

# Kapitel 6

## Das C-API von MySQL

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

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

Dieses Kapitel zeigt die Verwendung der Clientbibliothek zur Entwicklung eigener Programme. Unter anderem achten wir dabei auf Konsistenz mit bereits existierenden Clientprogrammen in der MySQL-Distribution. Ich gehe davon aus, dass Sie gewisse Grundkenntnisse in der C-Programmierung haben, aber Sie müssen kein Experte sein, um die Beschreibungen in diesem Kapitel verstehen zu können.

In diesem Kapitel entwickeln wir mehrere Clientprogramme, von ganz einfachen bis hin zu immer komplexeren. Im ersten Abschnitt erstellen wir die Umgebung für ein Clientgerüst, das keine andere Aufgabe hat, als eine Verbindung zum Server einzurichten und diese zu trennen. (MySQL-Clientprogramme mögen zwar für unterschiedliche Aufgaben geschrieben sein, aber eines haben sie stets gemeinsam: Sie richten eine Verbindung zum Server ein.) Die Entwicklung des Gerüsts verläuft in folgenden Schritten:

- Entwicklung des reinen Codes zum Aufbauen und Schließen einer Verbindung (*client1*)
- Einfügen einer Fehlerüberprüfung (*client2*)
- Einführen einer Möglichkeit, zur Laufzeit Verbindungsparameter wie Host- und Benutzernamen sowie ein Kennwort anzugeben (*client3*)

Dieses Gerüst ist weitgehend generisch, und Sie können es als Grundlage für die unterschiedlichsten Clientprogramme 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 Clientgerüst den Code für die Anfrageverarbeitung hinzu, um ein weiteres Programm (*client4*) zu entwickeln, das dem *mysql*-Client ganz ähnlich ist und zum interaktiven Absetzen von Anfragen verwendet werden kann.

Danach werden wir zeigen, wie man von zwei Funktionen profitiert, die seit MySQL 4 vorhanden sind. Es wird erläutert werden, wie man

- Clientprogramme schreibt, die mithilfe von SSL (Secure Sockets Layer) über sichere Verbindungen mit dem Server kommunizieren, und wie man
- Anwendungen schreibt, die die eingebettete Serverbibliothek *libmysqld* benutzen.

Abschließend werden wir auf häufig auftretende Probleme eingehen, z. B., wie man Informationen über die Struktur seiner Tabellen erhält und wie man Bilder in die Datenbank einfügen kann.

Dieses Kapitel beschreibt Funktionen und Datentypen aus der Clientbibliothek nur nach Bedarf. Eine vollständige Auflistung aller Funktionen und Typen finden Sie in Anhang F, »C-API-Referenz«. Sie können diesen Anhang auch als Referenz für Hintergrundinformationen zu jedem Bestandteil der Clientbibliothek verwenden, die Sie benutzen wollen.

Die Beispielprogramme stehen online zum Download bereit, Sie brauchen sie also nicht selbst einzugeben. Sie finden sie im Verzeichnis `\capi` der *sampdb*-Distribution. Anhang A, »Software beschaffen und installieren«, beschreibt, wie Sie die Dateien herunterladen.

### Wo man Beispielprogramme findet

Häufig taucht in der MySQL-Mailingliste die Frage auf, wo man denn Beispiele für Clients in C fände. Die Antwort lautet natürlich: »In diesem Buch!« Viele Benutzer sehen gar nicht, dass die MySQL-Distribution mehrere Clientprogramme enthält (unter anderem *mysql*, *mysqladmin* oder *mysqldump*), die in C geschrieben sind. Weil die Distribution als Quellcode vorliegt, finden Sie direkt in MySQL Beispiele für Clientcode. Sehen Sie sich die Programme im Verzeichnis `\client` der Distribution an, falls Sie das noch nicht getan haben.

## 6.1 Allgemeine Vorgehensweise bei der Entwicklung von Clientprogrammen

Dieser Abschnitt beschreibt das Kompilieren und Linken eines Programms, das die MySQL-Clientbibliothek nutzt. Die dabei verwendeten Befehle unterscheiden sich bei den verschiedenen Systemen, weshalb es vorkommen kann, dass Sie die hier gezeigten Befehle anpassen müssen. Die Beschreibung ist jedoch allgemein genug gehalten, sodass Sie sie auf fast jedes von Ihnen entwickelte Clientprogramm anwenden können.

## 6.1.1 Grundsätzliche Systemanforderungen

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

- MySQL-Header-Dateien
- MySQL-Clientbibliothek

Die MySQL-Header-Dateien und die Clientbibliothek bilden die Basis der Clientprogrammierung. Sofern sie nicht bereits auf Ihrem System installiert sind, müssen Sie sie sich beschaffen. Wurde MySQL aus einer Quellcode- oder Binärdistribution installiert, dann sollte die Unterstützung für die Clientprogrammierung bereits installiert sein; wurde MySQL hingegen aus RPM-Dateien installiert, so verfügen Sie nur dann über diese Unterstützung, wenn Sie die Entwickler-RPM installiert haben. Falls Sie die MySQL-Header-Dateien und die Bibliothek installieren müssen, finden Sie weitere Informationen in Anhang A.

## 6.1.2 Clientprogramme kompilieren und linken

Um ein Clientprogramm zu kompilieren und zu linken, müssen Sie angeben, wo sich die MySQL-Header-Dateien und die Clientbibliothek befinden, weil diese normalerweise nicht dort vorhanden sind, wo Compiler und Linker danach suchen. In den folgenden Beispielen gehen wir davon aus, dass sich die Header-Dateien und die Clientbibliothek in den Verzeichnissen */usr/local/include/mysql* bzw. */usr/local/lib/mysql* befinden.

Um dem Compiler mitzuteilen, wo er die MySQL-Header-Dateien findet, wenn Sie eine Quellcodedatei kompilieren, übergeben Sie ihm das Argument *-I*. Um etwa aus *myclient.c* die Objektdatei *myclient.o* zu machen, verwenden Sie den folgenden Befehl:

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

Damit der Linker weiß, wo er die Clientbibliothek 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 Clientprogramm aus mehreren Dateien besteht, geben Sie im Link-Befehl alle Objektdateien an.

Der Link-Schritt kann Fehlermeldungen ausgeben, die mit Funktionen zu tun haben, die nicht gefunden wurden. In solchen Fällen müssen Sie zusätzliche *-l*-Optionen angeben, um die Bibliotheken zu benennen, die die Funktionen enthalten. Wenn Sie eine Meldung bezüglich *compress()* oder *uncompress()* erhalten, fügen Sie versuchsshalber *-lz* bzw. *-lgz* hinzu, damit der Linker weiß, dass er die Komprimierungsbibliothek *zlib* suchen soll:

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

Falls beim Linken ein Fehler auftritt, der besagt, dass die Funktion `floor()` nicht gefunden werden konnte, binden Sie die Mathematikbibliothek ein, indem Sie hinter dem Befehl ein `-lm` einfügen. So werden Sie etwa unter Solaris die Optionen `-lsocket` und `-lnsl` ergänzen müssen.

Seit MySQL 3.23.21 können Sie das Dienstprogramm `mysql_config` verwenden, um die passenden Flags zum Kompilieren und Linken von MySQL-Programmen zu ermitteln. Dieses Dienstprogramm kann beispielsweise anzeigen, dass die folgenden Optionen benötigt werden:

```
% mysql_config --cflags
-I'/usr/local/mysql/include/mysql'
% mysql_config --libs
-L'/usr/local/mysql/lib/mysql' -lmysqlclient -lz -lcrypt -lnsl -lm
```

Um `mysql_config` direkt innerhalb der Kompilierungs- oder Link-Befehle zu verwenden, rufen Sie es in Backticks gesetzt auf:

```
% gcc -c `mysql_config --cflags` myclient.c
% gcc -o myclient myclient.o `mysql_config --libs`
```

Die Shell wird `mysql_config` ausführen und seine Ausgabe in den umgebenden Befehl einsetzen, d.h. es werden direkt die passenden Flags für `gcc` gesetzt.

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 Clientprogramm `myclient`, das aus zwei Quelldateien (`main.c` und `aux.c`) und einer Header-Datei (`myclient.h`) besteht. Ein einfache Steuerdatei zur Programmerstellung – ein so genanntes Makefile – könnte wie folgt aussehen (beachten Sie, dass manche Zeilen mit Tabulatoren eingezogen sind; würden Sie stattdessen Leerzeichen verwenden, dann würde das Makefile nicht funktionieren):

```
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
```

Mithilfe des Makefiles können Sie Ihr Programm nach jeder Änderung der Quelldateien neu erstellen, indem Sie einfach `make` eingeben. Auf diese Weise wird der gesamte Befehl inklusive aller Optionen aufgerufen:

```
% make
gcc -c -I/usr/local/mysql/include/mysql myclient.c
gcc -o myclient myclient.o -L/usr/local/mysql/lib/mysql -lmysqlclient
```

Das ist einfacher und weniger fehleranfällig als die Eingabe eines langen *gcc*-Befehls. Ein Makefile erleichtert auch den Kompilierungsvorgang. Wenn Sie beispielsweise zusätzliche Bibliotheken in Ihr System einbinden wollen – z.B. Mathematik- oder Komprimierungsbibliotheken –, dann müssen Sie die LIBS-Zeile im Makefile mit *-lm* und *-lz* ergänzen:

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

Alle sonstigen Bibliotheken, die Sie benötigen, müssen Sie ebenfalls in der LIBS-Zeile hinzufügen. Wenn Sie danach *make* ausführen, wird der aktualisierte LIBS-Wert automatisch verwendet.

Eine weitere Möglichkeit, *make*-Variablen zu ändern, ohne das Makefile zu editieren, besteht darin, sie über die Befehlszeile anzugeben. Wenn Ihr C-Compiler etwa *cc* statt *gcc* heißt (was etwa bei Mac OS X der Fall ist), dann können Sie Folgendes eingeben:

```
% make CC=cc
```

Steht Ihnen *mysql\_config* zur Verfügung, dann können Sie es verwenden, um die Pfadnamen von Include-Dateien und Bibliotheksverzeichnissen nicht in das Makefile schreiben zu müssen. Stattdessen sehen die Zeilen für *INCLUDES* und *LIBS* wie folgt aus:

```
INCLUDES = ${shell mysql_config --cflags}
LIBS = ${shell mysql_config --libs}
```

Wenn *make* läuft, führt es jeden *mysql\_config*-Befehl aus und verwendet die Ausgabe zur Definition des jeweiligen Variablenwerts. Das hier gezeigte Konstrukt *\${shell}* wird von der GNU-Version von *make* unterstützt. Basiert Ihre *make*-Version nicht auf GNU, dann müssen Sie möglicherweise eine etwas andere Syntax verwenden.

