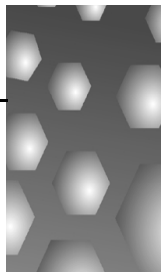
Kapitel 6



Das MySQL-C-API

MySQL stellt eine Client-Bibliothek bereit, die in der Programmiersprache C geschrieben ist und mit deren Hilfe Sie Client-Programme schreiben können, die auf MySQL-Datenbanken zugreifen. Diese Bibliothek definiert ein API (Application Programming Interface), also eine Schnittstelle für die Anwendungsprogrammierung, die die folgenden Leistungsmerkmale aufweist:

- Routinen zur Verbindungsverwaltung, die eine Sitzung mit dem Server einrichten oder beenden
- Routinen, mit denen Anfragen konstruiert, an den Server geschickt und die Ergebnisse verarbeitet werden
- Funktionen zur Meldung von Fehlern und Statuszuständen, um die genaue Ursache für das Fehlschlagen anderer C-API-Aufrufe ermitteln zu können

Dieses Kapitel zeigt die Verwendung der Client-Bibliothek bei der Entwicklung eigener Programme. Unter anderem achten wir dabei auf Konsistenz mit bereits existierenden Client-Programmen in der MySQL-Distribution sowie auf Modularität und Wiederverwendbarkeit des Codes. Ich nehme an, Sie haben gewisse Grundkenntnisse in der C-Programmierung, bin aber bei den Beschreibungen nicht davon ausgegangen, dass Sie ein Fachmann oder eine Fachfrau sind.

Das Kapitel entwickelt mehrere Client-Programme, von ganz einfachen bis hin zu immer komplexeren. Im ersten Abschnitt erstellen wir die Umgebung für ein Client-Gerüst, das keine andere Aufgabe hat, als eine Verbindung zum Server einzurichten oder diese zu trennen. Denn MySQL-Client-Programme sind zwar alle für unterschiedliche Aufgaben geschrieben, aber eines haben sie stets gemeinsam: sie richten eine Verbindung zum Server ein.

Dieses Gerüst wird in einzelnen Schritten aufgebaut:

1. Entwicklung des reinen Codes für das Einrichten und Schließen der Verbindung (`client1`)
2. Einfügen einer Fehlerüberprüfung (`client2`)
3. Modularität und Wiederverwendbarkeit des Verbindungscodes realisieren (`client3`)
4. Einführen einer Möglichkeit, zur Laufzeit Verbindungsparameter anzugeben (Host, Benutzer, Kennwort) (`client4`)

Dieses Gerüst ist weitgehend generisch und Sie können es als Grundlage für die unterschiedlichsten Client-Programme nutzen. Nach der Entwicklung werden wir zeigen, wie verschiedene Anfragen verarbeitet werden. Zunächst beschreiben wir, wie bestimmte fest kodierte SQL-Anweisungen ausgeführt werden, und entwickeln dann Code, der beliebige Anweisungen verarbeitet. Danach fügen wir unserem Client-Gerüst den Code für die Anfrageverarbeitung hinzu, um ein weiteres Programm zu entwickeln (`client5`), das dem `mysql`-Client ganz ähnlich ist.

Dabei gehen wir auch auf häufig auftretende Probleme ein, beispielsweise »Wie erhalte ich Informationen über die Struktur meiner Tabellen?« oder »Wie kann ich Bilder in meine Datenbank einfügen?«

Dieses Kapitel beschreibt Funktionen und Datentypen aus der Client-Bibliothek nur nach Bedarf. Eine vollständige Auflistung aller Funktionen und Typen finden Sie in Anhang F.

Die Beispielprogramme stehen online zum Download bereit, Sie brauchen sie also nicht selbst einzugeben. Anhang A beschreibt, wie Sie die Dateien herunterladen.

Wo finden Sie Beispiele?

> Häufig taucht in der MySQL-Mailing-Liste die Frage auf »Wo finde ich Beispiele zu Clients in C?« Die Antwort lautet natürlich: »In diesem Buch!« Viele Benutzer sehen gar nicht, dass die MySQL-Distribution mehrere Client-Programme enthält (unter anderem `mysql`, `mysqladmin` oder `mysqldump`), die größtenteils in C geschrieben sind. Weil die Distribution als Quellcode vorliegt, finden Sie direkt in MySQL Beispiele für Client-Code. Wenn das also noch nicht geschehen ist, sehen Sie sich die Programme im Verzeichnis `client` der Distribution an. Die MySQL-Client-Programme sind public domain und Sie können den darin enthaltenen Code für Ihre eigenen Programme nutzen.
>
> Unter den Beispielen aus diesem Kapitel und den Client-Programmen der MySQL-Distribution finden Sie sicher etwas, was Sie für Ihre eigenen Programme brauchen können. Kopieren Sie einfach ein bereits existierendes Programm und ändern Sie es nach Ihren Bedürfnissen ab. Sie sollten dieses Kapitel lesen, um die Arbeitsweise der Client-Bibliothek zu verstehen. Denken Sie jedoch daran, dass Sie nicht alles selbst neu schreiben müssen. (Sie werden bemerken, dass die Wiederverwendbarkeit von Code eines der Ziele ist, die wir uns für die Programmentwicklung in diesem Kapitel gesteckt haben.) Sie können sich viel Arbeit ersparen, indem Sie Dinge nutzen, die es bereits gibt.

6.1 Allgemeine Vorgehensweise bei der Entwicklung von Client-Programmen

Dieser Abschnitt beschreibt das Kompilieren und Linken von Programmen, die die MySQL-Client-Bibliothek nutzen. Die dabei verwendeten Befehle unterscheiden sich bei den verschiedenen Systemen, deshalb kann es vorkommen, dass Sie die hier gezeigten Befehle etwas abändern müssen. Die Beschreibung ist jedoch allgemein genug gehalten, so dass Sie sie auf fast jedes von Ihnen entwickelte Client-Programm anwenden können.

6.1.1 Grundsätzliche Systemanforderungen

Für die Entwicklung eines MySQL-Client-Programms in C brauchen Sie natürlich einen C-Compiler. Die hier gezeigten Beispiele verwenden `gcc`. Außerdem brauchen Sie neben Ihren eigenen Quelldateien Folgendes:

- die MySQL-Header-Dateien
- die MySQL Client-Bibliothek

Die MySQL-Header-Dateien und die Client-Bibliothek unterstützen die Client-Programmierung. Sie sind möglicherweise bereits auf Ihrem System installiert. Andernfalls müssen Sie sie sich beschaffen. Wurde MySQL von einer Quell- oder Programm-Distribution installiert, ist die Client-Programmier-Unterstützung möglicherweise bereits installiert. Wurde MySQL von RPM-Dateien installiert, besitzen Sie diese Unterstützung nur dann, wenn Sie die Entwickler-RPM installiert haben. Falls Sie die MySQL-Header-Dateien und die Bibliothek installieren müssen, lesen Sie in Anhang A nach, wie das geht.

6.1.2 Kompilieren und Linken des Clients

Um ein Client-Programm zu kompilieren und zu linken, müssen Sie angeben, wo sich die MySQL-Header-Dateien und die Client-Bibliothek befindet, weil diese sich normalerweise nicht an den Positionen befinden, wo der Compiler und Linker danach suchen. Angenommen, die Header-Dateien und die Client-Bibliothek befinden sich an den Positionen `/usr/local/include/mysql` und `/usr/local/lib/mysql`.

Um dem Compiler mitzuteilen, wo er die MySQL-Header-Dateien findet, übergeben Sie ihm das Argument `-I/usr/local/include/mysql`, wenn Sie eine Quelldatei zu einer Objektdatei kompilieren. Sie könnten beispielsweise die folgende Kommandozeile verwenden:

```
% gcc -c -I/usr/local/include/mysql myclient.c
```

Um dem Linker mitzuteilen, wo er die Client-Bibliothek findet und wie sie heißt, übergeben Sie ihm die Argumente `-L/usr/local/lib/mysql` und `-lmysqlclient`, wenn Sie die Objektdatei linken, um eine ausführbare Programmdatei zu erzeugen:

```
% gcc -o myclient myclient.o -L/usr/local/lib/mysql -lmysqlclient
```

Falls Ihr Client-Programm aus mehreren Dateien besteht, geben Sie im Link-Befehl alle Objektdateien an. Falls beim Linken ein Fehler auftritt, der besagt, dass die Funktion `floor()` nicht gefunden werden konnte, binden Sie die Mathematik-Bibliothek ein, indem Sie hinter dem Befehl ein `-lm` einfügen:

```
% gcc -o myclient myclient.o -L/usr/local/lib/mysql -lmysqlclient -lm
```

Möglicherweise brauchen Sie auch noch andere Bibliotheken, beispielsweise `-lsocket -lnsl` auf Solaris.

Wenn Sie zum Erstellen Ihrer Programme noch nicht `make` verwenden, sollten Sie es lernen, damit Sie nicht alle Befehle zur Programmerstellung manuell eingeben müssen. Angenommen, Sie haben das Client-Programm `myclient`, das aus zwei Quelldateien besteht, `main.c` und `aux.c`, und einer Header-Datei, `myclient.h`. Ein einfaches Makefile zur Programmerstellung könnte wie folgt aussehen:

```
CC = gcc
INCLUDES = -I/usr/local/include/mysql
LIBS = -L/usr/local/lib/mysql -lmysqlclient

all: myclient

main.o: main.c myclient.h
    $(CC) -c $(INCLUDES) main.c
aux.o: aux.c myclient.h
    $(CC) -c $(INCLUDES) aux.c

myclient: main.o aux.o
    $(CC) -o myclient main.o aux.o $(LIBS)

clean:
    rm -f myclient main.o aux.o
```

Falls Sie bei Ihrem System die Mathematik-Bibliothek linken müssen, ändern Sie den Wert von `LIBS` so ab, dass am Ende `-lm` steht:

```
LIBS = -L/usr/local/lib/mysql -lmysqlclient -lm
```

Falls Sie weitere Bibliotheken brauchen, beispielsweise `-lsocket` und `-lnsl`, fügen Sie diese ebenfalls `LIB` hinzu.

Mit Hilfe des Makefiles können Sie Ihr Programm nach jeder Änderung der Quelldateien neu erstellen, indem Sie einfach `make` eingeben. Das ist einfacher und weniger fehleranfällig als die Eingabe eines langen `gcc`-Befehls.

6.2 Client 1 – Verbindung zum Server

Unser erstes MySQL-Client-Programm ist ganz einfach: Es stellt eine Verbindung zu einem Server her, baut die Verbindung wieder ab und wird dann beendet. Das ist noch nicht sehr viel, aber Sie müssen wissen, wie das geht, bevor Sie mit einer MySQL-Datenbank arbeiten können – weil Sie eine Verbindung zum Server brauchen. Das ist eine so gebräuchliche Operation, dass Sie diesen Code in jedes von Ihnen entwickelte Client-Programm aufnehmen werden. Darüber hinaus ist die Aufgabenstellung ein guter Ausgangspunkt. Später bauen wir den Client weiter aus, damit er etwas Sinnvolles für uns erledigt.

Die Quelle für unser erstes Client-Programm, `client1`, besteht aus einer einzigen Datei: `client1.c`:

```
/* client1.c */

#include <stdio.h>
#include <mysql.h>

#define def_host_name   NULL /* Host (Standard = localhost) */
#define def_user_name   NULL /* Benutzername (Standard = Login-Name) */
#define def_password    NULL /* Kennwort (Standard = keines) */
#define def_db_name     NULL /* Datenbank (Standard = keine) */

MYSQL   *conn;              /* Zeiger auf den Verbindungs-Handle */

int
main (int argc, char *argv[])
{

    conn = mysql_init (NULL);
    mysql_real_connect (
                conn,           /* Zeiger auf den Verbindungs-Handle */
                def_host_name,  /* Host */
                def_user_name,  /* Benutzername */
                def_password,   /* Kennwort */
                def_db_name,    /* Datenbank */
                0,              /* Port (Standard verwenden) */
                NULL,           /* Socket (Standard verwenden) */
                0);             /* Flags (keines) */
    mysql_close (conn);
    exit (0);
}
```

Die Quelldatei beginnt mit der Angabe von `stdio.h` und `mysql.h`. MySQL-Clients können auch weitere Header-Dateien einbinden, aber im Allgemeinen brauchen Sie mindestens diese beiden.

Um alles so einfach wie möglich zu halten, sind die Standardwerte für Host, Benutzername, Kennwort und Datenbank im Code festgeschrieben. Später parametrisieren wir diese Werte, so dass Sie sie in Optionsdateien oder in der Kommandozeile eingeben können.

Die Funktion `main()` richtet die Verbindung zum Server ein und baut sie ab. Der Verbindungsaufbau erfolgt in zwei Schritten:

1. Rufen Sie `mysql_init()` auf, um einen Verbindungs-Handle zu erhalten. Der `MYSQL`-Datentyp ist eine Struktur mit Informationen über eine Verbindung. Variablen dieses Typs werden als Verbindungs-Handle bezeichnet. Wenn Sie

`mysql_INIT()` `NULL` übergeben, reserviert es eine `MYSQL`-Variable, initialisiert sie und gibt einen Zeiger darauf zurück.

2. Rufen Sie `mysql_real_connect()` auf, um eine Verbindung zum Server einzurichten. `mysql_real_connect()` nimmt unzählige Parameter entgegen:
   - Einen Zeiger auf einen Verbindungs-Handle. Dieser Parameter sollte nicht `NULL` sein, sondern der von `mysql_init()` zurückgegebene Wert.
   - Den Server-Host. Wenn Sie hierfür `NULL` oder den Host »`localhost`« angeben, stellt der Client unter Verwendung eines UNIX-Sockets eine Verbindung zu dem Server auf dem lokalen Host her. Wenn Sie einen Hostnamen oder eine IP-Adresse eines Hosts angeben, stellt der Client unter Verwendung einer TCP/IP-Verbindung eine Verbindung zum angegebenen Host her.
   - Unter Windows liegt ein ähnliches Verhalten vor, außer dass statt UNIX-Sockets TCP/IP-Verbindungen genutzt werden. (Unter Windows NT wird vor TCP/IP versucht, eine Named Pipe zu verwenden, falls der Host gleich `NULL` ist.)
   - Den Benutzernamen und das Kennwort. Ist der Name gleich `NULL`, sendet die Client-Bibliothek Ihren Login-Namen an den Server. Ist das Kennwort gleich `NULL`, wird kein Kennwort gesendet.
   - Die Portnummer und die Socketdatei. Sie werden als `0` und `NULL` angegeben, um der Client-Bibliothek mitzuteilen, dass sie ihre Standardwerte verwenden soll. Sind Port und Socket nicht angegeben, werden die Standards dem gewünschten Host entsprechend festgelegt. Weitere Informationen finden Sie unter der Beschreibung von `mysql_real_connect()` in Anhang F.
   - Der Flags-Wert. Er ist 0, weil wir keine speziellen Verbindungsoptionen verwenden werden. Die möglichen Optionen für diesen Parameter sind detailliert unter der Beschreibung von `mysql_real_connect()` in Anhang F beschrieben.

Um die Verbindung zu beenden, übergeben Sie `mysql_close()` einen Zeiger auf den Verbindungs-Handle. Ein Verbindungs-Handle, der durch `mysql_init()` automatisch zugewiesen wird, wird auch automatisch wieder verworfen, wenn Sie ihn `mysql_close()` übergeben, um die Verbindung zu trennen.

Um `client1` auszuprobieren, kompilieren und linken Sie es, wie in diesem Kapitel bereits beschrieben wurde, und führen es dann aus:

```
% client1
```

Das Programm richtet eine Verbindung zum Server ein, baut sie ab und wird beendet. Das ist nicht besonders aufregend, aber ein Anfang. Mehr ist es aber auch nicht, weil es zwei wichtige Mängel gibt:

- Der Client führt keine Fehlerprüfung durch, Sie können also nicht sicher sein, ob er überhaupt funktioniert!
- Die Verbindungsparameter (Hostname, Benutzername usw.) sind im Quellcode festgeschrieben. Es wäre besser, dem Benutzer die Möglichkeit zu bieten, sie zu überschreiben, indem er die Parameter in einer Optionsdatei oder auf der Kommandozeile bereitstellt.

Beide Probleme können ganz einfach gelöst werden, wie Sie in den nächsten Abschnitten noch sehen werden.

6.3 Client 2 – Einführen einer Fehlerprüfung

Unser zweiter Client ist dem ersten ganz ähnlich, ergänzt um eine Fehlerprüfung. In Programmierbüchern findet man häufig die Aussage »Die Fehlerprüfung soll dem Leser als Übung überlassen bleiben«, vermutlich weil die Fehlerprüfung so langweilig ist – sagen wir doch, wie es ist! Dennoch möchte ich betonen, dass in MySQL-Client-Programmen unbedingt eine Fehlerprüfung stattfinden und eine entsprechende Reaktion erfolgen sollte. Die Aufrufe der Client-Bibliothek geben die Statuswerte nicht grundlos zurück und Sie sollten sie nicht ignorieren. Irgendwann treten seltsame Probleme in Ihren Programmen auf, die Sie nicht lösen können, weil Sie versäumt haben, eine Fehlerprüfung auszuführen, oder die Benutzer Ihrer Programme fragen sich, warum diese sich so seltsam verhalten.

