std::cout << e.what() << std::endl;
  }
}
```

Die zurückgegebene Tabelle enthält als erste Zeile die Spaltenüberschriften und in den Folgezeilen alle Positionen mit Rechnungsdatum, Kundenname usw. Wegen der zu langen Zeilen verkürzter Auszug:

```
Rechnungsnr Datum Name Plz Ort StrNr Menge Artikelnr Bezeichnung Preis ...
1 2011-01-16 Daisy Duck 12345 Entenhausen Erpelweg 7 20 4727 Schokolinsen 1.1 ..
1 2011-01-16 Daisy Duck 12345 Entenhausen Erpelweg 7 1 1912 Rezeptbuch 9 ...
usw.
```

Am besten ist es, Sie probieren selbst die Anfrage mit dem Programm *abfrage.exe* oder *sqlite3* aus, um die Tabelle zu sehen. Bei der Auswertung ist zu beachten, dass sich die Rechnungsnummer und Kundennamen bei allen zur selben Rechnung gehörenden Positionen wiederholen. Ein Wechsel in der Rechnungsnummer bedeutet damit Abschluss der laufenden Rechnung und Beginn einer neuen Rechnung. So sieht eine der erzeugten Rechnungen aus:

```
Herrn/Frau/Firma Daisy Duck
Erpelweg 7
12345 Entenhausen

Rechnung Nr. 1 vom 2011-01-16
Pos. Menge  Artikelnr.      Bezeichnung Einzelpreis Gesamtpreis
   1    20       4727       Schokolinsen      1.10      22.00
   2     1       1912         Rezeptbuch      9.00       9.00
   3     1        123         Champagner     39.99      39.99
                                                ------------
                                                     70.99
```

Die Preise werden mit einer SQL-Anweisung aufsummiert, die auf die Spalte Gesamtpreis wirkt (d.h. Preis*Menge):

```
select sum(Preis*Menge) from alles where Rechnungsnr=X;
```

`X` ist hier nur ein Platzhalter. In der zweiten Variante werden zur Demonstration gezielt die Rechnungsnummern ermittelt, und zu jeder Rechnungsnummer werden der zugehörige Kunde und alle zugehörigen Positionen bestimmt. Für jeden Kunden wird die Adresse erfragt. Damit entfallen redundante Einträge wie die Wiederholungen von Rechnungsnummer, Name und Adresse im ersten Beispiel. Das und die Demonstration der verschiedenen Abfragen und Auswertungen bilden den Vorteil dieses Programms. Der Vorteil wird jedoch mit einer erhöhte Komplexität des Codes und durch einen Laufzeitnachteil, bedingt durch die vergleichsweise vielen SQL-Anweisungen, erkauft.

Listing 16.6: Rechnungen schreiben; zweite Variante

```
// Auszug aus cppbuch/k16/rechnung1.cpp
int main(int argc, char **argv){
  try {
    Datenbank db("shopdb.db");
    Array2d<std::string> rechnungen = db.execute("select * from Rechnung");
    // z=1: Ueberschrift ausblenden
    for(int z = 1; z < rechnungen.getZeilen(); ++z) {
      std::string abfrage("select Name,Id from Kunde where Kunde.Id ="
                          " (select  Kundennr from Rechnung where "
                          "Rechnung.Kundennr=Kunde.Id and Rechnung.Id=");
      abfrage += rechnungen[z][0] + ");";
      // Kunde zur Rechnung ermittel
      Array2d<std::string> kunde = db.execute(abfrage);
      if(kunde.getZeilen() > 1) { // Kunde existiert, kein Barverkauf
```

```
            std::cout << "Herrn/Frau/Firma " << kunde[1][0] // Name
                      << std::endl;
            // Adresse des Kunden ermitteln
            abfrage = "select Plz, Ort, StrNr from Adresse where "
               "(select Id from Kunde where Kunde.AdressId=Adresse.Id "
               " and Id=" + kunde[1][1] + ");";
            Array2d<std::string> adresse = db.execute(abfrage);
            std::cout << adresse[1][2] << std::endl // Strasse Hausnr
                      << adresse[1][0] << " " << adresse[1][1] // PLZ Ort
                      << "\n\n\nRechnung Nr. " << rechnungen[z][0]
                      << "  vom " << rechnungen[z][1] << std::endl;
            // Rechnungspositionen ermitteln (View nutzen)
            abfrage = "select * from positionen where Rechnungsnr="
               + rechnungen[z][0] + ";";
            Array2d<std::string> positionen = db.execute(abfrage);
            // Positionen ausgeben
            std::cout << "Pos. Menge Artikelnr.          Bezeichnung "
               "Einzelpreis Gesamtpreis" << std::endl;
            enum Positionen {NR, ARTNR, MENGE, BEZ, EPREIS, GPREIS};
            // p=1: Ueberschrift ausblenden
            for(int p = 1; p < positionen.getZeilen(); ++p) {
               std::cout.width(4);
               std::cout << p;
               std::cout.width(6);
               std::cout << positionen[p][MENGE];
               std::cout.width(12);
               std::cout << positionen[p][ARTNR];
               std::cout.width(20);
               std::cout << positionen[p][BEZ];
               std::cout.width(12);
               std::cout << formatPreis(positionen[p][EPREIS]);
               std::cout.width(12);
               std::cout << formatPreis(positionen[p][GPREIS]) << std::endl;
            }
            // Summe bilden
            abfrage = "select sum(Preis*Menge) from positionen where "
                      "Rechnungsnr=" + rechnungen[z][0] + ";";
            Array2d<std::string> summe = db.execute(abfrage);
            std::cout.width(66);
            std::cout << "------------" << std::endl;
            std::cout.width(66);
            std::cout << formatPreis(summe[1][0]) << std::endl
                      << '\f' << std::endl; // Seitenvorschub
         }
      }
   } catch(const SQLError& e) {
      std::cout << e.what() << std::endl;
   }
}
```