Wenn Sie eine integrierte Entwicklungsumgebung (Integrated Development Environment, IDE) verwenden, dann können Sie ein Makefile möglicherweise nicht finden oder verwenden. Wie sich das genau verhält, hängt von der verwendeten IDE ab.

## 6.2 Client 1: Verbindung zum Server

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

Der Quellcode unseres ersten Clientprogramms *client1* besteht aus einer einzigen Datei namens *client1.c*:

```
/* client1.c - Verbindung mit dem MySQL-Server herstellen und trennen */

#include <my_global.h>
#include <mysql.h>

static char *opt_host_name = NULL;      /* Host (Standard: localhost) */
static char *opt_user_name = NULL;     /* Benutzername (Standard: Anmelde-
                                         name) */
static char *opt_password = NULL;     /* Kennwort (Standard: keines) */
static unsigned int opt_port_num = 0;  /* Portnummer (Standardwert verwenden) */
static char *opt_socket_name = NULL;   /* Socketname (Standardwert verwenden) */
static char *opt_db_name = NULL;      /* Datenbankname (Standard: keiner) */
static unsigned int opt_flags = 0;     /* Verbindungsflags (keine) */

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

int
main (int argc, char *argv[])
{
    /* Verbindungs-Handle initialisieren */
    conn = mysql_init (NULL);
    /* Serververbindung aufbauen */
    mysql_real_connect (conn, opt_host_name, opt_user_name, opt_password,
                       opt_db_name, opt_port_num, opt_socket_name, opt_flags);
    /* Serververbindung trennen */
    mysql_close (conn);
    exit (0);
}
```

Die Quelldatei beginnt mit dem Einfügen der Header-Dateien *my\_global.h* und *mysql.h*. Je nach Einsatzzweck des MySQL-Clients können Sie auch weitere Header-Dateien einbinden, aber im Allgemeinen brauchen Sie zumindest diese beiden:

- *my\_global.h* bindet mehrere andere Header-Dateien ein, die gemeinhin als nützlich gelten, so etwa *stdio.h*. Zwecks Kompatibilität wird für den Fall, dass Sie das Programm unter Windows kompilieren, auch die Datei *windows.h* eingebunden (dies ist auch dann notwendig, wenn Sie das Programm zwar nicht unter Windows erstellen, Ihren Code aber vertreiben wollen, denn jeder Benutzer, der das Programm unter Windows kompilieren will, braucht *windows.h*).
- *mysql.h* definiert die primären MySQL-bezogenen Konstanten und Datenstrukturen.

Die Reihenfolge, in der Sie Header-Dateien einbinden, ist wichtig: *my\_global.h* sollte vor allen anderen MySQL-spezifischen Header-Dateien eingebunden werden.

Als Nächstes deklariert das Programm Variablen, die den Parametern entsprechen, die beim Aufbau der Serververbindung angegeben werden müssen. Bei diesem Client sind die Werte fest vorgegeben; wir werden später einen flexibleren Ansatz entwickeln, der das Überschreiben der Standardwerte mit Angaben aus einer Optionsdatei oder Befehlszeileneingaben erlaubt (dies ist auch der Grund dafür, dass alle Namen mit `opt_` beginnen; wir haben vor, diese Variablen später über Befehlsoptionen einstellen zu lassen). Außerdem deklariert das Programm einen Zeiger auf einen `MYSQL`-Datentyp, der als Verbindungs-Handle dient.

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

1. Sie rufen `mysql_init()` auf, um einen Verbindungs-Handle zu erhalten. Wenn Sie `NULL` an `mysql_INIT()` übergeben, reserviert es eine `MYSQL`-Variable, initialisiert sie und gibt einen Zeiger darauf zurück. Der `MYSQL`-Datentyp ist eine Struktur mit Informationen über eine Verbindung. Variablen dieses Typs werden als Verbindungs-Handle bezeichnet.
2. Rufen Sie `mysql_real_connect()` auf, um eine Verbindung zum Server herzustellen. `mysql_real_connect()` nimmt eine Unmenge Parameter entgegen:
  - **Einen Zeiger auf den Verbindungs-Handle.** Dieser Parameter sollte nicht `NULL` sein, sondern der von `mysql_init()` zurückgegebene Wert.
  - **Serverhost.** Dieser Wert wird plattformspezifisch interpretiert. 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. Wenn Sie hingegen `NULL` oder den Host `localhost` angeben, stellt der Client unter Verwendung eines Unix-Sockets eine Verbindung zu dem Server auf dem lokalen Host her.  
Unter Windows liegt ein ähnliches Verhalten vor, nur werden statt Unix-Sockets TCP/IP-Verbindungen genutzt. Unter Windows NT wird zudem vor TCP/IP versucht, eine Named Pipe zu verwenden, falls der Host `.` oder `NULL` ist.
  - **Benutzername und Kennwort.** Ist der Name `NULL`, dann sendet die Clientbibliothek Ihren Anmeldenamen an den Server. Ist das Kennwort `NULL`, so wird kein Kennwort gesendet.
  - **Name der Datenbank, die nach Herstellung der Verbindung als Standarddatenbank gewählt wird.** Wenn der Wert `NULL` ist, wird keine Datenbank ausgewählt.
  - **Portnummer und Socket-Datei.** Die Portnummer wird für TCP/IP-Verbindungen benötigt, der Socket-Name für Socket-Verbindungen (unter Unix) bzw. für Named Pipes (unter Windows) benutzt. Die Werte `0` und `NULL` weisen die Clientbibliothek an, die Standardwerte zu verwenden.
  - **flags-Wert.** Er ist `0`, weil wir keine speziellen Verbindungsoptionen verwenden werden.

Weitere Informationen finden Sie bei der Beschreibung von `mysql_real_connect()` in Anhang F, »C-API-Referenz«. Die dortige Erläuterung

beschreibt eingehend, was der Parameter *hostname* mit der Portnummer und dem Socket-Namen zu tun hat, und listet außerdem die Optionen auf, die mit dem Parameter *flags* angegeben werden können. Ferner beschreibt der Anhang auch `mysql_options()`, das Sie verwenden können, um andere verbindungsbezogene Optionen vor dem Aufruf von `mysql_real_connect()` anzugeben.

Um die Verbindung zu beenden, rufen Sie `mysql_close()` auf und übergeben der Funktion 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 das Programm *client1* auszuprobieren, kompilieren und linken Sie es, wie bereits beschrieben, und führen es dann aus. Unter Unix sieht der Aufruf wie folgt aus:

```
% ./client1
```

Die vorangestellten Zeichen `./` können unter Unix notwendig sein, wenn Ihre Shell kein aktuelles Verzeichnis (`.`) im Suchpfad aufweist. Wenn das Verzeichnis im Suchpfad vorhanden ist oder Sie Windows verwenden, dann können Sie `./` weglassen:

```
% client1
```

Das Programm stellt eine Verbindung zum Server her, baut sie wieder ab und wird beendet. Das ist nicht besonders aufregend, aber es ist ein Anfang – mehr ist es aber auch nicht, weil es zwei wichtige Mängel aufweist:

- 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 über die Befehlszeile bereitstellt.

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

## 6.3 Client 2: Fehlerprüfung ergänzen

Unser zweiter Client ist dem ersten ganz ähnlich, wird jedoch um eine Fehlerprüfung ergänzt. In Programmierbüchern findet man häufig die Ansicht, die Fehlerprüfung solle dem Leser als Übung überlassen bleiben; das liegt – sagen wir doch, wie es ist – vermutlich daran, dass die Fehlerprüfung so langweilig ist! Dennoch möchte ich betonen, dass in MySQL-Clientprogrammen unbedingt eine Fehlerprüfung stattfinden und eine entsprechende Verarbeitung erfolgen sollte. Die Aufrufe der Clientbibliothek geben die Statuswerte nicht grundlos zurück, und Sie sollten sie nicht ignorieren. Irgendwann treten seltene Probleme in Ihren Programmen auf, die Sie nicht lösen können, weil Sie es

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:

```
020816 21:52:14      20 Connect   sampadm@localhost on
                   20 Quit
```

Möglicherweise sehen Sie aber auch einen Eintrag `Access denied`, der angibt, dass überhaupt keine Verbindung hergestellt wurde:

```
020816 22:01:47      21 Connect   Access denied for user: 'sampadm@localhost'
                   (Using password: NO)
```

Leider teilt uns *client1* nicht mit, was genau passiert ist – das Programm kann es schlichtweg nicht. Es nimmt keine Fehlerprüfung vor, deshalb weiß es selbst nicht, was passiert ist. Dies ist bei Clientprogrammen nicht akzeptabel; Sie sollten jedenfalls 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-Clientbibliothek, 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` ist in diesem Kontext ein `NULL`-Zeiger in C und kein `NULL`-Spaltenwert in MySQL). Von den Routinen der Clientbibliothek, die wir bisher verwendet haben, geben sowohl `mysql_init()` als auch `mysql_real_connect()` 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 `0`. Dabei ist es wichtig, auf bestimmte Werte ungleich Null abzufragen, beispielsweise `-1`. Es ist nicht garantiert, dass eine Funktion der Clientbibliothek einen bestimmten Wert zurückgibt, wenn sie fehlgeschlagen ist; hin und wieder stößt man auch auf Code, der einen Rückgabewert der API-Funktion `mysql_XXX()` fehlerhaft überprüft:

```
if (mysql_XXX() == -1)      /* dieser Test ist nicht korrekt */
    fprintf(stderr, "something bad happened\n");
```

Diese Überprüfung kann funktionieren, muss es 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() != 0)      /* dieser Test ist korrekt */
    fprintf(stderr, "something bad happened\n");
```

So geht es auch (und ist zudem etwas einfacher zu schreiben):