Betrachten Sie unser Programm `client1`. Wie erkennen Sie, ob es wirklich eine Verbindung zum Server eingerichtet hat? Sie stellen es fest, indem Sie im Serverprotokoll nach `Connect`- und `Quit`-Ereignissen suchen, die stattfanden, während Sie das Programm ausführten:

```
990516 21:52:14      20 Connect    paul@localhost on
                     20 Quit
```

Möglicherweise sehen Sie aber auch einen Eintrag für einen verweigerten Zugriff, `Access denied`:

```
990516 22:01:47      21 Connect    Access denied for user: 'paul@local-
                                   host'
                                   (Using password: NO)
```

Diese Meldung zeigt, dass keine Verbindung eingerichtet wurde. Leider teilt uns `client1` nicht mit, was passiert ist. Es kann einfach nicht. Es nimmt keine Fehlerprüfung vor, deshalb weiß es selbst nicht, was passiert ist. Jedenfalls sollten Sie nicht erst in das Protokoll sehen müssen, um herauszufinden, ob Sie eine Verbindung zum Server einrichten können. Dieses Problem werden wir sofort beheben.

Routinen der MySQL-Client-Bibliothek, die einen Rückgabewert erzeugen, zeigen den Erfolg oder Misserfolg auf zweierlei Arten an:

- Funktionen mit Zeigerwerten geben einen Zeiger ungleich `NULL` zurück, wenn sie erfolgreich ausgeführt werden konnten, andernfalls `NULL`. (`NULL` bedeutet in diesem Kontext »ein C-`NULL`-Zeiger«, nicht »ein MySQL-`NULL`-Spaltenwert«.)

  Von den Routinen der Client-Bibliothek, die wir bisher verwendet haben, geben `mysql_init()` und `mysql_real_connect()` beide einen Zeiger auf den Verbindungs-Handle zurück, wenn sie erfolgreich waren, andernfalls `NULL`.

- Funktionen mit ganzzahligen Werten geben normalerweise 0 zurück, wenn sie erfolgreich ausgeführt werden konnten, andernfalls einen Wert ungleich Null. Dabei ist es wichtig, auf bestimmte Werte ungleich Null abzufragen, beispielsweise -1. Es ist nicht garantiert, dass eine Funktion der Client-Bibliothek einen bestimmten Wert zurückgibt, wenn sie fehlgeschlagen ist. Es gibt immer noch älteren Code, der einen Rückgabewert fehlerhaft überprüft, beispielsweise wie folgt:

  ```
  if (mysql_XXX() == -1)        /* das ist falsch */
      fprintf (stderr, "Etwas Schlimmes ist passiert\n");
  ```

  Diese Überprüfung kann funktionieren, muss aber nicht. Das MySQL-API legt nicht fest, dass eine Fehlerrückgabe ungleich Null einen bestimmten Wert hat, außer dass sie (offensichtlich) nicht Null ist. Die Überprüfung sollte also wie folgt aussehen:

  ```
  if (mysql_XXX())            /* das ist richtig */
      fprintf (stderr, "Etwas Schlimmes ist passiert\n");
  ```

  oder so:

  ```
  if (mysql_XXX() != 0)        /* das ist richtig */
      fprintf (stderr, "Etwas Schlimmes ist passiert\n");
  ```

Diese beiden Überprüfungen sind äquivalent. Im Quellcode von MySQL finden Sie im Allgemeinen die erste Form der Überprüfung, die kürzer zu schreiben ist.

Nicht jeder API-Aufruf erzeugt einen Rückgabewert. Die andere Client-Routine, die wir verwendet haben, `mysql_close()`, gibt keinen Wert zurück. (Wie könnte sie fehlschlagen? Und was würde es ausmachen? Sie brauchen die Verbindung ja ohnehin nicht mehr.)

Es gibt zwei sehr praktische Aufrufe im API, falls der Aufruf einer Client-Bibliothek fehlschlägt und Sie mehr Informationen brauchen. `mysql_error()` gibt eine Zeichenkette mit der Fehlermeldung zurück, `mysql_errno()` einen numerischen Fehlercode. Sie sollten sie unmittelbar nach dem Auftreten des Fehlers aufrufen, weil sonst in der Zwischenzeit ein anderer API-Aufruf einen Status zurückgibt und sich die mit `mysql_error()` oder `mysql_errno()` ermittelte Information auf diesen Aufruf bezieht.

Im Allgemeinen wird ein Benutzer eines Programms die Fehlermeldung als aussagekräftiger betrachten als den Fehlercode. Wenn Sie nur eines von beiden bereitstellen wollen, sollten Sie sich für die Meldung entscheiden. Der Vollständigkeit halber zeigen die Beispiele in diesem Buch beide Werte an.

Nach diesen Vorbesprechungen entwickeln wir unseren zweiten Client, `client2`. Er ist ähnlich `client1`, besitzt aber korrekten Code für die Fehlerprüfung. Die Quelldatei, `client2.c`, sieht wie folgt aus:

```
/* client2.c */

#include <stdio.h>
#include <mysql.h>

#define def_host_name   NULL /* Host (Standard = localhost) */
#define def_user_name   NULL /* Benutzername(Standard = Login-Name) */
#define def_password    NULL /* Kennwort (Standard = keines) */
#define def_db_name     NULL /* Datenbank (Standard = keine) */

MYSQL   *conn;              /* Zeiger auf Verbindungs-Handle */

int
main (int argc, char *argv[])
{
    conn = mysql_init (NULL);
    if (conn == NULL)
    {
        fprintf (stderr, "mysql_init() failed (probably out of memory)
        \n");
        exit (1);
    }
    if (mysql_real_connect (
                conn,           /* Zeiger auf Verbindungs-Handle */
                def_host_name,  /* Host */
                def_user_name,  /* Benutzername */
                def_password,   /* Kennwort */
                def_db_name,    /* Datenbank */
                0,              /* Port (Standard verwenden) */
                NULL,           /* Socket (Standard verwenden) */
                0)              /* Flags (keine) */
            == NULL)
    {
        fprintf (stderr, "mysql_real_connect() failed:\nError %u (%s)
        \n",
                            mysql_errno (conn), mysql_error (conn));
        exit (1);
    }
    mysql_close (conn);
    exit (0);
}
```

Die Logik für die Fehlerprüfung basiert auf der Tatsache, dass `mysql_init()` und auch `mysql_real_connect()` `NULL` zurückgeben, wenn sie fehlschlagen. Beachten Sie, dass das Programm zwar den Rückgabewert von `mysql_init()` überprüft, aber beim Fehlschlagen keine Funktion aufgerufen wird, die den Fehler meldet. Der Verbindungs-Handle muss nämlich nicht unbedingt sinnvolle Informationen enthalten, wenn `mysql_init()` fehlschlägt. Schlägt dagegen `mysql_real_connect()` fehl, zeigt der Verbindungs-Handle keine gültige Verbindung an, sondern enthält Fehlerinformationen, die den Funktionen zur Fehlermeldung übergeben werden können. (Übergeben Sie den Handle jedoch an keine anderen Client-Routinen! Weil sie im Allgemeinen eine gültige Verbindung voraussetzen, könnte Ihr Programm abstürzen.)

Kompilieren und linken Sie `client2` und führen es dann aus:

```
% client2
```

Falls `client2` keine Ausgabe erzeugt, wurde die Verbindung erfolgreich aufgebaut. Es könnte jedoch die folgende Ausgabe erscheinen:

```
% client2
mysql_real_connect() failed:
Error 1045 (Access denied for user: 'paul@localhost' (Using password:
NO))
```

Diese Ausgabe zeigt an, dass keine Verbindung eingerichtet wurde, und teilt Ihnen auch den Grund dafür mit. Außerdem bedeutet es, dass unser erstes Programm, `client1`, niemals eine Verbindung zum Server einrichten konnte! (Schließlich verwendete `client1` dieselben Verbindungsparameter.) Wir wussten es nur nicht, weil `client1` keine Fehlerprüfung ausgeführt hat. `client2` führt eine Fehlerprüfung durch und teilt uns mit, ob Fehler aufgetreten sind. Deshalb sollten Sie die Rückgabewerte von API-Funktionen immer auswerten.

Die Fragen und Antworten auf der MySQL-Mailing-Liste haben häufig mit der Fehlerprüfung zu tun. Typische Fragen sind »Warum stürzt mein Programm ab, wenn ich diese oder jene Anfrage ausführe?« oder »Warum gibt meine Anfrage nichts zurück?« Häufig wird in dem betreffenden Programm nicht geprüft, ob die Verbindung erfolgreich eingerichtet werden konnte, bevor die Anfrage abgesetzt wurde, oder ob der Server erfolgreich ausgeführt wird, bevor man versucht, Ergebnisse zu ermitteln. Glauben Sie nicht, dass alle Aufrufe der Client-Bibliothek erfolgreich seien.

Die restlichen Beispiele in diesem Kapitel führen eine Fehlerprüfung aus und Sie sollten das auch tun. Es scheint einen zusätzlichen Aufwand zu bedeuten, aber letztlich sparen Sie damit Zeit, wenn Probleme auftreten. Diesen Ansatz der Fehlerprüfung werde ich auch in Kapitel 7 verwenden.

Angenommen, Sie sehen bei der Ausführung des Programms `client2` die Meldung `Access denied`. Wie können Sie das Problem lösen? Eine Möglichkeit wäre, die `#define`-Zeilen für den Hostnamen, den Benutzernamen und das Kennwort so zu

ändern, dass Ihnen die Werte den Zugriff auf Ihren Server erlauben. Damit könnten Sie zumindest eine Verbindung einrichten. Die Werte sind dann aber immer noch in Ihrem Programm festgeschrieben. Ich rate von dieser Vorgehensweise ab, insbesondere in Hinblick auf den Wert für das Kennwort. Sie denken vielleicht, das Kennwort sei verborgen, wenn Sie Ihr Programm in ein binäres Format kompilieren, aber wenn jemand den Befehl `strings` für Ihr Programm ausführen kann, sieht er auch das Kennwort. (Abgesehen davon, dass jeder, der den Quellcode Ihres Programms lesen kann, das Kennwort ohne jeden weiteren Aufwand erfährt.)

Mehr über das Zugriffsproblem erfahren Sie im Abschnitt »Client 4 – Verbindungsparameter zur Laufzeit übergeben«. Zuerst will ich Ihnen jedoch andere Methoden zeigen, Ihren Verbindungscode zu schreiben.

6.4 Client 3 – Modularer Verbindungscode

In unserem dritten Client, `client3`, machen wir den Code für den Aufbau und den Abbau der Verbindung modularer, indem wir ihn in die Funktionen `do_connect()` und `do_disconnect()` kapseln, die von anderen Client-Programmen ebenfalls genutzt werden können. Das ist eine Alternative dazu, den Verbindungscode direkt in die Funktion `main()` aufzunehmen. Sie ist immer dann sinnvoll, wenn in vielen Anwendungen immer wieder derselbe Code benötigt wird. Sie bringen ihn in einer Funktion unter, auf die Sie von mehreren Programmen aus zugreifen können, statt ihn überall neu zu schreiben. Wenn Sie im Code einen Fehler korrigieren oder die Funktion verbessern, brauchen Sie diese Änderung nur ein einziges Mal vorzunehmen und alle Programme, die die Funktion nutzen, werden sofort die korrigierte Version verwenden, nachdem sie einfach nur neu kompiliert wurden. Einige Client-Programme sind darauf ausgelegt, dass sie während der Ausführung mehrmals Verbindungen aufbauen und wieder abbauen. Es ist viel einfacher, einen solchen Client zu entwickeln, wenn Sie den Code modular aufbauen.

Die Kapselung erfolgt nach der folgenden Strategie:

1. Schreiben Sie den gemeinsam zu nutzenden Code in Hüllfunktionen in einer separaten Quelldatei, `common.c`.
2. Stellen Sie eine Header-Datei bereit, `common.h`., die Prototypen für die gemeinsam zu nutzenden Routinen enthält.
3. Binden Sie `common.h` in Client-Quelldateien ein, die die gemeinsamen Routinen nutzen sollen.
4. Kompilieren Sie die gemeinsame Quelle, um eine Objektdatei zu erhalten.
5. Linken Sie diese gemeinsam zu nutzende Objektdatei zu Ihrem Client-Programm.

Auf diese Weise wollen wir jetzt die Funktionen `do_connect()` und `do_disconnect()` aufbauen.

`do_connect()` ersetzt die Aufrufe von `mysql_init()` und `mysql_real_connect()` sowie den Code für die Fehlermeldung. Sie rufen es genau wie `mysql_real_connect()` auf, außer dass Sie hier keinen Verbindungs-Handle übergeben. Stattdessen alloziert und initialisiert `do_connect()` den Handle selbst und gibt nach dem Verbindungsaufbau einen Zeiger darauf zurück. Schlägt `do_connect()` fehl, gibt es nach der Ausgabe einer Fehlermeldung `NULL` zurück. (Auf diese Weise kann jedes Programm, das `do_connect()` aufruft und den Rückgabewert `NULL` erhält, einfach beendet werden, ohne selbst eine Meldung ausgeben zu müssen.)

`do_disconnect()` nimmt einen Zeiger auf den Verbindungs-Handle entgegen und ruft `mysql_close()` auf.

Hier ist der Code für `common.c`:

```
#include <stdio.h>
#include <mysql.h>
#include "common.h"

MYSQL *
do_connect (char *host_name, char *user_name, char *password, char
            *db_name, unsigned int port_num, char *socket_name,
            unsigned int flags)
{
MYSQL   *conn;  /* Zeiger auf Verbindungs-Handle */

    conn = mysql_init (NULL);   /* Verb.handle allozieren & initial. */
    if (conn == NULL)
    {
        fprintf (stderr, "mysql_init() failed\n");
        return (NULL);
    }
    if (mysql_real_connect (conn, host_name, user_name, password,
                        db_name, port_num, socket_name, flags) == NULL)
    {
        fprintf (stderr, "mysql_real_connect() failed:\nError %u (%s)
                 \n",
                            mysql_errno (conn), mysql_error (conn));
        return (NULL);
    }
    return (conn);          /* Verbindung ist eingerichtet */
}

void
do_disconnect (MYSQL *conn)
{
    mysql_close (conn);
}
```

common.h deklariert die Prototypen für die Routinen in common.c:

```
MYSQL *
do_connect (char *host_name, char *user_name, char *password, char
            *db_name,unsigned int port_num, char *socket_name, unsigned
            int flags);
void
do_disconnect (MYSQL *conn);
```

Um auf diese gemeinsam zu nutzenden Routinen zuzugreifen, binden Sie common.h in Ihre Quelldateien ein. Beachten Sie, dass common.c auch common.h beinhaltet. Auf diese Weise erhalten Sie sofort eine Compilerwarnung, wenn die Funktionsdefinition in common.c nicht mit den Deklarationen in der Header-Datei übereinstimmt. Wenn Sie eine Aufruf-Folge in common.c ändern, ohne die entsprechende Anpassung in common.h vorzunehmen, warnt der Compiler ebenfalls, wenn Sie common.c neu kompilieren.

Sie fragen sich vielleicht, warum jemand eine Hüllfunktion wie do_disconnect() verwenden sollte, die so wenig tut. do_diconnect() und mysql_close() sind tatsächlich äquivalent. Aber stellen Sie sich vor, Sie wollen beim Verbindungsabbau irgendwann irgendwelche Aufräumarbeiten ausführen. Durch den Aufruf einer Hüllfunktion, über die Sie volle Kontrolle haben, können Sie die Hülle so abändern, dass sie das tut, was Sie von ihr erwarten, und die Änderung wird in jedem fortan ausgeführten Verbindungsabbau berücksichtigt. Das ist nicht möglich, wenn Sie mysql_close() direkt aufrufen.

Ich habe bereits erklärt, dass es sinnvoll ist, häufig genutzten Code zu modularisieren, indem man ihn in eine Funktion einkapselt, die von mehreren Programmen oder von mehreren Positionen innerhalb eines einzelnen Programms aus genutzt wird. Der vorige Abschnitt hat eine Begründung dafür geliefert, warum das sinnvoll ist, und die beiden folgenden Beispiele verdeutlichen das Ganze weiter.

- **Beispiel 1**. In MySQL-Verisonen vor 3.22 unterschied sich der Aufruf von mysql_real_connect() etwas von dem der jetzigen Version: Es gab keinen Parameter für den Datenbanknamen. Es ist nicht möglich, do_connect() zusammen mit einer älteren MySQL Client-Bibliothek einzusetzen. Sie können do_connect() jedoch so abändern, dass es auch für Versionen vor 3.22 genutzt werden kann. Das bedeutet, durch eine Abänderung von do_connect() können Sie die Portierbarkeit aller Programme verbessern, die es nutzen. Nehmen Sie dagegen den Verbindungscode in jeden Client auf, müssen Sie jeden davon einzeln abändern.

  Um do_connect() so zu ändern, dass es auch mit der älteren Version von mysql_real_connect() zurechtkommt, verwenden Sie das Makro MYSQL_VERSION_ID, das die aktuelle MySQL-Versionsnummer enthält. Das geänderte do_connect() liest den Wert von MYSQL_VERSION_ID und verwendet dann die entsprechende Version von mysql_real_connect():

```
MYSQL *
do_connect (char *host_name, char *user_name, char *password, char
            *db_name, unsigned int port_num, char *socket_name, unsigned
            int flags)
{
MYSQL   *conn;  /* Zeiger auf Verbindungs-Handle */

    conn = mysql_init (NULL);   /* Verbindungshandle allozieren & initia
                                lisieren */
    if (conn == NULL)
    {
        fprintf (stderr, "mysql_init() failed\n");
        return (NULL);
    }
#if defined(MYSQL_VERSION_ID) && MYSQL_VERSION_ID >= 32200  /* 3.22 und
 höher */
    if (mysql_real_connect (conn, host_name, user_name, password,
                    db_name, port_num, socket_name, flags) == NULL)
    {
        fprintf (stderr, "mysql_real_connect() failed:\nError %u (%s)\n",
                          mysql_errno (conn), mysql_error (conn));
        return (NULL);
    }
#else                           /* vor 3.22 */
    if (mysql_real_connect (conn, host_name, user_name, password,
                    port_num, socket_name, flags) == NULL)
    {
        fprintf (stderr, "mysql_real_connect() fehlgeschlagen:\nError
        %u (%s)\n",
                          mysql_errno (conn), mysql_error (conn));
        return (NULL);
    }
    if (db_name != NULL)        /* Effekt des Param. db_name simulieren */
    {
        if (mysql_select_db (conn, db_name) != 0)
        {
            fprintf (stderr, "mysql_select_db() fehlgeschlagen:\nError
            %u (%s)\n",
                          mysql_errno (conn), mysql_error (conn));
            mysql_close (conn);
            return (NULL);
        }
    }
#endif
    return (conn);          /* Verbindung eingerichtet */
}
```

Die geänderte Version von `do_connect()` ist bis auf zwei Punkte identisch mit der vorhergehenden Version:

- Es übergibt keinen `db_name`-Parameter an die ältere Form von `mysql_real_connect()`, weil es diesen Parameter dort nicht gibt.
- Falls der Datenbankname ungleich `NULL` ist, ruft `do_connect()` `mysql_select_db()` auf, um die angegebene Datenbank zur aktuellen Datenbank zu machen. (Das simuliert den Effekt des fehlenden Parameters `db_name`.) Falls die Datenbank nicht ausgewählt werden kann, gibt `do_connect()` eine Fehlermeldung aus, schließt die Verbindung und gibt `NULL` zurück, um zu zeigen, dass die Ausführung nicht erfolgreich war.

- **Beispiel 2**. Dieses Beispiel basiert auf den Änderungen, die im ersten Beispiel an `do_connect()` vorgenommen wurden. Diese Änderungen führen zu drei verschiedenen Aufrufmengen der Fehlerfunktionen `mysql_errno()` und `mysql_error()` und es ist relativ müßig, diesen Code jedes Mal neu zu schreiben, um ein Problem kundzutun. Darüber hinaus ist der Code für die Fehlermeldung schwer zu überblicken und zu lesen. Einfacher ist es, etwas wie das Folgende zu lesen:

  ```
  print_error (conn, "mysql_real_connect() failed");
  ```

  Deshalb wollen wir die Fehleranzeige in die Funktion `print_error()` einkapseln. Wir können sie so schreiben, dass sie auch dann reagiert, wenn `conn` gleich `NULL` ist. Auf diese Weise können wir `print_error()` aufrufen, wenn `mysql_init()` fehlschlägt, und wir haben keine Mixtur aus unterschiedlichen Aufrufen (einige für `fprintf()` und andere für `print_error()`).

  Ich höre Ihre Argumente: »Aber Sie *müssen* doch nicht jedes Mal beide Fehlerfunktionen aufrufen, wenn Sie auf einen Fehler hinweisen wollen, damit machen Sie Ihren Code nur bewusst unübersichtlich, um Ihr Kapselungsbeispiel besser aussehen zu lassen. Und Sie müssten nie diesen gesamten Code zur Fehlerausgabe immer wieder neu schreiben; Sie würden ihn einmal schreiben und ihn dann bei Bedarf einfach kopieren und einfügen.« Das mag sein, aber ich habe die folgenden Gegenargumente:

  - Selbst wenn Sie Kopieren&Einfügen verwenden, ist es einfacher, wenn die entsprechenden Codeabschnitte kürzer sind.
  - Egal, ob Sie für jeden Fehler beide Fehlerfunktionen aufrufen: Wenn Sie den gesamten Code in der langen Form schreiben, sind Sie irgendwann versucht, Abkürzungen zu schaffen, und die werden bei der Anzeige der Fehler inkonsistent. Wenn Sie den Code für die Fehleranzeige in eine Hüllfunktion einbetten, die einfach aufzurufen ist, ist diese Versuchung weniger groß und Ihr Code bleibt konsistenter.

- Wenn Sie das Format Ihrer Fehlermeldungen irgendwann ändern wollen, ist es viel einfacher, das an zentraler Stelle zu erledigen, als mehrfach im gesamten Programm. Und wenn Sie beschließen, die Fehlermeldungen in eine Protokolldatei zu schreiben, statt (oder zusätzlich) auf `stderr`, ist das einfacher, wenn Sie dafür nur `print_error()` ändern müssen. Dieser Ansatz ist weniger fehleranfällig und mildert ebenfalls die Versuchung, das Ganze nur halbherzig auszuführen, was Inkonsistenz zur Folge hätte.
- Wenn Sie zum Testen Ihrer Programme einen Debugger verwenden, ist es praktisch, in der Funktion zur Fehleranzeige einen Haltepunkt zu setzen, so dass das Programm in den Debugger wechselt, sobald es eine Fehlerbedingung erkennt.

Und hier unsere Funktion zur Fehleranzeige, `print_error()`:

```
void
print_error (MYSQL *conn, char *message)
{
    fprintf (stderr, "%s\n", message);
    if (conn != NULL)
    {
        fprintf (stderr, "Fehler %u (%s)\n",
                mysql_errno (conn), mysql_error (conn));
    }
}
```

`print_error()` befindet sich in `common.c`, deshalb fügen wir `common.h`, einen Prototyp dafür, hinzu:

```
void
print_error (MYSQL *conn, char *message);
```

Jetzt kann `do_connect()` so geändert werden, dass es `print_error()` nutzt:

```
MYSQL *
do_connect (char *host_name, char *user_name, char *password, char
            *db_name, unsigned int port_num, char *socket_name,
            unsigned int flags)
{
MYSQL   *conn;  /* Zeiger auf Verbindungs-Handle */

    conn = mysql_init (NULL);  /* Verb.handle allozieren & init. */
    if (conn == NULL)
    {
        print_error (NULL, "mysql_init() fehlgeschlagen (möglicherweise
                           zu wenig Speicher)");
        return (NULL);
    }
```

```
#if defined(MYSQL_VERSION_ID) && MYSQL_VERSION_ID >= 32200  /* 3.22 und
                                                              höher */
    if (mysql_real_connect (conn, host_name, user_name, password,
                    db_name, port_num, socket_name, flags) == NULL)
    {
        print_error (conn, "mysql_real_connect() fehlgeschlagen");
        return (NULL);
    }
#else                           /* vor 3.22 */
    if (mysql_real_connect (conn, host_name, user_name, password,
                    port_num, socket_name, flags) == NULL)
    {
        print_error (conn, "mysql_real_connect() fehlgeschlagen");
        return (NULL);
    }
    if (db_name != NULL)        /* Effekt des Parameters db_name
                                simulieren */
    {
        if (mysql_select_db (conn, db_name) != 0)
        {
            print_error (conn, "mysql_select_db() fehlgeschlagen");
            mysql_close (conn);
            return (NULL);
        }
    }
#endif
    return (conn);          /* Verbindung ist eingerichtet */
}
```

Unsere Haupt-Quelldatei, `client3.c`, ist ähnlich `client2.c`, aber der gesamte Code für Auf- und Abbau der Verbindung wurde entfernt und durch Aufrufe der Hüllfunktion ersetzt:

```
/* client3.c */

#include <stdio.h>
#include <mysql.h>
#include "common.h"

#define def_host_name   NULL /* Host (Standard = localhost) */
#define def_user_name   NULL /* Benutzername (Standard = Login-Name) */
#define def_password    NULL /* Kennwort (Standard = keines) */
#define def_port_num    0    /* Standardport verwenden */
#define def_socket_name NULL /* Standard-Socketname verwenden */
#define def_db_name     NULL /* Datenbank (Standard = keine) */

MYSQL   *conn;  /* Zeiger auf Verbindungs-Handle */
```

```
int
main (int argc, char *argv[])
{
    conn = do_connect (def_host_name, def_user_name, def_password, def_
    db_name,
                                    def_port_num, def_socket_name, 0);
    if (conn == NULL)
        exit (1);

    /* hier passiert die eigentliche Arbeit */

    do_disconnect (conn);
    exit (0);
}
```

6.5 Client 4 – Verbindungsparameter zur Laufzeit übergeben

Jetzt haben wir ihn, unseren einfach veränderbaren und für den Fehlerfall gerüsteten Verbindungscode, und sollten überlegen, wie wir statt der `NULL`-Verbindungsparameter etwas Gefälligeres verwenden könnten – beispielsweise könnten wir es dem Benutzer überlassen, diese Werte zur Laufzeit einzugeben.

Der vorherige Client, `client3`, weist noch ein wesentliches Defizit auf, weil die Verbindungsparameter dort im Code festgeschrieben sind. Um einen dieser Werte zu ändern, müssen Sie die Quelldatei ändern und neu kompilieren. Das ist nicht sehr praktisch, insbesondere wenn Sie Ihr Programm anderen Benutzern zur Verfügung stellen wollen.

Häufig werden Verbindungsparameter zur Laufzeit mit Hilfe von Kommandozeilenoptionen übergeben. Die Programme der MySQL-Distribution unterstützen zwei Methoden zur Übergabe von Verbindungsparametern, wie in Tabelle 6.1 beschrieben.

Parameter	Kurzform	Langform
Hostname	`-h host_name`	`--host=host_name`
Benutzername	`-u user_name`	`--user=user_name`
Kennwort	`-p or -pyour_password`	`--password oder --password=your_password`
Portnummer	`-P port_num`	`--port=port_num`
Socketname	`-S socket_name`	`--socket=socket_name`

Tab. 6.1: Standard-Kommandozeilenoptionen in MySQL

Der Konsistenz mit den Standard-MySQL-Clients wegen soll unser Client dieselben Formate unterstützen. Das ist ganz einfach, weil die Client-Bibliothek eine Funktion zur Auswertung der Optionen enthält.

Darüber hinaus bietet unser Client die Möglichkeit, Informationen aus Optionsdateien zu ziehen. Damit können Sie Verbindungsparameter in `~/.my.cnf` ablegen (d.h. in der Datei `.my.cnf` in Ihrem Basisverzeichnis), so dass es nicht mehr nötig ist, sie in der Kommandozeile einzugeben. Die Client-Bibliothek macht es einfach, nach MySQL-Optionsdateien zu suchen und alle relevanten Werte daraus zu beziehen. Durch ein paar zusätzliche Zeilen machen Sie Ihren Code optionsdateifähig – und das, ohne das Rad neu erfinden zu müssen. Die Syntax für Optionsdateien ist in Anhang E beschrieben.

6.5.1 Zugriff auf den Inhalt von Optionsdateien

Mit Hilfe der Funktion `load_defaults()` lesen Sie Verbindungsparameterwerte aus Optionsdateien. `load_defaults()` sucht nach Optionsdateien, wertet ihren Inhalt aus, um Optionsgruppen zu finden, an denen Sie interessiert sind, und formuliert den Argumentvektor (das Array `argv[]`) Ihres Programms so um, dass die Information aus diesen Gruppen in Form von Kommandozeilenoptionen am Anfang von `argv[]` steht. Auf diese Weise entsteht der Eindruck, als wären die Optionen in der Kommandozeile eingegeben worden. Wenn Sie die Befehlsoptionen auswerten, erhalten Sie die Verbindungsparameter als Teil Ihrer normalen Schleife zur Optionsauswertung. Die Optionen werden am Anfang von `argv[]` und nicht am Ende eingetragen, so dass diese später erscheinen als alle von `load_defaults()` übergebenen Optionen (und diese deshalb überschreiben), falls wirklich Verbindungsparameter in der Kommandozeile angegeben werden.

Hier haben wir ein kleines Programm, `show_argv`, das demonstriert, wie `load_defaults()` verwendet wird und wie damit Ihr Argumentvektor verändert wird:

```
/* show_argv.c */

#include <stdio.h>
#include <mysql.h>

char *groups[] = { "client", NULL };

int
main (int argc, char *argv[])
{
int i;

    my_init ();

    printf ("Ursprünglicher Argumentvektor:\n");
    for (i = 0; i < argc; i++)
        printf ("arg %d: %s\n", i, argv[i]);
```

```
    load_defaults ("my", groups, &argc, &argv);

    printf ("Geänderter Argumentvektor:\n");
    for (i = 0; i < argc; i++)
        printf ("arg %d: %s\n", i, argv[i]);

    exit (0);
}
```

Der Code für die Verarbeitung der Optionsdatei beinhaltet die folgenden Dinge:

- `groups[]` ist ein Zeichenketten-Array, das angibt, an welchen Optionsdateigruppen Sie interessiert sind. Für Client-Programme geben Sie immer mindestens »`client`« ein (für die Gruppe `[client]`). Das letzte Element des Arrays muss `NULL` sein.
- `my_init()` ist eine Initialisierungsroutine, die verschiedene Setup-Operationen erledigt, die für `load_defaults()` erforderlich sind.
- `load_defaults()` nimmt vier Argumente entgegen: das Präfix für Ihre Optionsdateien (das sollte immer »my« sein), das Array mit den gewünschten Optionsgruppen sowie die Adressen des Argumentzählers und des Argumentvektors Ihres Programms. Übergeben Sie hier nicht die Werte für den Zähler und den Vektor, sondern ihre Adressen, weil `load_defaults()` die Werte ändert. Beachten Sie insbesondere, dass `argv` zwar ein Zeiger ist, Sie aber dennoch `&argv` übergeben, die Adresse dieses Zeigers.

`show_argv` gibt seine Argumente zweimal aus – zuerst so, wie Sie sie auf der Kommandozeile eingegeben haben, und dann so, wie sie von `load_defaults()` abgeändert wurden. Um die Arbeitsweise von `load_defaults()` nachvollziehen zu können, sollten Sie die Datei `.my.cnf` in Ihrem Basisverzeichnis haben, in dem sich Einstellungen für die Gruppe [`client`] befinden. Angenommen, `.my.cnf` sieht wie folgt aus:

```
[client]
user=paul
password=secret
host=some_host
```

In diesem Fall erzeugt die Ausführung von `show_argv` die folgende Ausgabe:

```
% show_argv a b
Ursprünglicher Argumentvektor:
arg 0: show_argv
arg 1: a
arg 2: b
Geänderter Argumentvektor:
arg 0: show_argv
arg 1: --user=paul
```

```
arg 2: --password=secret
arg 3: --host=some_host
arg 4: a
arg 5: b
```

Möglicherweise zeigt die Ausgabe von `show_argv` Optionen, die sich weder in der Kommandozeile noch in Ihrer `~/.my.cnf`-Datei befinden. Möglicherweise wurden sie in einer systemübergreifenden Optionsdatei abgelegt. `load_defaults()` sucht nach `/etc/my.cnf` und der Datei `my.cnf` im MySQL-Datenverzeichnis, bevor es die Datei `.my.cnf` in Ihrem Basisverzeichnis liest. (Unter Windows sucht `load_defaults()` nach `C:\my.cnf`, `C:\mysql\data\my.cnf` und `my.ini` in Ihrem Windows-Systemverzeichnis.)

Client-Programme, die `load_defaults()` verwenden, geben in der Optionsgruppenliste fast immer »`client`« an (so dass Sie alle allgemeinen Client-Einstellungen aus Optionsdateien beziehen können), aber Sie können auch Werte anfordern, die für Ihr eigenes Programm spezifisch sind. Ändern Sie einfach die Zeile

```
char *groups[] = { "client", NULL };
```

wie folgt:

```
char *groups[] = { "show_argv", "client", NULL };
```

Dann können Sie Ihrer Datei `~/.my.cnf` die Gruppe [`show_argv`] hinzufügen:

```
[client]
user=paul
password=secret
host=some_host

[show_argv]
host=other_host
```

Nach diesen Änderungen erzeugt der Aufruf von `show_argv` ein anderes Ergebnis:

```
% show_argv a b
Ursprünglicher Argumentvektor:
arg 0: show_argv
arg 1: a
arg 2: b
Geänderter Argumentvektor:
arg 0: show_argv
arg 1: --user=paul
arg 2: --password=secret
arg 3: --host=some_host
arg 4: --host=other_host
arg 5: a
arg 6: b
```

Die Reihenfolge, in der Optionswerte im Argument-Array erscheinen, wird durch die Reihenfolge bestimmt, in der sie in Ihrer Optionsdatei aufgelistet sind, nicht durch die Reihenfolge, in der Ihre Optionsgruppen im Array `groups[]` aufgelistet sind. Das bedeutet, dass Sie in Ihrer Optionsdatei programmspezifische Gruppen hinter der Gruppe [`client`] angeben können. Wenn Sie eine Option in beiden Gruppen angeben, hat der programmspezifische Wert Priorität. Das sehen Sie auch in dem oben gezeigten Beispiel: Die Option `host` wurde in den Gruppen [`client`] und [`show_argv`] angegeben, aber weil [`show_argv`] die letzte Gruppe in der Optionsdatei ist, erscheint ihre `host`-Einstellung im Argumentvektor weiter hinten und hat somit Priorität.

`load_defaults()` übernimmt keine Werte aus Ihren Umgebungseinstellungen. Wenn Sie die Werte von Umgebungsvariablen wie `MYSQL_TCP_PORT` oder `MYSQL_UNIX_PORT` verwenden wollen, müssen Sie das mit Hilfe von `getenv()` selbst bewerkstelligen. Ich werde unseren Clients diese Möglichkeit nicht hinzufügen, aber das folgende Beispiel zeigt, wie die Werte einiger Standard-Umgebungsvariablen für MySQL ausgewertet werden können:

```
extern char *getenv();
char *p;
int port_num;
char *socket_name;

if ((p = getenv ("MYSQL_TCP_PORT")) != NULL)
    port_num = atoi (p);
if ((p = getenv ("MYSQL_UNIX_PORT")) != NULL)
    socket_name = p;
```

In den Standard-Clients von MySQL haben die Umgebungsvariablenwerte eine geringere Priorität als die Werte aus Optionsdateien oder von der Kommandozeile. Wenn Sie Umgebungsvariablen auswerten und dieser Konvention gehorchen wollen, werten Sie sie aus, bevor Sie `load_defaults()` aufrufen oder Kommandozeilenoptionen verarbeiten.