```
if (mysql_XXX())          /* dieser Test ist korrekt */
    fprintf (stderr, "something bad happened\n");
```

Wenn Sie einen Blick in den Quellcode von MySQL werfen, werden Sie feststellen, dass im Allgemeinen die zweite Testform verwendet wird.

Nicht jeder API-Aufruf erzeugt einen Rückgabewert. Die andere Clientroutine, 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 Clientbibliothek fehlschlägt und Sie mehr Informationen brauchen. `mysql_error()` gibt einen String mit der Fehlermeldung zurück, `mysql_errno()` einen numerischen Fehlercode. Das Argument beider Funktionen ist ein Zeiger auf den Verbindungs-Handle. Sie sollten die Funktion 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 der Benutzer eines Programms eine 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 einführenden Erläuterungen entwickeln wir jetzt unseren zweiten Client, *client2*. Er ist ähnlich wie *client1* aufgebaut, besitzt aber Code, der für die Fehlerprüfung geeignet ist. Die Quelldatei *client2.c* sieht wie folgt aus:

```
/*
 * client2.c - Verbindung mit dem MySQL-Server herstellen und trennen,
 * mit Fehlerprüfung
 */

#include <my_global.h>
#include <mysql.h>

static char *opt_host_name = NULL;    /* Host (Standard: localhost) */
static char *opt_user_name = NULL;    /* Benutzername (Standard: Anmeldename) */
static char *opt_password = NULL;    /* Kennwort (Standard: keines) */
static unsigned int opt_port_num = 0; /* Portnummer (Standardwert verwenden) */
static char *opt_socket_name = NULL;  /* Socketname (Standardwert verwenden) */
static char *opt_db_name = NULL;      /* Datenbankname (Standard: keiner) */
static unsigned int opt_flags = 0;    /* Verbindungsflags (keine) */

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

```

int
main (int argc, char *argv[])
{
    /* Verbindungs-Handle initialisieren */
    conn = mysql_init (NULL);
    if (conn == NULL)
    {
        fprintf (stderr, "mysql_init() failed (probably out of memory)\n");
        exit (1);
    }
    /* Serververbindung aufbauen */
    if (mysql_real_connect (conn, opt_host_name, opt_user_name, opt_password,
        opt_db_name, opt_port_num, opt_socket_name, opt_flags) == NULL)
    {
        fprintf (stderr, "mysql_real_connect() failed:\nError %u (%s)\n",
            mysql_errno (conn), mysql_error (conn));
        mysql_close (conn);
        exit (1);
    }
    /* Serververbindung trennen */
    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, dass 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. Der Handle kann auch an `mysql_close()` übergeben werden, um Speicher freizugeben, der eventuell automatisch von `mysql_init()` zugewiesen worden war. (Übergeben Sie den Handle jedoch nicht an andere Clientroutinen! Weil sie im Allgemeinen eine gültige Verbindung voraussetzen, könnte Ihr Programm abstürzen.)

Kompilieren und linken Sie *client2*, und führen Sie es dann aus:

```
% ./client2
```

Falls *client2* (wie gerade gezeigt) 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: 'sampadm@localhost' (Using password: NO))

```

Diese Ausgabe zeigt an, dass keine Verbindung hergestellt wurde, und teilt Ihnen auch den Grund dafür mit. Außerdem zeigt sie, dass unser erstes Pro-

ogramm *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.

Wenn Sie über potenzielle Probleme Bescheid wissen, ist das besser, als wenn Sie keine Ahnung haben. Deshalb sollten Sie die Rückgabewerte der API-Funktionen immer auswerten. Wenn Sie dies unterlassen, bringen Sie sich bei der Programmierung in unnötige Schwierigkeiten. Dieses Phänomen wird auch in der MySQL-Mailingliste sehr häufig angesprochen. 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. Und wenn ein Programm keine Fehlerprüfung enthält, dann kann der Programmierer ganz schön ins Schwimmen kommen. Glauben Sie nicht, dass alle Aufrufe der Clientbibliothek erfolgreich seien!

Die restlichen Programmbeispiele in diesem Kapitel führen eine Fehlerprüfung aus, und Ihre eigenen Programme sollten das auch tun. Es scheint zusätzlichen Aufwand zu bedeuten, aber letztendlich sparen Sie damit Zeit, wenn schwer erkennbare Probleme auftreten. Diesen Ansatz der Fehlerprüfung werde ich auch in den Kapiteln 7, »Das Perl-DBI-API«, und 8, »Das PHP-API«, verwenden.

Nehmen wir nun einmal an, Sie erhalten bei der Ausführung des Programms *client2* die Meldung `Access denied`. Wie können Sie das Problem lösen? Eine Möglichkeit bestünde darin, das Programm nach einer Modifikation des Quellcodes neu zu kompilieren. (Die Modifikation bewirkt, dass die Initialisierungswerte für die Verbindung so gesetzt werden, dass ein Zugriff auf Ihren Server möglich ist). Auf diese Weise könnten Sie zumindest eine Verbindung einrichten. Die Werte sind dann aber immer noch in Ihrem Programm festgeschrieben. Ich rate von dieser Vorgehensweise insbesondere in Hinblick auf den Kennwortwert ab. 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 (übrigens erfährt jeder, der den Quellcode Ihres Programms lesen kann, das Kennwort ohne jeden Aufwand).

Im nächsten Abschnitt werden wir flexiblere Methoden entwickeln, um anzuzeigen, wie die Verbindungsherstellung zum Server erfolgt. Zunächst jedoch möchte ich eine einfache Methode zur Fehleranzeige entwickeln, weil dies etwas ist, was wir wirklich häufig benötigen. Ich werde weiterhin den kombinierten Anzeigestil aus numerischem MySQL-Fehlercode und der Fehlerbeschreibung verwenden, aber ich werde die Aufrufe der Fehlerfunktionen `mysql_errno()` und `mysql_error()` nicht mehr jedes Mal wie folgt ausschreiben:

```

if (...MySQL-Funktion fehlschlägt...)
{
    fprintf(stderr, "...Fehlermeldung...:\nError %u (%s)\n",
            mysql_errno (conn), mysql_error (conn)); tbl_name
}

```

Stattdessen ist es einfacher, Fehler mithilfe einer Hüllfunktion anzuzeigen, die wie folgt aufgerufen wird:

```

if (...MySQL-Funktion fehlschlägt...)
{
    print_error (conn, "...Fehlermeldung...");
}

```

`print_error()` druckt die Fehlermeldung und ruft die MySQL-Fehlerfunktionen automatisch auf. Es ist einfacher, `print_error()` auszuschreiben, als einen langen `fprintf()-`Aufruf, und überdies ist das Programm dann leichter zu lesen. Außerdem können wir, wenn `print_error()` so geschrieben ist, dass es auch dann irgendetwas Sinnvolles tut, wenn `conn` den Wert `NULL` hat, es auch in Situationen verwenden, in denen etwa der Aufruf von `mysql_init()` fehlschlägt. In diesem Fall hätten wir auch keinen unübersichtlichen Mix aus Aufrufen von Fehleranzeigefunktionen nach dem Motto »ein wenig `fprintf()` hier, ein bisschen `print_error()` dort«. Eine Version von `print_error()`, die diesem Anforderungsprofil genügt, könnte wie folgt aussehen:

```

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

```

Von den hinteren Bänken höre ich jetzt Einwände: »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; eigentlich müssten Sie ihn nur einmal schreiben und ihn dann bei Bedarf einfach kopieren und einfügen«. Diese Einwände sind vernünftig, aber ich habe ihnen Folgendes entgegenzusetzen:

- Selbst wenn Sie Kopieren und 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, werden Sie irgendwann versucht sein, Abkürzungen zu schaffen, und diese 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, dies an zentraler Stelle zu erledigen, als mehrfach im gesamten Programm. Und wenn Sie beschließen, die Fehlermeldungen in eine Protokolldatei statt (oder zusätzlich) in `stderr` zu schreiben, ist es einfacher, dafür nur `print_error()` ändern zu 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, sodass das Programm in den Debugger wechselt, sobald es eine Fehlerbedingung erkennt.

Aus diesem Grund werden wir im verbleibenden Kapitel `print_error()` zur Berichterstattung über MySQL-bezogene Probleme verwenden.

## 6.4 Client 3: Verbindungsparameter zur Laufzeit übergeben

An dieser Stelle sind wir nun so weit, dass wir überlegen können, wie wir statt der festgeschriebenen Verbindungsparameter etwas Schlaues verwenden könnten, indem wir es beispielsweise dem Benutzer überlassen, diese Werte zur Laufzeit einzugeben. Der vorherige Client 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 mithilfe von Befehlszeilenoptionen übergeben. Die Programme der MySQL-Distribution unterstützen zwei Methoden zur Übergabe von Verbindungsparametern, die in Tabelle 6.1 beschrieben sind.

Parameter	Langform	Kurzform
Hostname	<code>--host=host_name</code>	<code>-h host_name</code>
Benutzername	<code>--user=user_name</code>	<code>-u user_name</code>
Kennwort	<code>--password=oder --password=your_pass</code>	<code>-p oder -pyour_pass</code>
Portnummer	<code>--port=port_num</code>	<code>-P port_num</code>
Socket-Name	<code>--socket=socket_name</code>	<code>-S socket_name</code>

Tabelle 6.1: Methoden zur Übergabe von Verbindungsparametern

Der Konsistenz mit den MySQL-Standardclients halber soll unser Client dieselben Formate unterstützen. Das ist einfach zu implementieren, weil die Clientbibliothek eine Funktion zur Auswertung der Optionen enthält. Darüber hinaus bietet unser Client die Möglichkeit, Informationen aus Optionsdateien abzurufen. Damit können Sie Verbindungsparameter in `~/my.cnf` (d.h. in der

Datei *.my.cnf* in Ihrem Basisverzeichnis) oder einer beliebigen globalen Optionsdatei ablegen, sodass es nicht mehr nötig ist, sie beim Programmaufruf in der Befehlszeile einzugeben. Die Clientbibliothek macht es einfach, nach MySQL-Optionsdateien zu suchen und alle relevanten Werte daraus zu beziehen. Durch ein paar zusätzliche Zeilen implementieren Sie in Ihrem Code die Unterstützung von Optionsdateien – und das, ohne das Rad neu erfinden zu müssen (die Syntax für Optionsdateien ist in Anhang E, »MySQL-Programmreferenz«, beschrieben).