6.5.2 Auswerten von Kommandozeilenargumenten

Jetzt können wir alle Verbindungsparameter in den Argumentvektor laden, aber wir brauchen noch eine Methode, den Vektor auszuwerten. Dafür gibt es die Funktion `getopt_long()`.

`getopt_long()` ist in der Client-Bibliothek von MySQL enthalten, Sie haben also Zugriff darauf, wenn Sie diese Bibliothek in Ihr Programm einbinden. In Ihre Quelldatei binden Sie die Header-Datei `getopt.h` ein. Sie können diese Header-Datei aus dem `include`-Verzeichnis der MySQL-Quellcode-Distribution in das Verzeichnis kopieren, in dem Sie Ihr Client-Programm entwickeln.

Das folgende Programm, `show_param`, verwendet `load_defaults()`, um Optionsdateien zu lesen, und ruft dann `getopt_long()` auf, um den Argumentvektor auszuwerten. `show_param` zeigt, was in jeder Phase der Argumentverarbeitung passiert:

1. Einrichten von Standardwerten für Hostname, Benutzername und Kennwort
2. Ausgabe der ursprünglichen Verbindungsparameter und Argumentvektorwerte
3. Aufruf von `load_defaults()`, um den Argumentvektor neu zu formulieren, so dass der Inhalt der Optionsdatei berücksichtigt wird, und Ausgabe des resultierenden Vektors
4. Aufruf von `getopt_long()` zur Verarbeitung des Argumentvektors und Ausgabe der resultierenden Parameterwerte und des restlichen Argumentvektorinhalts

`show_param` erlaubt Ihnen, mit den verschiedenen Methoden zur Übergabe von Verbindungsparametern zu experimentieren (egal ob in Optionsdateien oder von der Kommandozeile) und im Ergebnis anzuzeigen, welche Werte für den Verbindungsaufbau verwendet würden. Mit `show_param` können Sie ein Gefühl dafür entwickeln, was in unserem nächsten Client-Programm passiert, wenn wir diesen Code zur Parameterverarbeitung in unsere Verbindungsfunktion aufnehmen, `do_connect()`.

Und hier das Programm `show_param.c`:

```
/* show_param.c */

#include <stdio.h>
#include <stdlib.h> /* wird für atoi() benötigt */
#include "getopt.h"

char *groups[] = { "client", NULL };

struct option long_options[] =
{
    {"Host",     required_argument, NULL, 'h'},
    {"Benutzer",     required_argument, NULL, 'u'},
    {"Kennwort", optional_argument, NULL, 'p'},
    {"Port",     required_argument, NULL, 'P'},
    {"Socket",   required_argument, NULL, 'S'},
    { 0, 0, 0, 0 }
};

int
main (int argc, char *argv[])
{
```

```
char *host_name = NULL;
char *user_name = NULL;
char *password = NULL;
unsigned int port_num = 0;
char *socket_name = NULL;
int i;
int c, option_index;

    my_init ();

    printf ("Ursprüngliche Verbindungsparameter:\n");
    printf ("Hostname: %s\n", host_name ? host_name : "(null)");
    printf ("Benutzername: %s\n", user_name ? user_name : "(null)");
    printf ("Kennwort: %s\n", password ? password : "(null)");
    printf ("Portnummer: %u\n", port_num);
    printf ("Socketname: %s\n", socket_name ? socket_name : "(null)");

    printf ("Ursprünglicher Argumentvektor:\n");
    for (i = 0; i < argc; i++)
        printf ("arg %d: %s\n", i, argv[i]);

    load_defaults ("my", groups, &argc, &argv);

    printf ("Geänderter Argumentvektor nach load_defaults():\n");
    for (i = 0; i < argc; i++)
        printf ("arg %d: %s\n", i, argv[i]);

    while ((c = getopt_long (argc, argv, "h:p::u:P:S:", long_options,
                    &option_index)) != EOF)
    {
        switch (c)
        {
        case 'h':
            host_name = optarg;
            break;
        case 'u':
            user_name = optarg;
            break;
        case 'p':
            password = optarg;
            break;
        case 'P':
            port_num = (unsigned int) atoi (optarg);
            break;
        case 'S':
            socket_name = optarg;
            break;
```

```
            }
        }

        argc -= optind; /* Übergabe nach den Argumenten, die von */
        argv += optind; /* getopt_long() verarbeitet wurden */

        printf ("Verbindungsparameter nach getopt_long():\n");
        printf ("Hostname: %s\n", host_name ? host_name : "(null)");
        printf ("Benutzername: %s\n", user_name ? user_name : "(null)");
        printf ("Kennwort: %s\n", password ? password : "(null)");
        printf ("Portnummer: %u\n", port_num);
        printf ("Socketname: %s\n", socket_name ? socket_name : "(null)");

        printf ("Argumentvektor nach getopt_long():\n");
        for (i = 0; i < argc; i++)
            printf ("arg %d: %s\n", i, argv[i]);

        exit (0);
}
```

Um den Argumentvektor zu verarbeiten, verwendet `show_argv` die Funktion `getopt_long()`, die in der Regel in einer Schleife aufgerufen wird:

```
while ((c = getopt_long (argc, argv, "h:p::u:P:S:", long_options,
                &option_index)) != EOF)
{
    /* process option */
}
```

Die beiden ersten Argumente von `getopt_long()` sind der Argumentzähler und der Argumentvektor Ihres Programms. Das dritte Argument gibt die Optionen an, die Sie zu erkennen wünschen. Es handelt sich dabei um die Kurzformen für Ihre Programmoptionen. Auf die Optionsbuchstaben kann ein Doppelpunkt, ein doppelter Doppelpunkt oder kein Doppelpunkt folgen, um anzuzeigen, dass der Option ein Optionswert folgen muss, folgen kann oder nicht folgt. Das vierte Argument, `long_options`, ist ein Zeiger auf ein Array mit Optionsstrukturen, die jeweils Informationen für eine Option angeben, welche Ihr Programm verstehen soll. Seine Aufgabe ist ähnlich der Optionszeichenkette im dritten Argument. Die vier Elemente jeder `long_options[]`-Struktur sehen wie folgt aus:

- **Der lange Name der Option**
- **Ein Wert für die Option**. Dieser Wert kann `required_argument`, `optional_argument` oder `no_argument` sein, was angibt, ob der Option ein Optionsfeld folgen muss, folgen kann oder nicht folgt. (Die genannten Werte haben dieselbe Wirkung wie der Doppelpunkt, der doppelte Doppelpunkt oder kein Doppelpunkt im dritten Argument der Optionszeichenkette.)

- **Ein Flag-Argument**. Hier speichern Sie einen Zeiger auf eine Variable. Wird die Option gefunden, speichert `getopt_long()` den im vierten Argument angegebenen Wert in der Variablen. Ist das Flag `NULL`, setzt `getopt_long()` statt dessen die Variable `optarg` so, dass sie auf einen Wert zeigt, der der Option folgt, und gibt den Kurznamen der Option zurück. Unser `long_options[]`-Array gibt für alle Optionen `NULL` an. Deshalb gibt `getopt_long()` jedes Argument so zurück, wie es erkannt wurde, so dass wir es in der `switch`-Anweisung verarbeiten können.
- **Den Kurznamen (ein Zeichen) der Option**. Die im `long_options[]`-Array angegebenen Kurznamen *müssen* mit den Buchstaben der `getopt_long()` als drittes Argument übergebenen Optionszeichenkette übereinstimmen, sonst kann Ihr Programm die Kommandozeilenargumente nicht korrekt verarbeiten.

Das Array `long_options[]` muss mit einer Struktur abgeschlossen werden, deren Elemente alle 0 sind.

Das fünfte Argument von `getopt_long()` ist ein Zeiger auf eine `int`-Variable. `getopt_long()` speichert in dieser Variablen den Index der `long_options[]`-Struktur, die der zuletzt erkannten Option entspricht. (`show_param` macht mit diesem Wert nichts.)

Beachten Sie, dass die Kennwortoption (angegeben als `--password` oder `-p`) einen optionalen Wert annehmen kann. Das bedeutet, Sie können sie als `--password` oder `--password=`*ihr_kennwort* angeben, wenn Sie die lange Form verwenden, oder als `-p` oder `-p`*ihr_kennwort*, wenn Sie die kurze Form verwenden. Der doppelte Doppelpunkt hinter dem »p« in der Optionszeichenkette zeigt an, dass der Kennwortwert optional ist, ebenso wie `optional_argument` im `long_options[]`-Array. MySQL-Clients erlauben normalerweise, den Kennwortwert in der Kommandozeile wegzulassen, danach werden Sie zur Eingabe aufgefordert. Auf diese Weise können Sie vermeiden, das Kennwort in der Kommandozeile eingeben zu müssen, so dass Sie niemand bei der Eingabe beobachten kann. Bei der Entwicklung unseres nächsten Clients, `client4`, fügen wir diese Kennwortüberprüfung ein.

Hier ein Beispielaufruf von `show_param` mit der resultierenden Ausgabe (vorausgesetzt, `~.my.cnf` hat noch denselben Inhalt wie im Beispiel für `show_argv`):

```
% show_param -h yet_another_host x
Ursprüngliche Verbindungsparameter:
host name: (null)
user name: (null)
password: (null)
port number: 0
socket name: (null)
Ursprünglicher Argumentvektor:
arg 0: show_param
arg 1: -h
```

```
arg 2: yet_another_host
arg 3: x
Veränderter Argumentvektor nach load_defaults():
arg 0: show_param
arg 1: --user=paul
arg 2: --password=secret
arg 3: --host=some_host
arg 4: -h
arg 5: yet_another_host
arg 6: x
Verbindungsparameter nach getopt_long():
host name: yet_another_host
user name: paul
password: secret
port number: 0
socket name: (null)
Argumentvektor nach getopt_long():
arg 0: x
```

Die Ausgabe zeigt, dass der Hostname aus der Kommandozeile übernommen wurde (und damit den Wert aus der Optionsdatei überschreibt) und dass der Benutzername und das Kennwort aus der Optionsdatei stammen. `getopt_long()` wertet die Optionen korrekt aus, egal ob in Kurz- oder Langform angegeben.

Jetzt wollen wir alles weglassen, was nur zeigen sollte, wie die Routinen zur Optionsverarbeitung funktionieren, und den Rest als Grundlage für einen Client nutzen, der eine Verbindung zu einem Server einrichtet und dazu die Optionen aus der Optionsdatei oder von der Kommandozeile nutzt. Hier die Quelldatei, `client4.c`:

```
/* client4.c */

#include <stdio.h>
#include <stdlib.h> /* for atoi() */
#include <mysql.h>
#include "common.h"
#include "getopt.h"

#define def_host_name   NULL /* Host (Standard = localhost) */
#define def_user_name   NULL /* Benutzername (Standard = Login-Name) */
#define def_password    NULL /* Kennwort (Standard = keines) */
#define def_port_num    0    /* Standardport verwenden*/
#define def_socket_name NULL /* Standard-Socketname verwenden */
#define def_db_name     NULL /* Datenbank (Standard = keine) */

char *groups[] = { "client", NULL };

struct option long_options[] =
```

```
{
    {"host",     required_argument, NULL, 'h'},
    {"user",     required_argument, NULL, 'u'},
    {"password", optional_argument, NULL, 'p'},
    {"port",     required_argument, NULL, 'P'},
    {"socket",   required_argument, NULL, 'S'},
    { 0, 0, 0, 0 }
};

MYSQL   *conn;  /* Zeiger auf Verbindungs-Handle */

int
main (int argc, char *argv[])
{
char *host_name = def_host_name;
char *user_name = def_user_name;
char *password = def_password;
unsigned int port_num = def_port_num;
char *socket_name = def_socket_name;
char *db_name = def_db_name;
char    passbuf[100];
int ask_password = 0;
int c, option_index=0;
int i;

    my_init ();
    load_defaults ("my", groups, &argc, &argv);

    while ((c = getopt_long (argc, argv, "h:p::u:P:S:", long_options,
                    &option_index)) != EOF)
    {
        switch (c)
        {
        case 'h':
            host_name = optarg;
            break;
        case 'u':
            user_name = optarg;
            break;
        case 'p':
            if (!optarg)      /* kein Wert angegeben */
                ask_password = 1;
            else              /* Kennwort kopieren, Original löschen */
            {
                    (void) strncpy (passbuf, optarg, sizeof(passbuf)-1);
                    passbuf[sizeof(passbuf)-1] = '\0';
                    password = passbuf;
                    while (*optarg)
```

```
                        *optarg++ = ' ';
                }
                break;
            case 'P':
                port_num = (unsigned int) atoi (optarg);
                break;
            case 'S':
                socket_name = optarg;
                break;
            }
        }

        argc -= optind; /* nach den Argumenten, die von */
        argv += optind; /* getopt_long() verarbeitet wurden*/

        if (argc > 0)
        {
            db_name = argv[0];
            --argc; ++argv;
        }

        if (ask_password)
            password = get_tty_password (NULL);

        conn = do_connect (host_name, user_name, password, db_name,
                                        port_num, socket_name, 0);
        if (conn == NULL)
            exit (1);

        /* hier passiert die eigentliche Arbeit */

        do_disconnect (conn);
        exit (0);
}
```

Im Vergleich zu den zuvor entwickelten Programmen `client1`, `client2` und `client3` erledigt `client4` einige Dinge, die wir zuvor noch nicht gesehen haben:

- Es erlaubt, den Datenbanknamen in der Kommandozeile anzugeben und zwar hinter den Optionen, die von `getopt_long()` ausgewertet werden. Das ist konsistent mit dem Verhalten der Standard-Clients in der MySQL-Distribution.
- Es löscht alle Kennwortwerte im Argumentvektor, nachdem es eine Kopie davon angelegt hat. Damit ist das Kennwort weniger lang für `ps` oder andere Systemstatusprogramme sichtbar. (Die Zeitdauer ist nur weniger lang. Die Angabe von Kennwörtern in der Kommandozeile bleibt ein Sicherheitsrisiko.)

- Wurde eine Kennwortoption ohne Wert angegeben, fordert der Client den Benutzer mit `get_tty_password()` auf, es einzugeben. Das ist eine Dienstprogrammroutine in der Client-Bibliothek, die ein Kennwort anfordert, ohne die Eingabe auf dem Bildschirm anzuzeigen. (Die Client-Bibliothek enthält lauter derartige Dinge. Sie sollten den Quellcode der MySQL-Client-Programme unbedingt lesen, weil Sie dort erfahren, welche Routinen es gibt und wie Sie sie verwenden.) Sie fragen sich vielleicht, warum nicht einfach `getpass()` aufgerufen wird. Die Antwort ist, dass nicht alle Systeme diese Funktion unterstützen, beispielsweise Windows. `get_tty_password()` ist portierbar zwischen den Systemen, weil es Abweichungen zwischen den Systemen kompensiert.

`client4` verhält sich so, wie Sie es in den Optionen festgelegt haben. Angenommen, es gibt keine Optionsdatei, die das Ganze verkompliziert und Sie rufen `client4` ohne Argumente auf, dann stellt es eine Verbindung zu `localhost` her und übergibt dem Server Ihren UNIX-Login-Namen und kein Kennwort. Wenn Sie `client4` dagegen wie folgt aufrufen, fordert es Sie auf, ein Kennwort einzugeben (dem `-p` folgt kein Kennwortwert), stellt eine Verbindung zu `some_host` her und übergibt dem Server den Benutzernamen `some_user` sowie das eingegebene Kennwort:

```
% client4 -h some_host -u some_user -p some_db
```

`client4` übergibt außerdem `do_connect()` den Datenbanknamen, `some_db`, um diese Datenbank zur aktuellen Datenbank zu machen. Falls es eine Optionsdatei gibt, wird ihr Inhalt verarbeitet und die Verbindungsparameter werden entsprechend abgeändert..

Zuvor haben wir im Zuge der Code-Kapselung Hüllfunktionen für den Aufbau und Abbau von Verbindungen zum Server entwickelt. Jetzt sollten wir überlegen, ob es nicht auch sinnvoll wäre, den Code für die Optionsübergabe in eine Hüllfunktion einzuschließen. Das ist durchaus möglich, aber ich werde es hier nicht zeigen. Der Code für die Auswertung von Optionen ist – anders als der Code für den Verbindungsaufbau – nicht in jedem Programm gleich. Viele Programme unterstützen neben den Standardoptionen auch noch andere Optionen und unterschiedliche Programme verwenden sehr wahrscheinlich auch unterschiedliche Zusatzoptionen. Damit wird es schwierig, eine Funktion zu entwickeln, die die Schleife zur Optionsverarbeitung standardisiert.

6.6 Anfrageverarbeitung

Nachdem wir wissen, wie wir unsere Kommunikation mit dem Server beginnen, wollen wir betrachten, wie wir sie weiterführen. Dieser Abschnitt beschreibt die Kommunikation mit dem Server zur Anfrageverarbeitung.

Alle Anfragen laufen nach dem folgenden Muster ab:

1. **Konstruieren der Anfrage**. Wie Sie das machen, ist vom Inhalt der Anfrage abhängig – insbesondere davon, ob sie binäre Daten enthält.

2. **Absetzen der Anfrage, indem sie zur Ausführung an den Server geschickt wird**
3. **Verarbeiten des Anfrage-Ergebnisses.** Das ist davon abhängig, welchen Anfragetyp Sie ausgeführt haben. Beispielsweise gibt eine `SELECT`-Anweisung Datenzeilen zurück, die Sie verarbeiten können, eine `INSERT`-Anweisung dagegen nicht.

Beim Konstruieren von Anfragen sollten Sie berücksichtigen, mit welcher Funktion sie zum Server geschickt wird. Die allgemeinere Routine zum Absetzen von Anfragen ist `mysql_real_query()`. Mit dieser Routine übergeben Sie die Anfrage als gezählte Zeichenkette (eine Zeichenkette plus ihre Länge). Sie müssen die Länge Ihrer Anfragezeichenkette verwalten und sie `mysql_real_query()` übergeben, zusammen mit der eigentlichen Zeichenkette. Weil die Anfrage eine gezählte Zeichenkette ist, kann sie beliebige Dinge enthalten, auch binäre Daten oder Null-Bytes. Die Anfrage wird nicht als null-terminierte Zeichenkette behandelt.

Die andere Funktion zum Absetzen von Anfragen, `mysql_query()`, ist restriktiver in Hinblick auf den Inhalt der Anfragezeichenkette, aber häufig einfacher in der Anwendung. Anfragen, die Sie `mysql_query()` übergeben, sollten null-terminierte Zeichenketten sein, d.h. sie dürfen keine Null-Bytes im Anfragetext enthalten. (Falls die Anfragen Null-Bytes enthalten, werden sie irrtümlich kürzer interpretiert, als sie eigentlich sind.) Allgemein ausgedrückt, wenn Ihre Anfrage beliebige binäre Daten enthalten soll, möglicherweise auch Null-Bytes, dann sollten Sie nicht `mysql_query()` verwenden. Wenn Sie dagegen mit null-terminierten Zeichenketten arbeiten, genießen Sie den Luxus, Ihre Anfragen mit Hilfe der Zeichenkettenfunktionen aus der C-Bibliothek anlegen zu können, die Sie vielleicht schon kennen, beispielsweise `strcpy()` und `sprintf()`.

Ein weiterer Faktor, der bei der Anfragekonstruktion berücksichtigt werden sollte, sind die Escape-Operationen für Zeichen. Sie brauchen sie, wenn Sie vorhaben, Anfragen zu entwickeln, die binäre Daten oder andere problematische Zeichen enthalten, beispielsweise Anführungszeichen oder Backslashes. Eine detaillierte Beschreibung finden Sie im Abschnitt »Problematische Daten in Anfragen kodieren«.

Eine Anfrageverarbeitung könnte vereinfacht wie folgt aussehen:

```
if (mysql_query (conn, query) != 0)
{
    /* Fehler, Bericht ausgeben */
}
else
{
    /* Erfolg; feststellen, was die Anfrage bewirkt hat */
}
```

`mysql_query()` und `mysql_real_query()` geben für erfolgreiche Anfragen Null zurück, andernfalls einen Wert ungleich Null. Eine Anfrage war erfolgreich, wenn der

Server sie als erlaubt akzeptiert hat und sie ausführen konnte. Das sagt nichts über das Ergebnis der Anfrage aus. Beispielsweise ist damit nicht garantiert, dass eine `SELECT`-Anfrage Zeilen ermittelt hat oder dass eine `DELETE`-Anfrage Zeilen gelöscht hat. Das Ergebnis muss in einer weiteren Verarbeitung ausgewertet werden.

Eine Anfrage kann aus den unterschiedlichsten Gründen fehlschlagen. Einige häufige Ursachen sind:

- Sie enthält einen Syntaxfehler.
- Sie ist semantisch fehlerhaft – beispielsweise eine Anfrage, die auf eine nicht existierende Spalte einer Tabelle verweist.
- Sie haben nicht die erforderlichen Berechtigungen, um auf die betreffenden Daten zuzugreifen.

Anfragen können in zwei allgemeine Kategorien eingeteilt werden: solche, die kein Ergebnis erzeugen, und solche, die ein Ergebnis erzeugen. Anfragen für Anweisungen wie `INSERT`, `DELETE` und `UPDATE` erzeugen kein Ergebnis. Sie geben keine Zeilen zurück, auch nicht für Anfragen, die Ihre Datenbank ändern. Die einzige Information, die Sie zurückerhalten, ist die Anzahl der betroffenen Zeilen.

Anfragen für Anweisungen wie `SELECT` und `SHOW` erzeugen ein Ergebnis. Schließlich sollen Sie damit Informationen ermitteln. Die Zeilen, die eine Anfrage zurückgibt, werden als Ergebnismenge bezeichnet. In MySQL wird sie durch den Datentyp `MYSQL_RES` dargestellt, eine Struktur, die die Datenwerte der Zeilen enthält, ebenso wie Metadaten über die Werte (beispielsweise die Spaltennamen und Datenwertlängen). Eine leere Ergebnismenge (d.h. sie enthält null Zeilen) ist etwas anderes als »kein Ergebnis«.

6.6.1 Verarbeitung von Anfragen ohne Ergebnismenge

Um eine Anfrage zu verarbeiten, die keine Ergebnismenge zurückgibt, setzen Sie die Anfrage mit `mysql_query()` oder `mysql_real_query()` ab. Ist die Anfrage erfolgreich, finden Sie mit `mysql_affected_rows()` heraus, wie viele Zeilen eingefügt, gelöscht oder aktualisiert wurden.

Das folgende Beispiel zeigt, wie eine Anfrage ohne Ergebnismenge behandelt wird:

```
if (mysql_query (conn, "INSERT INTO my_tbl SET name = 'My Name'") != 0)
{
    print_error ("INSERT-Anweisung fehlgeschlagen");
}
else
{
    printf ("INSERT-Anweisung erfolgreich : %lu Zeilen bearbeitet\n",
                (unsigned long) mysql_affected_rows (conn));
}
```

Beachten Sie, dass das Ergebnis von `mysql_affected_rows()` für die Ausgabe in einen `unsigned long` umgewandelt wurde. Diese Funktion gibt einen Wert des Typs `my_ulonglong` zurück, aber auf einigen Systemen können solche Werte nicht direkt ausgegeben werden. (Beispielswiese funktioniert das unter FreeBSD, nicht aber unter Solaris.) Das Problem wird gelöst durch die Umwandlung des Werts in einen unsigned long und die Verwendung des Ausgabeformats `'%lu'`. Dasselbe gilt für alle anderen Funktionen, die `my_ulonglong`-Werte zurückgeben, beispielsweise `mysql_num_rows()` und `mysql_insert_id()`. Wenn Ihre Client-Programme über verschiedene Systeme portabel sein sollen, berücksichtigen Sie bitte diese Tatsache.

`mysql_rows_affected()` gibt die Anzahl der von der Anfrage bearbeiteten Zeilen zurück, aber die Bedeutung von »bearbeitete Zeilen« ist vom Anfragetyp abhängig. Für `INSERT`, `REPLACE` oder `DELETE` ist das die Anzahl der eingefügten, ersetzten oder gelöschten Zeilen. für `UPDATE` ist es die Anzahl der Zeilen, die aktualisiert wurden, d.h. die Anzahl der Zeilen, die MySQL wirklich geändert hat. MySQL aktualisiert eine Zeile nicht, wenn sie bereits den richtigen Wert enthält. Das bedeutet, eine Zeile muss nicht unbedingt geändert werden, wenn sie zur Aktualisierung ausgewählt (durch die `WHERE`-Klausel der `UPDATE`-Anweisung) wird.

Diese Bedeutung von »bearbeitete Zeilen« für `UPDATE` ist etwas irreführend, weil manche Anwender meinen, es bedeute »gefundene Zeilen« – d.h. die Anzahl der Zeilen, die zur Aktualisierung ausgewählt wurden, auch wenn diese letztlich ihre Werte nicht geändert hat. Wenn Ihre Anwendung diese Aussage benötigt, fordern Sie sie beim Verbindungsaufbau zum Server an. Übergeben Sie `mysql_real_connect()` den `flags`-Wert `CLIENT_FOUND_ROWS`. Sie können auch `do_connect()` `CLIENT_FOUND_ROWS` als `flags`-Argument übergeben; es gibt den Wert an `mysql_real_connect()` weiter.

6.6.2 Verarbeitung von Anfragen mit Ergebnismenge

Anfragen, die Daten zurückgeben, erzeugen eine Ergebnismenge, die Sie nach Ausführung der Anfrage durch Aufruf von `mysql_query()` oder `mysql_real_query()` verarbeiten müssen. Beachten Sie, dass in MySQL `SELECT` nicht die einzige Anweisung ist, die Zeilen zurückgibt. Auch `SHOW`, `DESCIRBE` und `EXPLAIN` erzeugen Ergebnisse. Für alle diese Anweisungen ist nach Ausführung der Anfrage eine zusätzliche Verarbeitung der Zeilen erforderlich.

Ergebnismengen werden wie folgt behandelt:

- **Erzeugen Sie die Ergebnismenge durch Aufruf von mysql_store_result() oder mysql_use_result().** Diese Funktionen geben einen `MYSQL_RES`-Zeiger bei Erfolg oder `NULL` bei Misserfolg zurück. Später werden wir die Unterschiede zwischen `mysql_store_result()` und `mysql_use_result()` betrachten, ebenso wie die Situationen, in denen sie eingesetzt werden. Unsere Beispiele hier verwenden `mysql_store_result()`, das die Zeilen unmittelbar vom Server zurückgibt und sie im Client speichert.

- **Rufen Sie für jede Zeile der Ergebnismenge mysql_fetch_row() auf.** Diese Funktion gibt einen `MYSQL_ROW`-Wert zurück, einen Zeiger auf ein Array mit Zeichenketten, die die Werte aller Spalten in der Zeile repräsentieren. Was Sie mit der Zeile letztlich machen, ist von der jeweiligen Anwendung abhängig. Sie könnten die Spaltenwerte einfach ausgeben, statistische Berechnungen durchführen oder ganz etwas anderes damit tun. `mysql_fetch_row()` gibt `NULL` zurück, wenn keine weiteren Zeilen in der Ergebnismenge stehen.
- **Nachdem Sie die Ergebnismenge verarbeitet haben, rufen Sie mysql_free_result() auf, um den davon belegten Speicher freizugeben**. Wenn Sie das versäumen, verliert Ihre Anwendung Speicher. (Insbesondere ist es für lang laufende Anwendungen wichtig, die Ergebnismengen korrekt freizugeben, andernfalls wird Ihr System langsamer, weil es von Prozessen belegt wird, die immer mehr Systemressourcen in Anspruch nehmen.)

Das folgende Beispiel demonstriert, wie eine Anfrage mit Ergebnismenge verarbeitet wird:

```
MYSQL_RES *res_set;

if (mysql_query (conn, "SHOW TABLES FROM mysql") != 0)
    print_error (conn, "mysql_query() failed");
else
{
    res_set = mysql_store_result (conn);    /* Ergebnismenge erzeugen */
    if (res_set == NULL)
            print_error (conn, "mysql_store_result() fehlgeschlagen");
    else
    {
        /* Ergebnismenge verarbeiten, dann freigeben */
        process_result_set (conn, res_set);
        mysql_free_result (res_set);
    }
}
```

Wir haben hier mit dem Aufruf der Funktion `process_result_set()` für jede Zeile ein bisschen geschummelt. Wir haben diese Funktion noch nicht definiert, was wir jetzt nachholen wollen. Im Allgemeinen basieren Funktionen zur Verarbeitung von Ergebnismengen auf einer Schleife, die wie folgt aussehen kann:

```
MYSQL_ROW row;

while ((row = mysql_fetch_row (res_set)) != NULL)
{
    /* irgendetwas mit dem Zeileninhalt machen */
}
```

Der MYSQL_ROW-Rückgabewert von mysql_fetch_row() ist ein Zeiger auf ein Array mit Werten. Der Zugriff auf die Werte erfolgt also einfach als row[i], wobei i im Bereich zwischen 0 und der Anzahl der Zeilenspalten minus 1 liegt.

Der Datentyp MYSQL_ROW hat einige wichtige Eigenschaften:

- MYSQL_ROW ist ein Zeigertyp; Variablen dieses Typs werden also als MYSQL_ROW row deklariert, nicht als MYSQL_ROW *row.
- Die Zeichenketten in MYSQL_ROW sind null-terminiert. Wenn eine Spalte jedoch binäre Daten enthalten darf, darf sie auch Null-Bytes enthalten, Sie sollten den Wert also nicht als null-terminierte Zeichenkette behandeln. Ermitteln Sie die Spaltenlänge, um die Länge des Spaltenwerts festzustellen.
- Alle Werte für alle Datentypen werden als Zeichenketten zurückgegeben, auch für numerische Typen. Wenn Sie einen Wert als Zahl verarbeiten wollen, müssen Sie ihn selbst umwandeln.
- NULL-Werte werden im MYSQL_ROW-Array als NULL-Zeiger dargestellt. Wenn Sie eine Spalte nicht als NOT NULL deklariert haben, sollten Sie immer prüfen, ob die Werte für diese Spalte NULL-Zeiger sind.

Ihre Anwendungen können mit dem Zeileninhalt tun, was immer Sie möchten. Der Demonstration halber wollen wir die Zeilen ausgeben und dabei die Spaltenwerte durch Tabulatorzeichen voneinander abtrennen. Dazu brauchen wir eine zusätzliche Funktion, mysql_num_fields(), aus der Client-Bibliothek; diese Funktion teilt uns mit, wie viele Werte (Spalten) die Zeile enthält.

Hier sehen Sie den Code für process_result_set():

```
void
process_result_set (MYSQL *conn, MYSQL_RES *res_set)
{
MYSQL_ROW       row;
unsigned int    i;

    while ((row = mysql_fetch_row (res_set)) != NULL)
    {
        for (i = 0; i < mysql_num_fields (res_set); i++)
        {
            if (i > 0)
                fputc ('\t', stdout);
            printf ("%s", row[i] != NULL ? row[i] : "NULL");
        }
        fputc ('\n', stdout);
    }
    if (mysql_errno (conn) != 0)
        print_error (conn, "mysql_fetch_row() fehlgeschlagen");
    else
```

```
        printf ("%lu Zeilen ermittelt\n", (unsigned long) mysql_num_rows
(res_set));
}
```

`process_result_set()` gibt die Zeilen aus und trennt dabei die einzelnen Spalten durch Tabulatorzeichen voneinander ab (`NULL`-Werte werden mit dem Wort »NULL« angezeigt), gefolgt von einem Zähler der ermittelten Zeilen. Dieser Zähler wird durch Aufruf von `mysql_num_rows()` zur Verfügung gestellt. Wie `mysql_affected_rows()` gibt `mysql_num_rows()` einen `my_ulonglong`-Wert zurück, den Sie in einen `unsigned long` umwandeln und unter Verwendung des Formats `%lu` ausgeben sollten.

Der Schleife zum Ermitteln der Zeilen folgt eine Fehlerprüfung. Wenn Sie mit `mysql_store_result()` eine Ergebnismenge erzeugen, bedeutet der Rückgabewert `NULL` von `mysql_fetch_row()` immer »keine weiteren Zeilen«. Erzeugen Sie die Ergebnismenge dagegen mit `mysql_use_reuslt()`, kann der Rückgabewert `NULL` von `mysql_fetch_row()` bedeuten, dass keine weiteren Zeilen vorliegen oder dass ein Fehler aufgetreten ist. Die Fehlerprüfung ermöglicht `process_result_set()`, Fehler zu erkennen, egal wie Sie Ihre Ergebnismenge erzeugen.

Diese Version von `process_result_set()` verfolgt einen eher minimalistischen Ansatz, der gewisse Unzulänglichkeiten aufweist um Spaltenwerte auszugeben. Angenommen, Sie führen die folgende Anfrage aus:

```
SELECT last_name, first_name, city, state FROM president
```

Damit erhalten Sie die folgende Ausgabe:

```
Adams   John    Braintree   MA
Adams   John Quincy Braintree   MA
Arthur  Chester A.  Fairfield   VT
Buchanan    James   Mercersburg PA
Bush    George W.   Milton  MA
Carter  James E. Jr Plains  GA
Cleveland   Grover  Caldwell    NJ
...
```

Wir könnten die Ausgabe verschönern, indem wir Spaltenüberschriften ausgeben und die Werte vertikal ausrichten. Dazu brauchen wir Beschriftungen und wir müssen wissen, wie groß der breiteste Wert jeder Spalte ist. Diese Information steht zur Verfügung, aber nicht als Teil der Spaltendatenwerte – sie ist in den Metadaten (Daten über die Daten) der Ergebnismenge festgehalten. Nachdem wir unsere Anfrageverarbeitung noch ein wenig verallgemeinert haben, werden wir sie auch ansprechender formatieren: im Abschnitt »Metadaten in Ergebnismengen«.

6.6.3 Eine allgemeine Anfrageverarbeitung

Bei den oben beschriebenen Beispielen zur Anfrageverarbeitung wusste man im Voraus, ob die Anweisungen Ergebnismengen erzeugen. Die Anfragen waren im Code festgeschrieben, deshalb war das möglich: Es ging um eine `INSERT`-Anweisung, die keine Ergebnismenge zurückgibt, und um eine `SHOW TABLES`-Anweisung, die eine Ergebnismenge zurückgibt.

Sie wissen aber nicht immer im Voraus, welche Anweisung in der Anfrage stehen wird. Wenn Sie beispielsweise eine Anfrage ausführen, die Sie von der Tastatur oder aus einer Datei einlesen, kann diese eine beliebige Anweisung enthalten. Sie wissen nicht vorab, ob dafür Zeilen zurückgegeben werden. Was also ist zu tun? Sicher wollen Sie nicht versuchen, die Anfrage auf ihre Anweisung hin auszuwerten. Das ist nämlich auch nicht so einfach, wie es scheint. Es reicht nicht, nur das erste Wort zu betrachten, weil die Anfrage auch einen Kommentar wie den folgenden beinhalten könnte:

```
/* comment */ SELECT ...
```

Glücklicherweise müssen Sie den Typ der Anfrage nicht im Voraus wissen, um sie korrekt verarbeiten zu können. Das MySQL-C-API ermöglicht, eine allgemeine Anfrageverarbeitung zu entwickeln, die jede Anweisung korrekt verarbeitet, egal ob diese eine Ergebnismenge erzeugt oder nicht.