Bevor wir uns an das Schreiben von *client3* machen, müssen wir erst ein paar Programme entwickeln, die zeigen, wie die Unterstützung der Optionsverarbeitung bei MySQL funktioniert. An diesen Programm kann einfach demonstriert werden, wie die Optionsnutzung abläuft, ohne dass man erst eine Verbindung mit dem MySQL-Server herstellen und komplizierte Anfragen absetzen muss.

### 6.4.1 Zugriff auf den Inhalt von Optionsdateien

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

Das kleine Programm *show\_argv* demonstriert im Folgenden, wie `load_defaults()` verwendet wird und Ihren Argumentvektor verändert:

```
/* show_argv.c - Einfluss von load_defaults() auf den Argumentvektor zeigen */

#include <my_global.h>
#include <mysql.h>

static const char *client_groups[] = { "client", NULL };

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

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

```
my_init ();
load_defaults ("my", client_groups, &argc, &argv);

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

exit (0);
}
```

Der Code zur Optionsdateiverarbeitung beinhaltet die folgenden Komponenten:

- `client_groups[]` ist ein String-Array, das angibt, an welchen Optionsdateigruppen Sie interessiert sind. Für Clientprogramme geben Sie immer mindestens `client` ein (das steht für die Gruppe `[client]` in der Optionsdatei), aber Sie können so viele Gruppen aufführen, wie Sie wollen. Das letzte Element des Arrays muss `NULL` sein, damit klar ist, wo die Liste endet.
- `my_init()` ist eine Initialisierungsroutine, die verschiedene Konfigurationsoperationen 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 Namen der 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, dass Sie aber dennoch `&argv` übergeben, die Adresse dieses Zeigers.

`show_argv` gibt seine Argumente zweimal aus: zuerst so, wie Sie sie auf der Befehlszeile eingegeben haben, und dann so, wie sie von `load_defaults()` abgeändert wurden. Um die Arbeitsweise von `load_defaults()` nachvollziehen zu können, sollte die Datei `.my.cnf`, in der sich Einstellungen für die Gruppe `[client]` befinden, in Ihrem Basisverzeichnis vorhanden sein (unter Windows verwenden Sie z.B. die Datei `c:\my.cnf`). Angenommen, die Datei sieht wie folgt aus:

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

In diesem Fall erzeugt `show_argv` die folgende Ausgabe:

```
% ./show_argv a b
Original argument vector:
arg 0: ./show_argv
arg 1: a
arg 2: b
Modified argument vector:
arg 0: ./show_argv
arg 1: --user=sampadm
```

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

Wenn *show\_argv* den Argumentvektor beim zweiten Mal angibt, werden die Werte in der Optionsliste als Teil der Argumentliste angezeigt. Möglicherweise werden auch Optionen angegeben, die sich weder in der Befehlszeile noch in Ihrer Datei *~/.my.cnf* befinden; in diesem Fall wurden diese Optionen höchstwahrscheinlich in einer systemübergreifenden Optionsdatei abgelegt. Ein solches Verhalten kann auftreten, weil `load_defaults()` mehrere Optionsdateien überprüft: Unter Unix werden vor dem Auslesen der Datei *.my.cnf* im Basisverzeichnis die Datei */etc/my.cnf* sowie die Datei *my.cnf* im Datenverzeichnis von MySQL ausgelesen, unter Windows die Datei *my.ini* im Windows-Systemverzeichnis, die Datei *c:\my.cnf* sowie ebenfalls die Datei *my.cnf* im Datenverzeichnis von MySQL.

Clientprogramme, die `load_defaults()` verwenden, geben in der Optionsgruppenliste fast immer `client` an (sodass sie alle allgemeinen Clientinstellungen aus Optionsdateien beziehen können); Sie können aber auch Werte anderer Gruppen anfordern. Nehmen wir etwa einmal an, Sie wollen *show\_argv* so abändern, dass die Optionen der Gruppen `[client]` und `[show_argv]` ausgelesen werden; um dies zu ändern, suchen Sie die folgende Zeile im Programm *show\_argv.c*:

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

Ändern Sie die Zeile wie folgt ab:

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

Kompilieren Sie *show\_argv* nun neu. Das geänderte Programm liest nun die Optionen beider Gruppen aus. Sie können dies überprüfen, indem Sie eine Gruppe `[show_argv]` in Ihrer Datei *~/.my.cnf* hinzufügen:

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

```
[show_argv]
host=other_host
```

Nach diesen Änderungen erzeugt der Aufruf von *show\_argv* ein anderes Ergebnis als zuvor:

```
% ./show_argv a b
Original argument vector:
arg 0: ./show_argv
arg 1: a
arg 2: b
Modified argument vector:
arg 0: ./show_argv
```

```
arg 1: --user=sampadm
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 `client_groups[]` aufgelistet sind. Das bedeutet, dass Sie in Ihrer Optionsdatei programmspezifische Gruppen hinter der Gruppe `[client]` angeben sollten. Wenn Sie eine Option in beiden Gruppen angeben, hat der programmspezifische Wert Vorrang. 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 Vorrang.

`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 mithilfe von `getenv()` selbst bewerkstelligen. Ich werde unseren Clients diese Möglichkeit nicht hinzufügen, aber das folgende Beispiel zeigt, wie die Werte einiger Standardumgebungsvariablen für MySQL ausgewertet werden können:

```
extern char *getenv();
char *p;
int port_num = 0;
char *socket_name = NULL;

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 Werte aus Optionsdateien oder von der Befehlszeile erhaltene Werte. Wenn Sie Umgebungsvariablen auswerten und dieser Konvention entsprechen wollen, werten Sie sie aus, *bevor* Sie `load_defaults()` aufrufen oder Befehlszeilenoptionen verarbeiten.

### `load_defaults()` und die Sicherheit

Auf Mehrbenutzersystemen können Dienstprogramme wie etwa das Programm `ps` Argumentlisten beliebiger Prozesse anzeigen, also auch derjenigen, die von anderen Benutzern ausgeführt werden. Angesichts dieser Tatsache fragen Sie sich vielleicht, ob es in Zusammenhang mit dem Einsatz von `load_defaults()` keine Bedenken in Hinblick auf einen möglichen Diebstahl von Kennwörtern gibt, die in den Optionsdateien vorhanden sind und von dort in Ihre Argumentliste übergeben werden.

Sie können allerdings beruhigt sein, denn *ps* zeigt den Originalinhalt von `argv[]` an; Kennwortargumente, die von `load_defaults()` erstellt werden, verweisen auf einen Speicherbereich, dessen Einsatz `load_defaults()` selbst vorbehalten ist; dieser Bereich ist nicht Teil des ursprünglichen Vektors, d.h., *ps* bekommt ihn niemals zu sehen.

Im Gegensatz dazu kann ein Kennwort, das über die Befehlszeile übergeben wird, *tatsächlich* mit *ps* erkannt werden. Dies ist einer von mehreren Gründen dafür, warum man Kennwörter nicht auf diese Art und Weise übergeben sollte. Eine Vorsichtsmaßnahme, die in Programme implementiert werden kann, um das Risiko zu verringern, besteht darin, das Kennwort unmittelbar nach Beginn der Ausführung aus der Argumentliste zu entfernen. Im nächsten Abschnitt, »Befehlszeilenargumente auswerten«, wird demonstriert, wie das funktioniert.

## 6.4.2 Befehlszeilenargumente auswerten

Mithilfe von `load_defaults()` können wir alle Verbindungsparameter in den Argumentvektor laden, aber nun benötigen wir auch eine Möglichkeit, den Vektor zu verarbeiten. Zu diesem Zweck ist die Funktion `handle_options()` vorgesehen. `handle_options()` ist Bestandteil der MySQL-Clientbibliothek, d.h., Sie können darauf zugreifen, wenn Sie diese Bibliothek einbinden.

Die hier beschriebenen Methoden zur Verarbeitung von Optionen wurden mit MySQL 4.0.2 eingeführt. Davor enthielt die Clientbibliothek einen Code zur Optionsverarbeitung, der auf der Funktion `getopt_long()` basierte. Dieser Code wurde durch eine neue Schnittstelle ersetzt, die eben auf `handle_options()` fußt.<sup>1</sup> Die neuen Routinen weisen eine Reihe von Verbesserungen auf:

- **Präzisere Angaben zu Typ und Wertebereich zulässiger Optionswerte.** Sie können nun beispielsweise nicht nur festlegen, dass eine Option ein Integerwert sein muss, sondern auch, dass dieser Wert positiv und ein Vielfaches von 1024 sein muss.
- **Integration von Hilfetexten, um durch Aufruf einer Standardbibliotheksfunktion Hilfmeldungen leichter anzeigen zu können.** Sie müssen nun keinen speziellen Code mehr schreiben, um Hilfmeldungen zu erzeugen.
- **Integrierter Support für die Standardoptionen `--no-defaults`, `--print-defaults`, `--defaults-file` und `--defaults-extra-file`.** Diese Optionen werden in Abschnitt E.1.3, »Optionsdateien«, beschrieben.

---

1. Wenn Sie MySQL-basierte Programme mit der Clientbibliothek einer MySQL-Version vor 4.0.2 entwickeln, lesen Sie die Version dieses Kapitels, die Sie in der ersten Auflage dieses Buchs finden. Dort wird beschrieben, wie man mit `getopt_long()` Befehlsoptionen verarbeitet. Das Kapitel steht in englischer und deutscher Sprache als PDF-Datei auf der Begleit-Website zu diesem Buch (<http://www.kitebird.com/mysql-book>) zum Download bereit.

- Unterstützung einer Standardmenge von Optionspräfixen wie etwa `--disable-` und `--enable-`, um die Implementierung Boole'scher Optionen zu erleichtern. Diese Funktionalität wird nicht in diesem Kapitel verwendet, aber im Abschnitt zur Optionsverarbeitung in Anhang E beschrieben.

## HINWEIS

Die neuen Optionsverarbeitungsroutinen tauchten zwar bereits in MySQL 4.0.2 auf, aber Sie sollten am besten die Version 4.0.5 oder höher verwenden. Während des Einführungszeitraums wurden verschiedene Probleme in Zusammenhang mit diesen Funktionen erkannt und beseitigt.

Um zu zeigen, wie man die MySQL-Funktionen zur Optionsverwaltung einsetzen kann, beschreiben wir in diesem Abschnitt das Programm `show_opt`, das `load_defaults()` aufruft, um die Optionsdateien auszulesen und den Argumentvektor zu konfigurieren, und das Ergebnis dann mit `handle_options()` verarbeitet.

`show_opt` erlaubt uns, mit den verschiedenen Möglichkeiten zur Angabe von Verbindungsparametern (über Optionsdateien oder die Befehlszeile) zu experimentieren und das Ergebnis – die Parameter, die für eine Verbindung zum MySQL-Server verwendet werden würden – dann zu betrachten. `show_opt` ist praktisch, um ein Gefühl dafür zu entwickeln, was in unserem nächsten Clientprogramm, `client3`, passieren wird, das diesen Code zur Optionsverarbeitung mit dem Code verknüpfen wird, der tatsächlich eine Serververbindung herstellt.

`show_opt` zeigt uns in jeder Phase der Argumentverarbeitung, was geschieht, indem es die folgenden Vorgänge ausführt:

1. Die Standardparameter für Host- und Benutzername, Kennwort und andere Verbindungsoptionen werden konfiguriert.
2. Die ursprünglichen Werte der Verbindungsparameter und des Argumentvektors werden ausgegeben.
3. `load_defaults()` wird aufgerufen, um den Argumentvektor neu zu schreiben, sodass er den Inhalt der Optionsdatei widerspiegelt. Der modifizierte Vektor wird ausgegeben.
4. Die Optionsverarbeitungsroutine `handle_options()` wird aufgerufen, um den Argumentvektor zu verarbeiten und die resultierenden Werte der Verbindungsparameter (und alles, was im Argumentvektor noch so übrig ist) auszugeben.

Im Folgenden werden wir erläutern, wie `show_opt` funktioniert, aber zunächst wollen wir einmal einen Blick auf den Quellcode `show_opt.c` werfen:

```
/*
 * show_opt.c - demonstriert die Optionsverarbeitung mit
 * load_defaults() und handle_options()
 */
```

```
#include <my_global.h>
#include <mysql.h>
#include <my_getopt.h>

static char *opt_host_name = NULL;      /* Host (Standard: localhost) */
static char *opt_user_name = NULL;     /* Benutzername (Standard: Anmeldenname) */
static char *opt_password = NULL;     /* Kennwort (Standard: keines) */
static unsigned int opt_port_num = 0;  /* Portnummer (Standardwert verwenden) */
static char *opt_socket_name = NULL;  /* Socketname (Standardwert verwenden) */

static const char *client_groups[] = { "client", NULL };

static struct my_option my_opts[] =    /* Optionsdatenstrukturen */
{
    {"help", '?', "Display this help and exit",
     NULL, NULL, NULL,
     GET_NO_ARG, NO_ARG, 0, 0, 0, 0, 0, 0},
    {"host", 'h', "Host to connect to",
     (gptr *) &opt_host_name, NULL, NULL,
     GET_STR_ALLOC, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
    {"password", 'p', "Password",
     (gptr *) &opt_password, NULL, NULL,
     GET_STR_ALLOC, OPT_ARG, 0, 0, 0, 0, 0, 0},
    {"port", 'P', "Port number",
     (gptr *) &opt_port_num, NULL, NULL,
     GET_UINT, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
    {"socket", 'S', "Socket path",
     (gptr *) &opt_socket_name, NULL, NULL,
     GET_STR_ALLOC, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
    {"user", 'u', "User name",
     (gptr *) &opt_user_name, NULL, NULL,
     GET_STR_ALLOC, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
    { NULL, 0, NULL, NULL, NULL, NULL, GET_NO_ARG, NO_ARG, 0, 0, 0, 0, 0, 0 }
};

my_bool
get_one_option (int optid, const struct my_option *opt, char *argument)
{
    switch (optid)
    {
        case '?':
            my_print_help (my_opts);    /* Hilfemeldung drucken */
            exit (0);
        }
    return (0);
}

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

```
int i;
int opt_err;

printf ("Original connection parameters:\n");
printf ("host name: %s\n", opt_host_name ? opt_host_name : "(null)");
printf ("user name: %s\n", opt_user_name ? opt_user_name : "(null)");
printf ("password: %s\n", opt_password ? opt_password : "(null)");
printf ("port number: %u\n", opt_port_num);
printf ("socket name: %s\n", opt_socket_name ? opt_socket_name : "(null)");

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

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

printf ("Modified argument vector after load_defaults():\n");
for (i = 0; i < argc; i++)
    printf ("arg %d: %s\n", i, argv[i]);

if ((opt_err = handle_options (&argc, &argv, my_opts, get_one_option)))
    exit (opt_err);

printf ("Connection parameters after handle_options():\n");
printf ("host name: %s\n", opt_host_name ? opt_host_name : "(null)");
printf ("user name: %s\n", opt_user_name ? opt_user_name : "(null)");
printf ("password: %s\n", opt_password ? opt_password : "(null)");
printf ("port number: %u\n", opt_port_num);
printf ("socket name: %s\n", opt_socket_name ? opt_socket_name : "(null)");

printf ("Argument vector after handle_options():\n");
for (i = 0; i < argc; i++)
    printf ("arg %d: %s\n", i, argv[i]);

exit (0);
}
```

Der Ansatz zur Optionsverarbeitung, den *show\_opt.c* illustriert, berücksichtigt die folgenden gemeinsamen Aspekte aller Programme, die zur Verarbeitung der Befehlsoptionen die MySQL-Clientbibliothek verwenden:

1. Neben den Header-Dateien *my\_global.h* und *mysql.h* wird auch *my\_getopt.h* mit eingebunden. *my\_getopt.h* definiert die Schnittstelle zu den Optionsverarbeitungsfunktionen von MySQL.
2. Es wird ein Array mit *my\_option*-Strukturen definiert. In *show\_opt.c* heißt dieses Array *my\_opts*. Es sollte pro vom Programm verstandener Option eine Struktur enthalten. Jede Struktur bietet Informationen wie etwa die kurzen

und langen Namen einer Option, ihren Standardwert, ob der Wert numerisch oder ein String ist usw. Details zu den Komponenten der Struktur `my_option` erhalten Sie in Kürze.

3. Nachdem `load_defaults()` aufgerufen wurde, um die Optionsdateien zu lesen und den Argumentvektor zu konfigurieren, werden die Optionen mit `handle_options()` verarbeitet. Die ersten beiden Argumente für `handle_options()` sind die Adressen des Argumentzählers und des Argumentvektors Ihres Programms (wie bereits bei `load_options()` übergeben Sie die Adressen dieser Variablen, nicht ihre Werte). Das dritte Argument verweist auf das Array mit den `my_option`-Strukturen, das vierte auf eine Hilfsfunktion. Die Routine `handle_options()` und die `my_options`-Strukturen sollen sicherstellen, dass die meisten Operationen bei der Optionsverarbeitung von der Clientbibliothek automatisch durchgeführt werden. Um aber auch spezielle Aktionen zu ermöglichen, die die Bibliothek nicht verarbeitet, sollte Ihr Programm zudem eine Hilfsfunktion definieren, die `handle_options()` aufrufen kann. Bei `show_opt.c` heißt diese Funktion `get_one_option()`. Ihre Verwendung werden wir weiter unten beschreiben.

Die Struktur `my_option` definiert die Datentypen, die für jede vom Programm verstandene Option angegeben werden müssen. Sie sieht wie folgt aus:

```
struct my_option
{
    const char *name;           /* langer Optionsname */
    int        id;             /* kurzer Optionsname oder Code */
    const char *comment;       /* Optionsbeschreibung für Hilfemeldung */
    gptr       *value;         /* Zeiger auf die Variable, in der der Wert
    ↳                                     gespeichert wird */
    gptr       *u_max_value;   /* benutzerdefinierter Maximalwert der
    ↳                                     Variablen */
    const char **str_values;    /* Array mit zulässigen Optionswerten (nicht
    ↳                                     verwendet) */
    enum get_opt_var_type var_type; /* Typen der Optionswerte */
    enum get_opt_arg_type arg_type; /* gibt an, ob der Wert vorhanden sein muss */
    longlong   def_value;      /* Standardwert der Option */
    longlong   min_value;      /* zulässiger Mindestwert der Option */
    longlong   max_value;      /* zulässiger Höchstwert der Option */
    longlong   sub_size;       /* Wertversatz */
    long       block_size;     /* Optionswertmultiplikator */
    int        app_type;       /* reserviert für anwendungsspezifische
    ↳                                     Verwendung */
};
```

Die Komponenten der Struktur `my_option` sind die folgenden:

- **name.** Der lange Optionsname. Dies ist die Form `--name` der Option, nur ohne vorangestellte Striche. Wenn beispielsweise der lange Optionsname `--user` heißt, dann wird er als `user` in der Struktur `my_option` geführt.
- **id.** Der kurze (aus einem Buchstaben bestehende) Optionsname oder – bei dessen Fehlen – ein Codewert, der mit der Option verknüpft ist. Heißt etwa

die Kurzoption `-u`, dann wird sie als `u` in der Struktur `my_option` geführt. Für solche Optionen, die nur einen langen Namen, aber keinen entsprechenden Einzeichennamen haben, sollten Sie eine Menge von Optionscodewerten definieren, die intern als Kurznamen verwendet werden. Die Werte müssen eindeutig sein und sich von allen anderen Einzeichennamen unterscheiden. (Um diese Bedingung zu erfüllen, erstellen Sie Codes mit Werten über 255, denn dies ist der höchste Wert, der sich mit einem Zeichen darstellen lässt. Ein Beispiel hierfür zeigt Abschnitt 6.7, »Clients mit SSL-Support entwickeln«.)