Bevor wir den Code für die Anfrageverarbeitung schreiben, wollen wir ihre Arbeitsweise beschreiben:

- Absetzen der Anfrage. Falls sie fehlschlägt, sind wir fertig.
- Ist die Anfrage erfolgreich, rufen wir `mysql_store_result()` auf, um die Zeilen vom Server zu ermitteln und eine Ergebnismenge zu erzeugen.
- Schlägt `mysql_store_result()` fehl, könnte es sein, dass die Anfrage keine Ergebnismenge erzeugt hat oder dass beim Versuch, die Ergebnismenge zu ermitteln, ein Fehler aufgetreten ist. Sie unterscheiden diese beiden Fälle, indem Sie `mysql_field_count()` aufrufen und seinen Wert überprüfen:
  - Ist `mysql_field_count()` ungleich Null, ist ein Fehler aufgetreten. Die Anfrage sollte eine Ergebnismenge zurückgeben, hat das aber nicht getan. Das kann unterschiedliche Ursachen haben. Beispielsweise könnte es sein, dass die Ergebnismenge zu groß war und nicht genügend Speicher dafür zur Verfügung stand oder dass während des Ladens der Zeilen das Netzwerk zwischen dem Client und dem Server ausgefallen ist.

  Erschwert wird diese Prozedur dadurch, dass es `mysql_field_count()` erst seit MySQL 3.22.24 gibt. In früheren Versionen verwenden Sie statt dessen `mysql_num_fields()`. Um Programme zu entwickeln, die mit allen MySQL-Versionen funktionieren, nehmen Sie den folgenden Codeabschnitt in alle Dateien auf, die `mysql_field_count()` aufrufen:

```
#if !defined(MYSQL_VERSION_ID) || MYSQL_VERSION_ID<32224
#define mysql_field_count mysql_num_fields
#endif
```

Damit werden alle Aufrufe von `mysql_field_count()` in MySQL-Versionen vor 3.22.24 als Aufruf von `mysql_num_fields()` behandelt.

- Gibt `mysql_field_count()` `0` zurück, hat die Anfrage keine Ergebnismenge erzeugt. (Das bedeutet, die Anfrage enthielt eine Anweisung wie `INSERT`, `DELETE` oder `UPDATE`.)
- War `mysql_store_result()` erfolgreich, gibt die Anfrage eine Ergebnismenge zurück. Verarbeiten Sie die Zeilen durch Aufruf von `mysql_fetch_row()`, bis es `NULL` zurückgibt.

Das folgende Listing zeigt eine Funktion, die von beliebigen Anfragen verarbeitet wird, wobei ein Verbindungs-Handle sowie eine null-terminierte Anfragezeichenkette vorgegeben werden:

```
#if !defined(MYSQL_VERSION_ID) || MYSQL_VERSION_ID<32224
#define mysql_field_count mysql_num_fields
#endif

void
process_query (MYSQL *conn, char *query)
{
MYSQL_RES *res_set;
unsigned int field_count;

    if (mysql_query (conn, query) != 0) /* Anfrage fehlgeschlagen */
    {
        print_error (conn, "process_query() fehlgeschlagen");
        return;
    }

    /* Anfrage erfolgreich; feststellen, ob sie Daten zurückgibt */

    res_set = mysql_store_result (conn);
    if (res_set == NULL)    /* keine Ergebnismenge zurückgegeben */
    {
        /*
         * bedeutet das Fehlen einer Ergebnismenge, dass ein Fehler
         * aufgetreten ist, oder dass es keine Ergebnismenge gibt?
         */
        if (mysql_field_count (conn) > 0)
        {
            /*
             * obwohl eine Ergebnismenge zu erwarten war, hat
             * mysql_store_result() keine zurückgegeben,
```

```
                 * d.h., ein Fehler ist aufgetreten
                 */
                print_error (conn, "Problem bei Verarbeitung der Ergebnis-
                menge");
            }
            else
            {
                /*
                 * keine Ergebnismenge zurückgegeben; Anfrage gab keine
                 * Daten zurück
                 * (es war kein SELECT, SHOW, DESCRIBE oder EXPLAIN),
                 * Anzahl der betroffenen Zeilen zurückmelden
                 */
                printf ("%lu Zeilen betroffen\n",
                            (unsigned long) mysql_affected_rows (conn));
            }
        }
        else    /* es wurde eine Ergebnismenge zurückgegeben */
        {
            /* Zeilen verarbeiten und Ergebnismenge freigeben */
            process_result_set (conn, res_set);
            mysql_free_result (res_set);
        }
    }
```

6.6.4 Alternative Ansätze zur Anfrageverarbeitung

Die oben gezeigte Version von `process_query()` hat die folgenden drei Eigenschaften:

- Sie verwendet `mysql_query()`, um die Anfrage abzusetzen.
- Sie verwendet `mysql_store_query()`, um die Ergebnismenge zu ermitteln.
- Wurde keine Ergebnismenge erhalten, ermittelt sie mit `mysql_field_count()`, ob ein Fehler aufgetreten ist oder ob keine Ergebnismenge zu erwarten ist.

Für diese drei Aspekte der Anfrageverarbeitung gibt es alternative Ansätze:

- Sie könnten statt einer null-terminierten Anfragezeichenkette und `mysql_query()` auch eine gezählte Anfragezeichenkette und `mysql_real_query()` verwenden.
- Sie könnten die Ergebnismenge durch Aufruf von `mysql_use_result()` statt von `mysql_store_result()` erzeugen.
- Sie könnten statt `mysql_field_count()` auch `mysql_error()` aufrufen, um festzustellen, ob die Ergebnismenge nicht ermittelt werden konnte oder ob es einfach keine Ergebnismenge gibt.

Hier ist die Funktion `process_real_query()`, die dasselbe erledigt wie `process_query()`, dazu aber die drei gezeigten Alternativen verwendet:

```
void
process_real_query (MYSQL *conn, char *query, unsigned int len)
{
MYSQL_RES *res_set;
unsigned int field_count;

    if (mysql_real_query (conn, query, len) != 0)   /* Anfrage fehlge-
                                                       schlagen */
    {
        print_error (conn, "process_real_query () fehlgeschlagen");
        return;
    }

    /* Anfrage erfolgreich; feststellen, ob sie Daten zurückgibt */

    res_set = mysql_use_result (conn);
    if (res_set == NULL)    /* keine Ergebnismenge zurückgegeben */
    {
        /*
         * bedeutet das Fehlen einer Ergebnismenge, dass ein Fehler
         * aufgetreten ist, oder dass es keine Ergebnismenge gibt?
         */
        if (mysql_errno (conn) != 0) /* ein Fehler ist aufgetreten */
            print_error (conn, "Problem bei der Verarbeitung der Ergeb-
                                nismenge");
        else
        {
            /*
             * es wurde keine Ergebnismenge zurückgegeben;
             * (es war kein SELECT, SHOW, DESCRIBE oder EXPLAIN),
             * also wird nur die Anzahl der betroffenen Zeilen gemeldet
             */
            printf ("%lu Zeilen gefunden\n",
                        (unsigned long) mysql_affected_rows (conn));
        }
    }
    else    /* es wurde eine Ergebnismenge zurückgegeben */
    {
        /* Zeilen verarbeiten und Ergebnismenge freigeben */
        process_result_set (conn, res_set);
        mysql_free_result (res_set);
    }
}
```

6.6.5 Ein Vergleich von mysql_store_result() und mysql_use_result()

Die Funktionen `mysql_store_result()` und `mysql_use_result()` sind ganz ähnlich. Beide nehmen einen Verbindungs-Handle als Argument entgegen und geben eine Ergebnismenge zurück. Es gibt jedoch entscheidende Unterschiede. Der wichtigste Unterschied ist darin zu finden, wie die Zeilen der Ergebnismenge vom Server ermittelt werden. `mysql_store_result()` ermittelt alle Zeilen unmittelbar nach dem Aufruf. `mysql_use_result()` initiiert die Suche, ermittelt aber noch keine der Zeilen. Stattdessen setzt es voraus, dass Sie die Datensätze später mit `mysql_fetch_row()` laden. Diese unterschiedlichen Ansätze führen zu allen möglichen Unterschieden zwischen den Funktionen. Dieser Abschnitt vergleicht die beiden Funktionen, so dass Sie besser entscheiden können, welche für Ihre Aufgabenstellung die richtige ist.

Wenn `mysql_store_result()` eine Ergebnismenge vom Server ermittelt, lädt es die Zeilen, reserviert Speicher für sie und legt sie auf dem Client ab. Nachfolgende Aufrufe von `mysql_fetch_row()` ergeben nie einen Fehler, weil sie einfach eine Zeile aus der Datenstruktur holen, die die Ergebnismenge bereits enthält. Die Rückgabe von `NULL` bedeutet immer, dass Sie das Ende der Ergebnismenge erreicht haben.

`mysql_use_result()` dagegen ermittelt keine Zeilen. Stattdessen initiiert es eine zeilenweise Suche, die Sie selbst vervollständigen müssen, indem Sie für jede Zeile `mysql_fetch_row()` aufrufen. In diesem Fall bedeutet die Rückgabe von `NULL` von `mysql_fetch_row()` normalerweise immer noch, dass das Ende der Ergebnismenge erreicht ist, es könnte aber auch bedeuten, dass während der Kommunikation mit dem Server ein Fehler aufgetreten ist. Sie unterscheiden diese beiden Ergebnisse durch Aufruf von `mysql_errno()` oder `mysql_error()`.

`mysql_store_result()` stellt höhere Anforderungen an Speicher und Rechenleistung als `mysql_use_result()`, weil die gesamte Ergebnismenge im Client abgelegt wird. Der zusätzliche Aufwand für die Speicherreservierung und die Einrichtung der Datenstruktur ist höher und ein Client, der große Ergebnismengen lädt, läuft Gefahr, diese nicht speichern zu können. Wenn Sie sehr viele Zeilen gleichzeitig anfordern, sollten Sie statt dessen `mysql_use_result()` verwenden.

`mysql_use_result()` stellt geringere Speicheranforderungen, weil immer nur Platz für eine einzelne Zeile gleichzeitig reserviert werden muss. Das kann schneller sein, weil Sie für die Ergebnismenge keine so komplizierte Datenstruktur einrichten müssen. Andererseits verursacht `mysql_use_result()` eine größere Last auf dem Server, der die Zeilen der Ergebnismenge vorhalten muss, bis der Client in der Lage ist, alle zu verarbeiten. Damit eignet sich `mysql_use_result()` für bestimmte Clients nicht:

- Interaktive Clients, die auf Anforderung des Benutzers von Zeile zu Zeile gehen. (Sie wollen schließlich nicht, dass der Server warten muss, bis er die nächste Zeile senden kann, nur weil der Benutzer gerade eine Kaffeepause macht.)

- Clients, die zwischen dem Laden der einzelnen Zeilen sehr viele Verarbeitungsvorgänge ausführen müssen.

In beiden Fällen kann der Client nicht alle Zeilen schnell in der Ergebnismenge ermitteln. Damit bindet er den Server an sich, was negative Auswirkungen auf andere Clients haben kann, weil Tabellen, aus denen Sie Daten laden, für die Dauer der Anfrage schreibgeschützt sind. Clients, die versuchen, diese Tabellen zu aktualisieren oder Zeilen einzufügen, werden blockiert.

Abgesehen von den zusätzlichen Speicheranforderungen, die `mysql_store_result()` benötigt, gibt es diverse Vorteile, wenn man sofort Zugriff auf die gesamte Ergebnismenge hat. Alle Zeilen der Ergebnismenge stehen zur Verfügung, so dass Sie wahlfrei darauf zugreifen können: Die Funktionen `mysql_data_seek()`, `mysql_row_seek()` und `mysql_row_tell()` erlauben Ihnen, in beliebiger Reihenfolge auf die Zeilen zuzugreifen. Bei Verwendung von `mysql_use_result()` können Sie nur in der Reihenfolge auf die Zeilen zugreifen, in der diese von `mysql_fetch_row()` geladen werden. Wenn Sie Zeilen nicht nur sequentiell verarbeiten wollen, wie sie vom Server zurückgegeben werden, sollten Sie statt dessen `mysql_store_result()` verwenden. Wenn Sie beispielsweise eine Anwendung haben, die dem Benutzer erlaubt, sich innerhalb der durch eine Anfrage ausgewählten Zeilen vorwärts und rückwärts zu bewegen, stellt `mysql_store_result()` die beste Wahl dar.

Mit `mysql_store_result()` können Sie bestimmte Spalteninformationen ermitteln, die bei Verwendung von `mysql_use_result()` nicht zur Verfügung stehen. Die Anzahl der Zeilen in der Ergebnismenge wird durch Aufruf von `mysql_num_rows()` ermittelt. Die maximalen Breiten der Werte in jeder Spalte werden im Element `max_width` von `MYSQL_FIELD` abgelegt, der Struktur für Spalteninformationen. Bei Verwendung von `mysql_use_result()` gibt `mysql_num_rows()` nicht den richtigen Wert zurück, bis Sie alle Zeilen geladen haben, und `max_withd` steht nicht zur Verfügung, weil es nur berechnet werden kann, nachdem die Daten aller Zeilen zurückgegeben wurden.

Weil `mysql_use_result()` weniger Arbeit leistet als `mysql_store_result()`, stellt es eine zusätzliche Anforderung, die es bei `mysql_store_result()` nicht gibt: Der Client muss `mysql_fetch_row()` für jede Zeile der Ergebnismenge aufrufen. Andernfalls werden die restlichen Datensätze der Ergebnismenge Teil der Ergebnismenge der nächsten Anfrage und es entsteht ein Synchronisierungsfehler. Das passiert bei `mysql_store_result()` nicht, denn wenn diese Funktion abgearbeitet ist, sind alle Zeilen bereits geladen. Bei `mysql_store_result()` müssen Sie `mysql_fetch_row()` überhaupt nicht selbst aufrufen. Das kann praktisch für Anfragen sein, bei denen Sie nur wissen wollen, ob die Ergebnismenge Einträge enthält, und nicht an den genauen Ergebnissen interessiert sind. Um beispielsweise festzustellen, ob es die Tabelle `my_tbl` gibt, könnten Sie die folgende Anfrage ausführen:

```
SHOW TABLES LIKE "my_tbl"
```

Falls der Wert von `mysql_num_rows()` nach Aufruf von `mysql_store_result()` ungleich Null ist, gibt es die Tabelle. `mysql_fetch_row()` muss nicht aufgerufen werden. (Sie müssen aber natürlich trotzdem `mysql_free_result()` aufrufen.)

Wenn Sie eine maximale Flexibilität bieten wollen, ermöglichen Sie den Benutzern, die Verarbeitungsmethode für die Ergebnismenge selbst auszusuchen. Zwei geeignete Programme dafür sind `mysql` und `mysqldump`. Sie verwenden standardmäßig `mysql_store_result()`, wechseln aber bei Angabe der Option `--quick` zu `mysql_use_result()`.

6.6.6 Metadaten in Ergebnismengen

Ergebnismengen enthalten nicht nur die Spaltenwerte für Datenzeilen, sondern auch Informationen über die Daten. Diese Informationen werden auch als Metadaten der Ergebnismenge bezeichnet. Sie enthalten die folgenden Informationen:

- Die Anzahl der Zeilen und Spalten in der Ergebnismenge, die durch Aufruf von `mysql_num_rows()` und `mysql_num_fields()` ermittelt werden.
- Die Länge der einzelnen Spaltenwerte in einer Zeile, die durch Aufruf von `mysql_fetch_legths()` ermittelt werden.
- Informationen über jede Spalte, beispielsweise ihren Namen und ihren Typ, die maximale Breite aller Spaltenwerte sowie die Tabelle, aus der sie stammt. Diese Information wird in `MYSQL_FIELD`-Strukturen gespeichert, die normalerweise mit `mysql_fetch_field()` ermittelt werden. Anhang F beschreibt die Struktur `MYSQL_FIELD` und listet alle Funktionen auf, die Zugriff auf die Spalteninformation bieten.

Die Verfügbarkeit von Metadaten ist zum Teil davon abhängig, mit welcher Methode Sie Ihre Ergebnismenge verarbeiten. Wie im vorigen Abschnitt bereits erklärt, müssen Sie die Ergebnismenge mit `mysql_store_result()`, nicht mit `mysql_use_result()` erzeugen, wenn Sie den Zeilenzähler oder maximale Spaltenlängen ermitteln wollen.

Metadaten zur Ergebnismenge helfen, Entscheidungen in Hinblick auf die Verarbeitung der Daten aus der Ergebnismenge zu treffen:

- Der Spaltenname und die Information über die Spaltenbreite erlauben eine wohl formatierte Ausgabe mit Spaltentiteln und vertikaler Ausrichtung.
- Mit Hilfe des Spaltenzählers ermitteln Sie, wie oft Sie eine Schleife zur Verarbeitung aufeinander folgender Spaltenwerte aus Datenzeilen durchlaufen müssen. Außerdem können Sie anhand der Zeilen- oder Spaltenzähler Datenstrukturen reservieren, die davon abhängig sind, dass Sie die Anzahl der Zeilen oder Spalten in der Ergebnismenge kennen.
- Sie können den Datentyp einer Spalte ermitteln. Auf diese Weise können Sie erkennen, ob eine Spalte eine Zahl repräsentiert, binäre Daten usw.

Im Abschnitt »Verarbeitung von Anfragen mit Ergebnismenge« haben wir eine Version von `process_result_set()` vorgestellt, die die Spalten aus den Zeilen der Ergebnismenge durch Tabulatorzeichen voneinander getrennt ausgab. Das kann durchaus sinnvoll sein (wenn Sie die Daten beispielsweise in eine Tabellenkalkulation importieren wollen), aber für Ausdrucke ist das Format nicht besonders ansprechend. Die Ausgabe dieser Version von `process_result_set()` sah so aus:

```
Adams   John    Braintree   MA
Adams   John Quincy Braintree   MA
Arthur  Chester A.  Fairfield   VT
Buchanan    James   Mercersburg PA
Bush    George W.   Milton  MA
Carter  James E. Jr Plains  GA
Cleveland   Grover  Caldwell    NJ
...
```

Jetzt wollen wir einige Änderungen an `process_result_set()` vornehmen, um eine tabellarische Ausgabe zu erhalten. Dazu führen wir eine Spaltenüberschrift ein und rahmen die einzelnen Spalten ein. Die überarbeitete Version zeigt dieselben Ergebnisse in einem gefälligeren Format an:

```
+------------+---------------+-------------+-------+
| last_name  | first_name    | city        | state |
+------------+---------------+-------------+-------+
| Adams      | John          | Braintree   | MA    |
| Adams      | John Quincy   | Braintree   | MA    |
| Arthur     | Chester A.    | Fairfield   | VT    |
| Buchanan   | James         | Mercersburg | PA    |
| Bush       | George W.     | Milton      | MA    |
| Carter     | James E., Jr. | Plains      | GA    |
| Cleveland  | Grover        | Caldwell    | NJ    |
| ...        | ...           | ...         | ...   |
+------------+---------------+-------------+-------+
```

Der Anzeigealgorithmus geht wie folgt vor:

1. Er ermittelt die Anzeigebreite aller Spalten.
2. Er gibt eine Zeile mit eingerahmten Spaltenüberschriften aus (durch vertikale Striche voneinander abgetrennt und zwischen Zeilen mit Trennstrichen eingerahmt).
3. Er gibt die Werte jeder Zeile der Ergebnismenge in den eingerahmten Spalten aus (abgetrennt durch vertikale Striche), vertikal ausgerichtet. Darüber hinaus werden Zahlen rechtsbündig ausgerichtet, und für `NULL`-Werte wird das Wort »`NULL`« ausgegeben.
4. Am Ende wird ein Zähler der Gesamtzeilenzahl ausgegeben.

Diese Übung demonstriert die Verwendung der Metadaten aus der Ergebnismenge. Um die oben beschriebene Ausgabe zu erzeugen, müssen wir mehr Dinge aus der Ergebnismenge anfragen, als nur die in den Zeilen enthaltenen Datenwerte.

Sie fühlen sich bei dieser Beschreibung vielleicht an die Vorgehensweise erinnert, wie `mysql` seine Ausgaben anzeigt. Das stimmt und Sie sollten den Code von `mysql` mit dem Code unserer überarbeiteten Version von `process_result_set()` vergleichen. Sie sind nicht gleich und vielleicht können Sie Nutzen aus diesem Vergleich zweier verschiedener Ansätze ziehen.

Als Erstes müssen wir die Anzeigebreite der einzelnen Spalten ermitteln, wie im folgenden Listing gezeigt. Beachten Sie, dass die Berechnungen ausschließlich anhand der Metadaten aus der Ergebnismenge erfolgen und dass die einzelnen Spaltenwerte dazu nicht herangezogen werden:

```
MYSQL_FIELD     *field;
unsigned int    i, col_len;

/* Spaltenanzeigebreite ermitteln */
mysql_field_seek (res_set, 0);
for (i = 0; i < mysql_num_fields (res_set); i++)
{
    field = mysql_fetch_field (res_set);
    col_len = strlen (field->name);
    if (col_len < field->max_length)
        col_len = field->max_length;
    if (col_len < 4 && !IS_NOT_NULL (field->flags))
        col_len = 4;    /* 4 = Länge des Worts "NULL" */
    field->max_length = col_len;    /* Spalteninfo zurücksetzen */
}
```

Die Spaltenbreiten werden berechnet, indem die `MYSQL_FIELD`-Strukturen für die Spalten der Ergebnismenge durchlaufen werden. `mysql_fetch_seek()` positioniert einen Zeiger auf die erste Struktur. Alle nachfolgenden Aufrufe von `mysql_fetch_field()` geben Zeiger auf die Strukturen für die jeweils nächsten Spalten zurück. Die Breite einer Spalte für Anzeigezwecke ist der Maximumwert aus drei Werten, die sich jeweils aus den Metadaten der Struktur mit den Spalteninformationen ergeben:

- Die Länge von `field->name`, dem Spaltentitel
- `field->max_length`, der Länge des längsten Datenwerts in der Spalte
- Der Länge der Zeichenkette »NULL«, falls in der Spalte `NULL`-Werte erlaubt sind. `field->flags` gibt an, ob die Spalte `NULL`-Werte enthalten darf

Beachten Sie, dass wir die Anzeigebreite einer Spalte, nachdem wir sie ermittelt haben, `max_length` zuweisen, einem Element einer Struktur, die wir aus der Client-Bibliothek erhalten. Ist das erlaubt, oder sollte der Inhalt der `MYSQL_FIELD`-Struktur schreibgeschützt sein? Normalerweise würde ich antworten »schreibgeschützt«, aber einige der Client-Programme in der MySQL-Distribution ändern den Wert von `max_length` auf ähnliche Weise. Ich nehme also an, es ist in Ordnung. (Wenn Sie einen alternativen Ansatz vorziehen, bei dem `max_length` nicht geändert wird, allozieren Sie ein Array mit `unsigned int`-Werten und legen die berechneten Breiten in diesem Array ab.)

Die Berechnung der Anzeigebreiten beinhaltet eine Tücke. Sie wissen, dass `max_length` keine Bedeutung hat, wenn Sie die Ergebnismenge mit `mysql_use_result()` erzeugen. Weil wir `max_length` brauchen, um die Anzeigebreite der Spaltenwerte zu ermitteln, ist es für eine korrekte Ausführung des Algorithmus erforderlich, dass Sie die Ergebnismenge mit `mysql_store_result()` erzeugen.[1]

Nachdem wir die Spaltenbreiten kennen, können wir mit der Ausgabe beginnen. Überschriften sind einfach zu realisieren. Wir verwenden für jede Spalte einfach nur die Struktur mit den Spalteninformationen, auf die `field` zeigt, und geben das Element `name` aus, wobei wir die zuvor berechnete Breite verwenden:

```
printf (" %-*s |", field->max_length, field->name);
```

Für die Ausgabe der Datenwerte durchlaufen wir die Zeilen in der Ergebnismenge und geben bei jeder Iteration die Spaltenwerte für die aktuelle Zeile aus. Die Ausgabe von Spaltenwerten aus der Zeile ist etwas kompliziert, weil ein Wert auch `NULL` sein, oder eine Zahl darstellen könnte (dann muss er rechts ausgerichtet werden). Spaltenwerte werden wie folgt ausgegeben, wobei `row[i]` den Datenwert enthält und `field` auf die Spalteninformation zeigt:

```
if (row[i] == NULL)
    printf (" %-*s |", field->max_length, "NULL");
else if (IS_NUM (field->type))
    printf (" %*s |", field->max_length, row[i]);
else
    printf (" %-*s |", field->max_length, row[i]);
```

Der Wert des Makros `IS_NUM()` ist `wahr`, wenn der durch `field->type` angegebene Spaltentyp ein numerischer Typ ist, beispielsweise `INT`, `FLOAT` oder `DECIMAL`.

Der endgültige Code zur Anzeige der Ergebnismenge sieht wie folgt aus. Beachten Sie, dass wir den Code zur Ausgabe der Zeilen mit den Trennstrichen, die wir mehrfach brauchen, in eine eigene Funktion eingekapselt haben, `print_dashes()`:

---

1. Das Element `length` der Struktur `MYSQL_FIELD` gibt die maximale Länge an, die ein Spaltenwert annimmt. Das kann eine sinnvolle Alternative sein, wenn Sie `mysql_use_result()` statt `mysql_store_result()` verwenden.

```
void
print_dashes (MYSQL_RES *res_set)
{
MYSQL_FIELD     *field;
unsigned int    i, j;

    mysql_field_seek (res_set, 0);
    fputc ('+', stdout);
    for (i = 0; i < mysql_num_fields (res_set); i++)
    {
        field = mysql_fetch_field (res_set);
        for (j = 0; j < field->max_length + 2; j++)
            fputc ('-', stdout);
        fputc ('+', stdout);
    }
    fputc ('\n', stdout);
}

void
process_result_set (MYSQL *conn, MYSQL_RES *res_set)
{
MYSQL_FIELD     *field;
MYSQL_ROW       row;
unsigned int    i, col_len;

    /* Spaltenanzeigebreite ermitteln */
    mysql_field_seek (res_set, 0);
    for (i = 0; i < mysql_num_fields (res_set); i++)
    {
        field = mysql_fetch_field (res_set);
        col_len = strlen (field->name);
        if (col_len < field->max_length)
            col_len = field->max_length;
        if (col_len < 4 && !IS_NOT_NULL (field->flags))
            col_len = 4;    /* 4 = Länge des Worts "NULL" */
        field->max_length = col_len;    /* Spalteninfo zurücksetzen */
    }

    print_dashes (res_set);
    fputc ('|', stdout);
    mysql_field_seek (res_set, 0);
    for (i = 0; i < mysql_num_fields (res_set); i++)
    {
        field = mysql_fetch_field (res_set);
        printf (" %-*s |", field->max_length, field->name);
    }
    fputc ('\n', stdout);
```

```
    print_dashes (res_set);

    while ((row = mysql_fetch_row (res_set)) != NULL)
    {
        mysql_field_seek (res_set, 0);
        fputc ('|', stdout);
        for (i = 0; i < mysql_num_fields (res_set); i++)
        {
            field = mysql_fetch_field (res_set);
            if (row[i] == NULL)
                printf (" %-*s |", field->max_length, "NULL");
            else if (IS_NUM (field->type))
                printf (" %*s |", field->max_length, row[i]);
            else
                printf (" %-*s |", field->max_length, row[i]);
        }
        fputc ('\n', stdout);
    }
    print_dashes (res_set);
    printf ("%lu Zeilen zurückgegeben\n", (unsigned long) mysql_num_rows
(res_set));
}
```

Die MySQL-Client-Bibliothek bietet mehrere Methoden, auf Strukturen mit Spalteninformationen zuzugreifen. Der Code im obigen Beispiel greift unter Verwendung von Schleifen der folgenden allgemeinen Form mehrmals auf die Strukturen zu:

```
mysql_field_seek (res_set, 0);
for (i = 0; i < mysql_num_fields (res_set); i++)
{
    field = mysql_fetch_field (res_set);
    ...
}
```

Die Kombination `mysql_field_seek() / mysql_fetch_field()` ist jedoch nur eine Methode, auf `MYSQL_FIELD`-Strukturen zuzugreifen. Weitere Möglichkeiten für den Zugriff auf Spalteninformationsstrukturen finden Sie in den Abschnitten zu `mysql_fetch_fields()` und `mysql_fetch_field_direct()` in Anhang F beschrieben.

6.7 Client 5 – Ein interaktives Anfrageprogramm

Jetzt wollen wir zusammentragen, was wir bisher entwickelt haben und damit einen einfachen interaktiven Client schreiben. Er erlaubt Ihnen, Anfragen einzugeben, führt sie mit Hilfe unserer allgemeinen Anfrageverarbeitung `process_query()` aus und zeigt das Ergebnis mit der im vorigen Abschnitt entwickelten Anzeigeformatierung `process_result_set()` an.

client5 ist mysql in mancher Hinsicht ganz ähnlich, bietet aber natürlich nicht so viele Funktionen. Für die Eingabe in client5 gibt es einige Einschränkungen:

- Jede Eingabezeile muss eine einzige, vollständige Anfrage enthalten.
- Anfragen sollten nicht durch ein Semikolon oder mit \g abgeschlossen werden.
- Befehle wie quit werden nicht erkannt; statt dessen verwenden Sie Strg-D, um das Programm zu beenden.

Es stellt sich heraus, dass client5 ganz einfach zu realisieren ist (weniger als zehn Zeilen neuer Code). Fast alles, was wir brauchen, ist bereits in unserem Client-Programmgerüst (client4.c) enthalten oder wird durch den anderen bisher entwickelten Code bereitgestellt. Wir brauchen nur noch eine Schleife, die Eingabezeilen entgegennimmt und ausführt.

Für die Entwicklung von client5 kopieren Sie zunächst das Client-Gerüst von client4.c in client5.c. Anschließend fügen Sie den Code für process_query(), process_result_set() und print_dashes() ein. Suchen Sie in client5.c nach der Zeile in main(), die wie folgt aussieht: :

```
/* hier passiert die eigentliche Arbeit */
```

Ersetzen Sie sie durch die folgende while-Schleife:

```
while (1)
{
char    buf[1024];
    fprintf (stderr, "query> ");     /* Eingabeaufforderung ausgeben */
    if (fgets (buf, sizeof (buf), stdin) == NULL)    /* Anfrage lesen */
        break;
    process_query (conn, buf);                  /* Anfrage ausführen */
}
```

Kompilieren Sie client5.c, um client5.o zu erzeugen, linken Sie dann client5.o mit common.o und der Client-Bibliothek, um client5 zu erzeugen – und fertig! Sie haben ein interaktives MySQL-Client-Programm, das beliebige Anfragen ausführt und das Ergebnis anzeigt.

6.8 Verschiedenes

In diesem Abschnitt geht es um verschiedene Dinge, die von der Entwicklung von client1 bis client5 nicht recht zum jeweiligen Themenbereich passten:

- Verwendung der Daten aus der Ergebnismenge, um ein Ergebnis zu berechnen, nachdem mit Hilfe der Metadaten aus der Ergebnismenge sichergestellt wurde, dass die Daten für Ihre Berechnungen geeignet sind
- Umgang mit Daten, deren Eingabe in Anfragen problematisch ist

- Bilddaten
- Informationen über die Struktur Ihrer Tabellen ermitteln
- Häufige Fehler bei der MySQL-Programmierung und wie Sie sie vermeiden

6.8.1 Berechnungen anhand von Ergebnismengen

Bisher haben wir die Metadaten aus den Ergebnismengen hauptsächlich für die Ausgabe von Zeilendaten verwendet, aber es gibt natürlich Situationen, wo man etwas anderes mit den Daten anfangen will, als sie nur auszugeben.

Das folgende Listing zeigt eine einfache Funktion, `summary_stats()`, die eine Ergebnismenge und einen Spaltenindex entgegennimmt und daraus statistische Werte für die Spalten berechnet. Die Funktion zeigt außerdem die Anzahl der fehlenden Werte an, die sie erkennt, indem sie auf `NULL`-Werte prüft. Für die Berechnung müssen die Daten zwei Voraussetzungen erfüllen, deshalb überprüft `summary_stats()` sie anhand der Metadaten aus der Ergebnismenge:

- Die angegebene Spalte muss existieren (d.h. der Spaltenindex muss eine Spalte in der Ergebnismenge angeben).
- Die Spalte muss numerische Werte enthalten.

Falls diese Bedingungen nicht zutreffen, gibt `summary_stats()` einfach eine Fehlermeldung aus und wird beendet. Der Code sieht wie folgt aus:

```
void
summary_stats (MYSQL_RES *res_set, unsigned int col_num)
{
MYSQL_FIELD     *field;
MYSQL_ROW       row;
unsigned int    n, missing;
double  val, sum, sum_squares, var;

    /* Bedingungen für die Daten überprüfen */
    if (mysql_num_fields (res_set) < col_num)
    {
        print_error (NULL, "Fehlerhafte Spaltennummer");
        return;
    }
    mysql_field_seek (res_set, 0);
    field = mysql_fetch_field (res_set);
    if (!IS_NUM (field->type))
    {
        print_error (NULL, "Spalte ist nicht numerisch");
        return;
    }
```

```
    /* Überblick berechnen */

    n = 0;
    missing = 0;
    sum = 0;
    sum_squares = 0;

    mysql_data_seek (res_set, 0);
    while ((row = mysql_fetch_row (res_set)) != NULL)
    {
        if (row[col_num] == NULL)
            missing++;
        else
        {
            n++;
            val = atof (row[col_num]);  /* Zeichenkette -> Zahl */
            sum += val;
            sum_squares += val * val;
        }
    }
    if (n == 0)
        printf ("Keine Beobachtungen\n");
    else
    {
        printf ("Anzahl Beobachtungen: %lu\n", n);
        printf ("Fehlende Beobachtungen: %lu\n", missing);
        printf ("Summe: %g\n", sum);
        printf ("Mittelwert: %g\n", sum / n);
        printf ("Summe der Quadrate: %g\n", sum_squares);
        var = ((n * sum_squares) - (sum * sum)) / (n * (n - 1));
        printf ("Varianz: %g\n", var);
        printf ("Standardabweichung: %g\n", sqrt (var));
    }
}
```

Beachten Sie den Aufruf von `mysql_data_seek()`, der der Schleife für `mysql_fetch_row()` vorausgeht. Er soll Ihnen ermöglichen, `summary_stats()` mehrfach für dieselbe Ergebnismenge aufzurufen (falls Sie statistische Werte für mehrere Spalten berechnen wollen). Bei jedem Aufruf von `summary_stats()` wird die Ergebnismenge an den Anfang »zurückgespult«. (Dabei wird vorausgesetzt, dass Sie die Ergebnismenge mit `mysql_store_result()` erzeugen. Wenn Sie sie mit `mysql_use_result()` erzeugen, können Sie die Zeilen nur der Reihe nach und nur ein einziges Mal verarbeiten.)