- **comment.** Dies ist ein Erklärungsstring, der den Zweck der Option beschreibt und ggf. als Hilfemitteilung angezeigt wird.
- **value.** Hierbei handelt es sich um einen `gptr`-Wert (generischer Zeiger). Er verweist auf die Variable, die das Argument der Option speichert. Wenn alle Optionen verarbeitet sind, können Sie diese Variable überprüfen, um zu sehen, wie der Optionswert gesetzt wurde. Wenn die Option kein Argument übernimmt, kann `value` auch `NULL` sein; andernfalls muss der Datentyp der Variable, auf die verwiesen wird, zum Wert der Komponente `var_type` konsistent sein.
- **u\_max\_value.** Dies ist ein weiterer `gptr`-Wert, der aber nur vom Server benutzt wird. Bei Clientprogrammen setzen Sie diesen Wert auf `NULL`.
- **str\_values.** Diese Komponente ist derzeit noch nicht in Verwendung. In zukünftigen MySQL-Versionen wird sie verwendet werden, um eine Liste erlaubter Werte angeben zu können, mit der alle übergebenen Optionswerte auf Zulässigkeit verglichen werden.
- **var\_type.** Diese Komponente zeigt, welcher Wert dem Optionsnamen in der Befehlszeile folgen muss. Hierbei sind die in Tabelle 6.2 angegebenen Werte zulässig.

Wert von <code>var_type</code>	Bedeutung
<code>GET_NO_ARG</code>	kein Wert
<code>GET_BOOL</code>	Boole'scher Wert
<code>GET_INT</code>	Integerwert
<code>GET_UINT</code>	Integerwert ohne Vorzeichen
<code>GET_LONG</code>	langer Integerwert
<code>GET_ULONG</code>	langer Integerwert ohne Vorzeichen
<code>GET_LL</code>	long long-Integerwert
<code>GET_ULL</code>	long long-Integerwert ohne Vorzeichen
<code>GET_STR</code>	Stringwert
<code>GET_STR_ALLOC</code>	Stringwert

Tabelle 6.2: Zulässige Werte für `var_type`

Der Unterschied zwischen `GET_STR` und `GET_STR_ALLOC` besteht darin, dass die Optionsvariable bei `GET_STR` so gesetzt wird, dass sie direkt auf den Wert im Argumentvektor verweist, wohingegen bei `GET_STR_ALLOC` eine Kopie des Arguments angefertigt wird, auf die die Optionsvariable dann verweist.

- `arg_type`. Dieser Wert gibt an, ob ein Wert im Optionsnamen folgt. Hierbei sind die in Tabelle 6.3 angegebenen Werte zulässig.

Wert von <code>arg_type</code>	Bedeutung
<code>NO_ARG</code>	Die Option übernimmt kein nachfolgendes Argument.
<code>OPT_ARG</code>	Die Option kann ein nachfolgendes Argument übernehmen.
<code>REQUIRED_ARG</code>	Die Option benötigt ein nachfolgendes Argument.

Tabelle 6.3: Zulässige Werte für `arg_type`

Hat `arg_type` den Wert `NO_ARG`, dann sollte `var_type` auf `GET_NO_ARG` gesetzt werden.

- **def\_value**. Bei Optionen mit numerischen Werten wird der Option standardmäßig dieser Wert zugewiesen, sofern kein expliziter Wert im Argumentvektor angegeben ist.
- **min\_value**. Bei Optionen mit numerischen Werten ist dies der kleinste spezifizierbare Wert. Kleinere Werte werden automatisch auf diesen Wert gesetzt. Wählen Sie 0, um die Einstellung »kein Mindestwert« zuzuweisen.
- **max\_value**. Bei Optionen mit numerischen Werten ist dies der größte spezifizierbare Wert. Größere Werte werden automatisch auf diesen Wert gesetzt. Wählen Sie 0, um die Einstellung »kein Höchstwert« zuzuweisen.
- **sub\_size**. Bei Optionen mit numerischen Werten ist dies ein Versatz, der benutzt wird, um Werte aus dem Bereich, der im Argumentvektor angegeben ist, in den intern verwendeten Bereich zu konvertieren. Wenn beispielsweise die in der Befehlszeile angegebenen Werte im Bereich 1...256 liegen, das Programm jedoch intern den Bereich 0...255 nutzt, dann muss `sub_size` den Wert 1 haben.
- **block\_size**. Bei Optionen mit numerischen Werten gibt dieser Wert, sofern er ungleich Null ist, eine Blockgröße an. Die Optionswerte werden bei Bedarf auf das nächstkleinere Vielfache dieser Blockgröße abgerundet. Wenn etwa die Werte gerade sein müssen, definieren Sie eine Blockgröße von 2, damit `handle_options()` ungerade Werte auf die nächste gerade Zahl abrundet.
- **app\_type**. Diese Komponente ist für eine anwendungsspezifische Verwendung reserviert.

Das Array `my_opts` sollte für jede gültige Option eine `my_option`-Struktur aufweisen. Am Ende sollte dann eine Abschlusstruktur stehen, die wie folgt konfiguriert wird, um das Ende des Arrays anzuzeigen:

```
{ NULL, 0, NULL, NULL, NULL, NULL, GET_NO_ARG, NO_ARG, 0, 0, 0, 0, 0, 0 }
```

Wenn Sie `handle_options()` aufrufen, um den Argumentvektor zu bearbeiten, dann überspringt die Funktion das erste Argument (den Programmnamen) und verarbeitet die Optionsargumente, d.h. Argumente, die mit einem Bindestrich beginnen. Dieser Vorgang setzt sich fort, bis das Ende des Vektors erreicht ist oder das spezielle Optionsabschlussargument (`--`) gefunden wird. Während sich `handle_options()` durch den Argumentvektor arbeitet, ruft es einmal pro Option die Hilfsfunktion auf, damit diese ggf. eine Spezialverarbeitung durchführen kann. `handle_options()` übergibt der Hilfsfunktion drei Argumente, nämlich den kurzen Optionswert, einen Zeiger auf die `my_option`-Struktur der Option und einen weiteren Zeiger auf das Argument, das der Option im Argumentvektor folgt (und das, sofern kein anderer Wert angegeben ist, den Wert `NULL` hat).

Wenn `handle_options()` seine Arbeit abgeschlossen hat, werden der Argumentzähler und der Argumentvektor entsprechend zurückgesetzt, um eine Argumentliste anzuzeigen, die nur die Argumente enthält, die nicht mit Optionen verbunden sind.

Das Folgende ist ein Beispielaufruf für `show_opt` und die entsprechende Ausgabe (hierbei wird davon ausgegangen, dass `~/my.cnf` noch immer den gleichen Inhalt hat wie beim letzten `show_argv`-Beispiel in Abschnitt 6.4.1, »Zugriff auf den Inhalt von Optionsdateien«):

```
% ./show_opt -h yet_another_host --user=bill x
Original connection parameters:
host name: (null)
user name: (null)
password: (null)
port number: 0
socket name: (null)
Original argument vector:
arg 0: ./show_opt
arg 1: -h
arg 3: yet_another_host
arg 3: --user=bill
arg 4: x
Modified argument vector after load_defaults():
arg 0: ./show_opt
arg 1: --user=sampadm
arg 2: --password=secret
arg 3: --host=some_host
arg 4: -h
arg 5: yet_another_host
arg 6: --user=bill
arg 7: x
Connection parameters after handle_options():
host name: yet_another_host
user name: bill
password: secret
port number: 0
```

```
socket name: (null)
Argument vector after handle_options():
arg 0: x
```

Die Ausgabe zeigt, dass der Hostname über die Befehlszeile ermittelt wurde (und damit Vorrang vor dem Wert in der Optionsdatei hat), während der Benutzername und das Kennwort aus der Optionsdatei stammen. `handle_options()` erkennt alle Option korrekt – unabhängig davon, ob sie in ihrer Kurzform (`-h yet_another_host`) oder mit langem Namen (`--user=bill`) angegeben wurden.

Die Hilfsfunktion `get_one_option()` wird in Verbindung mit `handle_options()` verwendet. Bei `show_opt` ist sie vergleichsweise minimalistisch und lediglich für die Optionen `--help` bzw. `-?` zuständig (für die `handle_options()` den `optid`-Wert ? übergibt):

```
my_bool
get_one_option(int optid, const struct my_option *opt, char *argument)
{
    switch (optid)
    {
        case '?':
            my_print_help(my_opts); /* Hilfefeldung drucken */
            exit(0);
    }
    return(0);
}
```

`my_print_help()` ist eine Routine der Clientbibliothek, die automatisch basierend auf den Optionsnamen und den Kommentarstrings im `my_opts`-Array eine Hilfemitteilung generiert. Um einmal zu sehen, wie das funktioniert, geben Sie den folgenden Befehl ein (die Hilfemitteilung steht ganz hinten in der Ausgabe):

```
% ./show_opt --help
```

Sie können der Funktion `get_one_option()` nach Bedarf weitere Fälle hinzufügen. Sie ist beispielsweise zur Verarbeitung von Kennwortoptionen nützlich. Wenn Sie eine solche Option angeben, dann kann, wie in der Optionsinformationsstruktur `OPT_ARG` angegeben, ein Kennwortwert übergeben oder eben nicht übergeben werden (d.h. Sie können die Option als `--password` oder `--password=ihr_kennwort` übergeben, wenn Sie die lange Optionsform verwenden, oder als `-p` bzw. `-pihr_kennwort`, wenn Sie die kurze Form vorziehen). In der Regel erlauben Ihnen MySQL-Clients, den Kennwortwert in der Befehlszeile wegzulassen, und fordern dann später zur Eingabe des Kennworts auf. Auf diese Weise wird verhindert, dass Sie Ihr Kennwort in der Befehlszeile angeben müssen, wo es von anderen Personen ausgespäht werden könnte. In späteren Programmen werden wir `get_one_option()` verwenden, um zu prüfen, ob ein Kennwortwert übergeben wurde oder nicht. Wir werden den Wert speichern, wenn ein Kennwort vorhanden ist, oder andernfalls ein Flag setzen, um anzuzeigen, dass das Programm den Benutzer zur Eingabe eines Kennworts auffordern soll, bevor es eine Verbindung mit dem Server herzustellen versucht.

Vielleicht finden Sie es lehrreich, die Optionsstrukturen in *show\_opt.c* zu modifizieren, um zu sehen, wie sich ihre Änderungen auf das Verhalten des Programms auswirkt. Wenn Sie beispielsweise für die Option `--port` die Mindest-, Höchst- und Blockgrößenwerte auf 100, 1000 bzw. 25 setzen, werden Sie nach der Neukompilierung des Programms feststellen, dass Sie keine Portnummern außerhalb eines Bereichs zwischen 100 und 1000 einstellen können und dass die eingegebenen Werte automatisch auf das nächstliegende Vielfache von 25 abgerundet werden.