`summary_stats()` ist eine relativ einfache Funktion, aber Sie demonstriert, wie Sie komplexere Berechnungen realisieren könnten, beispielsweise eine Regression nach den kleinsten Quadraten über zwei Spalten oder Standard-Statistiken wie etwa einen *t*-Test.

6.8.2 Kodierung problematischer Daten in Anfragen

Die direkte Eingabe von Datenwerten, die Anführungszeichen, Nullen oder Backslashes enthalten, kann Probleme bei der Ausführung einer Anfrage verursachen.

Angenommen, Sie wollen eine `SELECT`-Anfrage entwickeln, die den Inhalt der null-terminierten Zeichenkette verwendet, auf die `name` zeigt:

```
char query[1024];

sprintf (query, "SELECT * FROM my_tbl WHERE name='%s'", name);
```

Falls der Wert von `name` etwas wie `"O'Malley, Brian"` ist, entsteht eine fehlerhafte Anfrage, weil innerhalb einer Zeichenkette in Anführungszeichen ein weiteres Anführungszeichen erscheint:

```
SELECT * FROM my_tbl WHERE name='O'Malley, Brian'
```

Sie müssen das Anführungszeichen speziell kennzeichnen, damit der Server es nicht als Ende des Namens interpretiert. Dazu könnten Sie das Anführungszeichen innerhalb der Zeichenkette doppelt angeben. Das ist die Konvention in ANSI SQL. MySQL versteht diese Konvention, erlaubt aber auch, dem Anführungszeichen einen Backslash voranzustellen.

```
SELECT * FROM my_tbl WHERE name='O''Malley, Brian'
SELECT * FROM my_tbl WHERE name='O\'Malley, Brian'
```

Eine weitere problematische Situation ergibt sich, wenn Sie beliebige Binärdaten in einer Anfrage verwenden. Das passiert beispielsweise in Anwendungen, die Bilder in einer Datenbank ablegen. Weil ein Binärwert beliebige Zeichen enthalten kann, ist es nicht sicher, ihn ohne Überprüfung in Anfragen zuzulassen.

Um dieses Problem zu lösen, verwenden Sie `mysql_escape_string()`, welches Sonderzeichen kodiert, so dass Sie sie in Zeichenketten einsetzen können, die in Anführungszeichen eingeschlossen sind. `mysql_escape_string()` berücksichtigt die folgenden Sonderzeichen: Nullzeichen, einfaches Anführungszeichen, doppeltes Anführungszeichen, Backslash, Neue Zeile, Return sowie Strg-Z als Sonderzeichen. (Das letztere tritt in Windows-Kontexten auf.)

Wann sollten Sie `mysql_escape_string()` verwenden? Die sicherste Antwort ist »immer«. Wenn Sie jedoch sicher sind, dass Ihre Daten in Ordnung sind – möglicherweise, weil Sie sie zuvor bereits überprüft haben – müssen Sie sie nicht kodieren. Wenn Sie beispielsweise mit Zeichenketten arbeiten, in denen korrekte Telefonnummern abgelegt sind, die nur aus Ziffern und Trennstrichen bestehen, brauchen Sie `mysql_escape()` überhaupt nicht aufzurufen. Andernfalls wäre es vielleicht doch eine gute Idee.

`mysql_escape_string()` kodiert problematische Zeichen, indem es sie in Folgen aus zwei Zeichen umwandelt, die mit einem Backslash beginnen. Eine Null beispielsweise wird zu `\0`, wobei die `0` eine druckbare ASCII-Null ist, keine Null. Backslash, einfache Anführungszeichen und doppelte Anführungszeichen werden zu `\\`, `\'` und `\"`.

`mysql_escape_string()` wird wie folgt aufgerufen:

```
to_len = mysql_escape_string (to_str, from_str, from_len);
```

`mysql_escape_string()` kodiert `from_str` und schreibt das Ergebnis in `to_str`. Außerdem fügt es eine abschließende Null ein, was sehr praktisch ist, weil Sie die resultierende Zeichenkette dann auch in Funktionen wie `strcpy()` und `strlen()` verwenden können.

`from_str` zeigt auf einen `char`-Puffer mit der zu kodierenden Zeichenkette. Diese Zeichenkette kann beliebigen Inhalt haben, auch binäre Daten. `to_str` zeigt auf einen existierenden `char`-Puffer, in den die kodierte Zeichenkette geschrieben werden soll. Übergeben Sie keinen nichtinitialisierten oder `NULL`-Zeiger und erwarten Sie, dass `mysql_excape_string()` Speicher für Sie reserviert. Die Länge des Puffers, auf den `to_str` zeigt, muss mindestens (`from_len`*2)+1 Bytes betragen. (Es könnte sein, dass jedes Zeichen aus `from_str` mit zwei Zeichen kodiert werden muss; das zusätzliche Byte ist für die terminierende Null vorgesehen.)

`from_len` und `to_len` sind `unsigned int`-Werte. `from_len` gibt die Länge der Daten in `from_str` an; diese Länge muss ermittelt werden, weil `from_str` null Bytes enthalten und dann nicht als null-terminierte Zeichenkette behandelt werden kann. `to_len`, der Rückgabewert von `mysql_escape_string()`, ist die Länge der resultierenden kodierten Zeichenkette, wobei die terminierende Null nicht berücksichtigt ist.

Das von `mysql_escape_string()` kodierte und in `to_str` abgelegte Ergebnis kann als null-terminierte Zeichenkette behandelt werden, weil alle Nullen aus `from_str` als die druckbare Folge `\0` kodiert wurden.

Um den Code für die Konstruktion einer `SELECT`-Anweisung so umzuformulieren, dass er auch für Namenswerte funktioniert, die Anführungszeichen enthalten, könnten wir etwa folgendes schreiben:

```
char query[1024], *p;

p = strcpy (query, "SELECT * FROM my_tbl WHERE name='");
p += strlen (p);
p += mysql_escape_string (p, name, strlen (name));
p = strcpy (p, "'");
```

Das ist ziemlich hässlich. Wenn Sie das Ganze vereinfachen wollen, könnten Sie Folgendes schreiben, müssen dafür jedoch einen zweiten Puffer bereitstellen:

```
char query[1024], buf[1024];

(void) mysql_escape_string (buf, name, strlen (name));
sprintf (query, "SELECT * FROM my_tbl WHERE name='%s'", buf);
```

6.8.3 Bilddaten

Eine der Aufgaben, für die Sie `mysql_escape_string()` unbedingt brauchen, ist das Laden von Bilddaten in eine Tabelle. Dieser Abschnitt beschreibt, wie das geht. (Das gilt auch für alle anderen Formen von Binärdaten.)

Angenommen, Sie wollen Bilder aus Dateien lesen und sie zusammen mit einem eindeutigen Bezeichner in einer Tabelle ablegen. Für Binärdaten ist der Typ `BLOB` am besten geeignet, Sie könnten also etwa die folgende Tabellenspezifikation verwenden:

```
CREATE TABLE images
(
    image_id INT NOT NULL PRIMARY KEY,
    image_data BLOB
)
```

Um ein Bild aus einer Datei in die Tabelle `images` zu laden, rufen Sie die folgende Funktion auf, `load_image()`. Übergeben Sie ihr dazu eine ID sowie einen Zeiger auf eine geöffnete Datei, in der die Bilddaten enthalten sind:

```
int
load_image (MYSQL *conn, int id, FILE *f)
{
char            query[1024*100], buf[1024*10], *p;
unsigned int    from_len;
int             status;

    sprintf (query, "INSERT INTO images VALUES (%d,'", id);
    p = query + strlen (query);
    while ((from_len = fread (buf, 1, sizeof (buf), f)) > 0)
    {
        /* nicht das Ende des Anfragepuffers übergehen! */
        if (p + (2*from_len) + 3 > query + sizeof (query))
        {
            print_error (NULL, "image too big");
            return (1);
        }
        p += mysql_escape_string (p, buf, from_len);
    }
    (void) strcpy (p, "')");
    status = mysql_query (conn, query);
    return (status);
}
```

`load_image()` reserviert nur einen kleinen Anfragepuffer (100 Kbyte), deshalb funktioniert es nur für relativ kleine Bilder. In einer realen Anwendung sollten Sie den Puffer abhängig von der Bildgröße dynamisch reservieren.

Die Verarbeitung von Bilddaten (oder anderen binären Daten), die Sie aus der Datenbank zurückladen wollen, ist nicht annähernd so kompliziert, wie sie dort abzulegen, weil die Datenwerte in der Variablen `MYSQL_ROW` in Rohform bereitstehen. Ihre Länge ermitteln Sie durch Aufruf von `mysql_fetch_lengths()`. Behandeln Sie die Werte jedoch immer als gezählte Zeichenketten, nicht als null-terminierte Zeichenketten.

6.8.4 Tabelleninformation ermitteln

MySQL ermöglicht Ihnen, Informationen über die Struktur Ihrer Tabellen zu ermitteln. Dazu verwenden Sie eine der folgenden Anfragen (die äquivalent sind):

```
DESCRIBE tbl_name
SHOW FIELDS FROM tbl_name
```

Beide Anweisungen verhalten sich wie SELECT und geben eine Ergebnismenge zurück. Um etwas über die Spalten in der Tabelle zu erfahren, verarbeiten Sie die Zeilen im Ergebnis, um die gewünschten Informationen zu extrahieren. Wenn Sie beispielsweise im `mysql`-Client die Anweisung `DESCRIBE images` ausführen, erhalten Sie die folgende Information:

```
+------------+---------+------+-----+---------+-------+
| Field      | Type    | Null | Key | Standard | Extra|
+------------+---------+------+-----+---------+-------+
| image_id   | int(11) |      | PRI | 0       |       |
| image_data | blob    | YES  |     | NULL    |       |
+------------+---------+------+-----+---------+-------+
```

Wenn Sie diese Anfrage auf Ihrem Client ausführen, erhalten Sie dieselbe Information (ohne die Rahmen).

Wenn Sie nur zu einer einzigen Spalte Informationen brauchen, verwenden Sie statt dessen diese Anfrage:

```
SHOW FIELDS FROM tbl_name LIKE "col_name"
```

Die Anfrage gibt dieselben Spalten zurück, aber nur eine Zeile (oder keine Zeilen, falls die Spalte nicht existiert).

6.8.5 Häufige Fehler bei der Client-Programmierung

Dieser Abschnitt beschreibt einige bei der Programmierung des MySQL-C-API häufig auftretende Fehler und wie man sie vermeidet. (Diese Probleme erscheinen immer wieder auf der MySQL-Mailing-Liste – ich habe sie nicht erfunden.)

Fehler 1 – Verwendung nicht initialisierter Zeiger auf Verbindungs-Handles

In den in diesem Kapitel gezeigten Beispielen haben wir `mysql_init()` aufgerufen und ihm das Argument `NULL` übergeben. `mysql_init()` alloziert und initialisiert eine `MYSQL`-Struktur und gibt einen Zeiger darauf zurück. Ein anderer Ansatz wäre, einen Zeiger auf eine bereits existierende `MYSQL`-Struktur zu übergeben. In diesem Fall initialisiert `mysql_init()` diese Struktur und gibt einen Zeiger darauf zurück, ohne die eigentliche Struktur zu allozieren. Wenn Sie diesen zweiten Ansatz verfolgen wollen, sollten Sie beachten, dass er zu subtilen Problemen führen kann. Die folgenden Beschreibungen sprechen einige Probleme an, die Sie im Auge behalten sollten.

Wenn Sie `mysql_init()` einen Zeiger übergeben, sollte dieser auf etwas zeigen. Betrachten Sie den folgenden Codeausschnitt:

```
main ()
{
MYSQL    *conn;
   mysql_init (conn);
   ...
}
```

Das Problem dabei ist, dass `mysql_init()` einen Zeiger entgegennimmt, dieser Zeiger aber auf nichts Sinnvolles zeigt. `conn` ist eine lokale Variable und damit ein nicht initialisierter Speicher, der auf irgendetwas zeigen kann, wenn die Ausführung von `main()` beginnt. Das bedeutet, `mysql_init()` verwendet den Zeiger und gelangt damit in einen beliebigen Speicherbereich. Wenn Sie Glück haben, zeigt `conn` auf eine Position außerhalb des Adressraums Ihres Programms, das System beendet dieses sofort und Sie erkennen, dass das Problem am Anfang Ihres Codes auftritt. Wenn Sie weniger Glück haben, zeigt `conn` auf irgendwelche Daten, die Sie erst später in Ihrem Programm verwenden, und Sie erkennen das Problem erst, wenn Ihr Programm versucht, diese Daten zu nutzen. In diesem Fall tritt Ihr Problem erst viel später auf als da, wo es entstanden ist, und kann schwer zurückverfolgt werden.

Hier ist ein ähnlicher Abschnitt problematischen Codes:

```
MYSQL    *conn;

main ()
{
    mysql_init (conn);
    mysql_real_connect (conn, ...)
    mysql_query(conn, "SHOW DATABASES");
    ...
}
```

In diesem Fall ist `conn` eine globale Variable, wird also mit `0` (d.h. `NULL`) initialisiert, bevor die Programmausführung beginnt. `mysql_init()` erhält ein `NULL`-Argument, deshalb initialisiert und reserviert es einen neuen Verbindungs-Handle. Leider ist `conn` immer noch `NULL`, weil ihm nie ein Wert zugewiesen wird. Sobald Sie `conn` einer MySQL-C-API-Funktion übergeben, die einen Verbindungs-Handle ungleich `NULL` benötigt, stürzt Ihr Programm ab. Eine Lösung für beide Codeabschnitte wäre, sicherzustellen, dass `conn` einen sinnvollen Wert enthält. Beispielsweise könnten Sie es mit der Adresse einer bereits reservierten `MYSQL`-Struktur initialisieren:

```
MYSQL conn_struct, *conn = &conn_struct;
...
mysql_init (conn);
```

Die empfohlene (und einfachere) Lösung ist, `mysql_init()` explizit `NULL` zu übergeben, es dieser Funktion zu überlassen, die `MYSQL`-Struktur für Sie zu reservieren, und `conn` den Rückgabewert zuzuweisen:

```
MYSQL *conn;
...
conn = mysql_init (NULL);
```

Vergessen Sie aber auch hier nicht, den Rückgabewert von `mysql_init()` zu überprüfen, um sicherzustellen, dass er nicht `NULL` ist.

Fehler 2 – Es wird nicht geprüft, ob eine gültige Ergebnismenge vorliegt

Denken Sie daran, den Status der Aufrufe zu überprüfen, von denen Sie eine Ergebnismenge erwarten. Der folgende Code macht das nicht:

```
MYSQL_RES *res_set;
MYSQL_ROW row;

res_set = mysql_store_result (conn);
while ((row = mysql_fetch_row (res_set)) != NULL)
{
    /* process row */
}
```

Wenn `mysql_store_result()` fehlschlägt, ist `res_set` gleich `NULL` und die `while`-Schleife wird leider nicht ausgeführt. Überprüfen Sie den Rückgabewert von Funktionen, die Ergebnismengen zurückgeben, um sicherzustellen, dass wirklich etwas vorliegt, mit dem Sie arbeiten können.

Fehler 3 – NULL-Spaltenwerte werden nicht berücksichtigt

Vergessen Sie nicht zu prüfen, ob die Spaltenwerte aus dem `MYSQL_ROW`-Array, die von `mysql_fetch_row()` zurückgegeben werden, `NULL`-Zeiger sind. Der folgende Code stürzt auf einigen Maschinen ab, falls `row[i]` gleich `NULL` ist:

```
for (i = 0; i < mysql_num_fields (res_set); i++)
{
    if (i > 0)
        fputc ('\t', stdout);
    printf ("%s", row[i]);
}
fputc ('\n', stdout);
```

Das Schlimmste bei diesem Fehler ist, dass einige Versionen von `printf()` tolerant sind und für `NULL`-Zeiger »(null)« ausgeben, so dass Sie weiterarbeiten können, ohne das Problem zu lösen. Wenn Sie Ihr Programm an einen Freund weitergeben, der ein weniger tolerantes `printf()` einsetzt, stürzt das Programm ab und Ihr Freund hält Sie für einen gnadenlos schlechten Programmierer. Die Schleife sollte deshalb wie folgt geschrieben werden:

```
for (i = 0; i < mysql_num_fields (res_set); i++)
{
    if (i > 0)
        fputc ('\t', stdout);
    printf ("%s", row[i] != NULL ? row[i] : "NULL");
}
fputc ('\n', stdout);
```

Nur wenn Sie bereits aus der Struktur mit den Spalteninformationen ermittelt haben, dass `IS_NOT_NULL()` gilt, brauchen Sie nicht zu prüfen, ob ein Spaltenwert gleich `NULL` ist.

Fehler 4 – Es werden keine sinnvollen Ergebnis-Puffer übergeben

Funktionen aus der Client-Bibliothek, für die Sie Puffer bereitstellen müssen, erwarten normalerweise, dass diese bereits existieren. Der folgende Code verletzt diese Forderung:

```
char *from_str = "some string";
char *to_str;
unsigned int len;

len = mysql_escape_string (to_str, from_str, strlen (from_str));
```

Wo ist das Problem? `to_str` muss auf einen existierenden Puffer zeigen. In diesem Beispiel ist das nicht der Fall – es zeigt auf irgendeine beliebige Position. Übergeben Sie `mysql_escape_string()` keinen nicht initialisierten Zeiger als `to_str`-Argument.