Die Routinen zur Optionsverarbeitung verwenden die Optionen `--no-defaults`, `--print-defaults`, `--defaults-file` und `--defaults-extra-file` automatisch. Rufen Sie *show\_opt* einmal mit jeder dieser Optionen auf, um zu sehen, was passiert.

### 6.4.3 Optionsverarbeitung in einem MySQL-Clientprogramm implementieren

So, nun wollen wir aber den Teil aus *show\_opt.c* entfernen, der lediglich zeigen sollte, wie die Routinen zur Optionsverarbeitung funktionieren, und den Rest als Basis eines Clients verwenden, der entsprechend den in einer Optionsdatei vorhandenen oder über die Befehlszeile übergebenen Optionen eine Verbindung mit einem Server herstellt. Die daraus resultierende Quellcodedatei *client3.c* sieht wie folgt aus:

```

/*
 * client3.c - Verbindung mit dem MySQL-Server unter Verwendung
 * von in einer Optionsdatei oder über die Befehlszeile
 * angegebenen Parametern
 */

#include <string.h>      /* für strdup() */
#include <my_global.h>
#include <mysql.h>
#include <my_getopt.h>

static char *opt_host_name = NULL;      /* Host (Standard: localhost) */
static char *opt_user_name = NULL;     /* Benutzername (Standard: Anmeldenname) */
static char *opt_password = NULL;     /* Kennwort (Standard: keines) */
static unsigned int opt_port_num = 0;  /* Portnummer (Standardwert verwenden) */
static char *opt_socket_name = NULL;   /* Socketname (Standardwert verwenden) */
static char *opt_db_name = NULL;      /* Datenbankname (Standard: keiner) */
static unsigned int opt_flags = 0;     /* Verbindungsflags (keine) */

static int ask_password = 0;           /* Kennworteingabe erzwingen? */
static MYSQL *conn;                   /* Zeiger auf Verbindungs-Handle */

static const char *client_groups[] = { "client", NULL };

static struct my_option my_opts[] =    /* Optionsdatenstrukturen */
{

```

```

{"help", '?', "Display this help and exit",
NULL, NULL, NULL,
GET_NO_ARG, NO_ARG, 0, 0, 0, 0, 0, 0},
{"host", 'h', "Host to connect to",
(gptr *) &opt_host_name, NULL, NULL,
GET_STR_ALLOC, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
{"password", 'p', "Password",
(gptr *) &opt_password, NULL, NULL,
GET_STR_ALLOC, OPT_ARG, 0, 0, 0, 0, 0, 0},
{"port", 'P', "Port number",
(gptr *) &opt_port_num, NULL, NULL,
GET_UINT, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
{"socket", 'S', "Socket path",
(gptr *) &opt_socket_name, NULL, NULL,
GET_STR_ALLOC, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
{"user", 'u', "User name",
(gptr *) &opt_user_name, NULL, NULL,
GET_STR_ALLOC, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
{ NULL, 0, NULL, NULL, NULL, NULL, GET_NO_ARG, NO_ARG, 0, 0, 0, 0, 0, 0 }
};

```

```

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

```

```

my_bool
get_one_option (int optid, const struct my_option *opt, char *argument)
{
    switch (optid)
    {
    case '?':
        my_print_help (my_opts);    /* Hilfemeldung drucken */
        exit (0);
    case 'p':
        /* Kennwort */
        if (!argument)
            /* Kein Wert angegeben, deshalb später
            anfordern */
            ask_password = 1;
        else
            /* Kennwort kopieren, Original löschen */
            {
                opt_password = strdup (argument);
                if (opt_password == NULL)
                    {

```

```
        print_error (NULL, "could not allocate password buffer");
        exit (1);
    }
    while (*argument)
        *argument++ = 'x';
    }
    break;
}
return (0);
}

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

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

if ((opt_err = handle_options (&argc, &argv, my_opts, get_one_option))
    exit (opt_err);

/* Kennwort ggf. anfordern */
if (ask_password)
    opt_password = get_tty_password (NULL);

/* Datenbanknamen holen, soweit in der Befehlszeile vorhanden */
if (argc > 0)
{
    opt_db_name = argv[0];
    --argc; ++argv;
}

/* Verbindungs-Handle initialisieren */
conn = mysql_init (NULL);
if (conn == NULL)
{
    print_error (NULL, "mysql_init() failed (probably out of memory)");
    exit (1);
}

/* Serververbindung herstellen */
if (mysql_real_connect (conn, opt_host_name, opt_user_name, opt_password,
    opt_db_name, opt_port_num, opt_socket_name, opt_flags) == NULL)
{
    print_error (conn, "mysql_real_connect() failed");
    mysql_close (conn);
    exit (1);
}
```

```
/* ... hier Anfragen und Prozessergebnisse absetzen ... */  
  
/* Serververbindung trennen */  
mysql_close (conn);  
exit (0);  
}
```

Verglichen mit den Programmen *client1*, *client2* und *show\_opt*, die wir bereits entwickelt haben, tut *client3* ein paar ganz neue Dinge:

- Das Programm ermöglicht die Auswahl einer Datenbank über die Befehlszeile. Sie müssen lediglich die Datenbank nach den anderen Argumenten angeben. Dies entspricht dem Verhalten von Standardclients in der MySQL-Distribution.
- Ist im Argumentvektor ein Kennwortwert vorhanden, dann fertigt `get_one_copy()` eine Kopie davon an und löscht nachfolgend das Original. Auf diese Weise wird das Zeitfenster verkürzt, in dessen Verlauf ein über die Befehlszeile eingegebenes Kennwort für *ps* oder andere Systemstatusprogramme sichtbar ist. Beachten Sie, dass dieses Zeitfenster nur *minimiert*, nicht aber beseitigt wird; die Angabe von Kennwörtern in der Befehlszeile ist nach wie vor ein Sicherheitsrisiko.
- Wird die Kennwortoption ohne Wert angegeben, dann setzt `get_one_option()` ein Flag, um anzuzeigen, dass das Programm den Benutzer zur Eingabe eines Kennworts auffordern soll. Diese Aufforderung erfolgt unter `main()` mithilfe der Funktion `get_tty-password()`, nachdem alle Optionen verarbeitet worden sind. `get_tty-password()` ist eine Dienstroutine in der Clientbibliothek, die ein Kennwort anfordert, ohne die Eingabe am Bildschirm anzuzeigen. Vielleicht fragen Sie sich, warum wir nicht einfach `getpass()` aufrufen; die einfache Antwort auf diesen Einwand lautet, dass nicht alle Systeme diese Funktion bieten – Windows beispielsweise kennt `getpass()` nicht. `get_tty-password()` ist systemübergreifend portierbar, weil es so konfiguriert werden kann, dass es sich an Systemeigenarten anpasst.

*client3* stellt entsprechend den von Ihnen angegebenen Optionen eine Verbindung mit dem MySQL-Server her. Nun wollen wir, um die Sache nicht zu verkomplizieren, einmal annehmen, dass keine Optionsdatei vorhanden ist. Wenn Sie nun *client3* ohne Argumente aufrufen, dann stellt es eine Verbindung zu *localhost* her und übergibt dem Server Ihren Unix-Anmeldename und kein Kennwort. Rufen Sie *client3* jedoch wie nachfolgend gezeigt auf, dann fordert das Programm zur Eingabe eines Kennworts auf (denn unmittelbar auf `-p` folgt kein Kennwort), verbindet sich mit *ein\_host* und übergibt den Benutzernamen *ein\_benutzer* sowie das eingegebene Kennwort an den Server:

```
% ./client3 -h ein_host -p -u ein_benutzer eine_db
```

*client3* übergibt außerdem den Datenbanknamen *eine\_db* an `mysql_real_connect()`, um diese zur aktuellen Datenbank zu machen. Ist eine Optionsdatei vorhanden, dann wird ihr Inhalt verarbeitet und zur entsprechenden Änderung der Verbindungsparameter verwendet.

Die Änderungen, die wir bislang zur Entwicklung von *client3* vorgenommen haben, ermöglichen uns etwas, das für jeden MySQL-Client unabdingbar ist: die Verbindung mit dem Server unter Verwendung geeigneter Parameter. Der Prozess wird durch das »Clientskelett« *client3.c* implementiert, das Sie als Basis für andere Programme verwenden können. Kopieren Sie den Code, und fügen Sie nach Belieben anwendungsspezifische Details hinzu. Auf diese Weise können Sie sich auf das konzentrieren, was Sie wirklich interessiert: die Möglichkeit, auf den Inhalt Ihrer Datenbanken zuzugreifen. Der wirklich spannende Teil Ihrer Anwendung findet zwischen den Aufrufen `mysql_real_connect()` und `mysql_close()` statt, aber was wir jetzt vor uns haben, dient als wesentlicher Rahmen, den Sie für viele unterschiedliche Clients verwenden können. Um ein neues Programm zu schreiben, gehen Sie wie folgt vor:

1. Erstellen Sie eine Kopie von *client3.c*.
2. Ändern Sie, wenn Sie über die Standardoptionen, die *client3.c* kennt, hinaus Zusatzoptionen verarbeiten wollen, die Schleife zur Verarbeitung von Optionen entsprechend.
3. Fügen Sie Ihren eigenen anwendungsspezifischen Code zwischen den Aufrufen zum Aufbau und zum Abbau der Verbindung ein.

Das war's schon.

## 6.5 Anfragen verarbeiten

Der Sinn der Verbindung mit einem Server besteht darin, mit ihm zu kommunizieren, solange diese Verbindung vorhanden ist. In diesem Abschnitt wollen wir Ihnen zeigen, wie Sie mit dem Server zwecks Verarbeitung von Anfragen kommunizieren. Jede Anfrage, die Sie ausführen, besteht aus den folgenden Schritten:

1. **Anfrage schreiben.** Wie Sie das machen, hängt vom Inhalt der Anfrage ab – insbesondere davon, ob sie binäre Daten enthält.
2. **Anfrage absetzen, indem sie an den Server gesendet wird.** Der Server wird die Anfrage ausführen und ein Ergebnis erzeugen.
3. **Anfrageergebnis verarbeiten.** Das ist davon abhängig, welchen Anfragetyp Sie ausgeführt haben. Beispielsweise gibt eine `SELECT`-Anweisung Datenzeilen zurück, die Sie verarbeiten können. Eine `INSERT`-Anweisung dagegen tut dies nicht.

Beim Konstruieren von Anfragen sollten Sie berücksichtigen, mit welcher Funktion sie zum Server geschickt werden. Die allgemeinere Routine zum Absetzen von Anfragen ist `mysql_real_query()`. Mit dieser Routine übergeben Sie die Anfrage als gezählten String (d.h. als einen String plus seine Länge). Sie müssen die Länge Ihres Anfragestrings verwalten und sie `mysql_real_query()` zusammen mit dem eigentlichen String übergeben. Weil die Anfrage ein gezählter String ist, kann sie beliebige Dinge enthalten, auch binäre Daten oder Nullbytes. Die Anfrage wird nicht als nullterminierter String behandelt.

Die andere Funktion zum Absetzen von Anfragen, `mysql_query()`, ist restriktiver in Hinblick auf den Inhalt des Anfragestrings, aber häufig einfacher in der Anwendung. Anfragen, die Sie `mysql_query()` übergeben, sollten nullterminierte Strings sein, d.h., sie dürfen keine Nullbytes im Anfragetext enthalten. (Falls die Anfragen Nullbytes enthalten, werden sie irrtümlich kürzer interpretiert, als sie eigentlich sind.) Allgemein gesagt, sollten Sie `mysql_query()` nicht verwenden, wenn Ihre Anfrage beliebige binäre Daten enthalten soll, denn dann können auch Nullbytes vorhanden sein. Wenn Sie dagegen mit nullterminierten Strings arbeiten, genießen Sie den Luxus, Ihre Anfragen mithilfe der Stringfunktionen 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 in Abschnitt 6.9.2, »Kodierung problematischer Daten in Anfragen«.

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 zulässig akzeptiert hat und sie ausführen konnte. Das sagt aber noch nichts über das Ergebnis der Anfrage aus. Beispielsweise ist damit nicht garantiert, dass eine `SELECT`-Anfrage Zeilen ermittelt oder 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 die folgenden:

- Sie enthält einen Syntaxfehler.
- Sie ist semantisch fehlerhaft (eine Anfrage könnte beispielsweise auf eine nicht existierende Spalte einer Tabelle verweisen).
- Sie haben nicht die erforderlichen Berechtigungen, um auf eine Tabelle zuzugreifen, die in der Anfrage referenziert wird.

Anfragen lassen sich in zwei allgemeine Kategorien einteilen: in Anfragen, die kein Ergebnis erzeugen, und in Anfragen, 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` hingegen 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 ebenso wie Metadaten zu den Werten (beispielsweise die Spaltennamen und Datenwertlängen) enthält. Eine leere Ergebnismenge (d.h. eine mit null Zeilen) ist etwas anderes als »kein Ergebnis«.

### 6.5.1 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 (conn, "INSERT statement failed");
}
else
{
    printf ("INSERT statement succeeded: %lu rows affected\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 (das funktioniert zwar beispielsweise unter FreeBSD, nicht aber unter Solaris). Das Problem wird durch die Umwandlung des Werts in den Datentyp `unsigned long` und die Verwendung des Ausgabeformats `%lu` gelöst. Dasselbe gilt für alle anderen Funktionen, die `my_ulonglong`-Werte zurückgeben, beispielsweise `mysql_num_rows()` und `mysql_insert_id()`. Wenn Ihre Clientprogramme über verschiedene Systeme hinweg portabel sein sollen, müssen Sie diese Tatsache berücksichtigen.

`mysql_affected_rows()` gibt die Anzahl der von der Anfrage bearbeiteten Zeilen zurück, aber die Bedeutung von *Affected Rows* (also *betroffene Zeilen*) ist vom Anfragetyp abhängig. Bei `INSERT`, `REPLACE` oder `DELETE` handelt es sich dabei um die Anzahl der eingefügten, ersetzten oder gelöschten Zeilen, bei `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 heißt, eine Zeile muss nicht unbedingt

geändert werden, wenn sie (durch die WHERE-Klausel der UPDATE-Anweisung) zur Aktualisierung ausgewählt wird.

*Affected Rows* ist in Zusammenhang mit der UPDATE-Anweisung etwas irreführend, weil manche Anwender meinen, es bedeute *gefundene* Zeilen, also die Anzahl der Zeilen, die zur Aktualisierung ausgewählt wurden, auch wenn diese letztlich ihre Werte nicht geändert haben. Wenn Ihre Anwendung diese Aussage benötigt, fordern Sie sie beim Verbindungsaufbau zum Server an. Übergeben Sie `mysql_real_connect()` mit dem *flags*-Wert `CLIENT_FOUND_ROWS`.

## 6.5.2 Verarbeitung von Anfragen mit einer Ergebnismenge

Anfragen, die Daten zurückgeben, erzeugen eine Ergebnismenge, die Sie nach Ausführung der Anfrage durch einen 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, DESCRIBE, EXPLAIN und CHECK TABLE erzeugen Ergebnisse. Für alle diese Anweisungen ist nach Ausführung der Anfrage eine zusätzliche Verarbeitung der Zeilen erforderlich.

Ergebnismengen werden wie folgt behandelt:

1. **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.
2. **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 Strings, 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 etwas ganz anderes damit tun. Stehen keine weiteren Zeilen in der Ergebnismenge, dann gibt `mysql_fetch_row()` den Wert NULL zurück.
3. **Nachdem Sie die Ergebnismenge verarbeitet haben, rufen Sie `mysql_free_result()` auf, um den davon belegten Speicher freizugeben.** Wenn Sie das versäumen, geht Ihrer Anwendung Speicher verloren. Sie müssen insbesondere bei lange laufenden Anwendungen darauf achten, 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 verarbeitet wird, die eine Ergebnismenge besitzt:

```
MYSQL_RES *res_set;

if (mysql_query (conn, "SHOW TABLES FROM sampdb") != 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() failed");
    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)
{
    /* macht irgendetwas mit dem Zeileninhalt */
}
```

Der `MYSQL_ROW`-Rückgabewert von `mysql_fetch_row()` ist ein Zeiger auf ein Array mit Werten. Wenn der Rückgabewert einer Variablen namens `row` zugewiesen wird, dann kann jeder Wert innerhalb der Zeile als `row[i]` referenziert werden, wobei `i` einen Wertebereich zwischen 0 und *Anzahl der Zeilenspalten* -1 hat. 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`.
- Werte für alle Datentypen (einschließlich der numerischen) werden als Strings zurückgegeben. Wenn Sie einen Wert als Zahl verarbeiten wollen, müssen Sie ihn selbst umwandeln.
- Die Strings in `MYSQL_ROW` sind nullterminiert. Wenn eine Spalte jedoch binäre Daten enthalten darf, darf sie auch Nullbytes enthalten. Sie sollten den Wert also nicht als nullterminierten String behandeln. Ermitteln Sie die Spaltenlänge, um die Länge des Spaltenwerts festzustellen (wie Spaltenlängen ermittelt werden, ist in Abschnitt 6.5.6, »Metadaten in Ergebnismengen«, beschrieben).

- **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; andernfalls kann sich Ihr Programm beim Versuch aufhängen, einen **NULL-Zeiger** zu dereferenzieren.

Was Sie mit den einzelnen Zeilen tun, hängt natürlich vom Zweck Ihrer Anwendung ab. Wir wollen die Zeilen zum besseren Verständnis ausgeben und dabei die Spaltenwerte durch Tabulatorzeichen voneinander abtrennen. Dazu brauchen wir zusätzlich die Funktion `mysql_num_fields()` aus der Clientbibliothek, denn sie teilt uns mit, wie viele Spalten eine 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() failed");
    else
        printf ("%lu rows returned\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 als *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. Beachten Sie aber, dass `mysql_num_rows()` anders als `mysql_affected_rows()` nicht den Verbindungs-Handle, sondern einen Ergebnismengenzeiger als Argument verwendet.

Auf die Schleife zum Ermitteln der Zeilen folgt eine Fehlerprüfung (dies ist eine reine Vorsichtsmaßnahme). Wenn Sie mit `mysql_store_result()` eine Ergebnismenge erzeugen, bedeutet der `mysql_fetch_row()`-Rückgabewert `NULL` immer »keine weiteren Zeilen«. Erzeugen Sie die Ergebnismenge dagegen mit `mysql_use_result()`, dann 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 zur Ausgabe von Spaltenwerten, der auch gewisse Unzulänglichkeiten aufweist. Führen Sie beispielsweise einmal die folgende Anfrage aus:

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

Damit erhalten Sie die folgende Ausgabe, die nicht besonders einfach zu lesen ist:

```
Adams   John   Braintree  MA
Adams   John Quincy Braintree  MA
Arthur  Chester A.  Fairfield  VT
Buchanan James  Mercersburg PA
Bush    George H.W. Milton    MA
Bush    George W.   New Haven  CT
Carter  James E.    Plains    GA
```

...

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 ist verfügbar, aber nicht als Teil der Spaltendatenwerte; sie ist vielmehr in den Metadaten (Daten über die Daten) der Ergebnismenge festgehalten. Nachdem wir unsere Anfrageverarbeitung noch ein wenig verallgemeinert haben, werden wir sie – in Abschnitt 6.5.6, »Metadaten in Ergebnismengen« – auch ansprechender formatieren.

### Binärdaten ausgeben

Spaltenwerte, die Binärdaten enthalten, können problematisch sein, weil diese Binärdaten eben auch Nullbytes sein können, die mithilfe der Format-spezifizierung `%s printf()` nicht korrekt ausgegeben werden können. (Das liegt daran, dass `printf()` einen nullterminierten String erwartet und den Spaltenwert in diesem Fall nur bis zum ersten Nullbyte ausgeben wird.) Es ist deswegen am besten, die Spaltenlänge zu benutzen, sodass Sie den vollständigen Wert ausgeben können. Beispielsweise könnten Sie `fwrite()` verwenden.

## 6.5.3 Eine allgemeine Anfrageverarbeitung

Bei den oben beschriebenen Beispielen zur Anfrageverarbeitung wusste man im Voraus, ob die Anweisungen Ergebnismengen erzeugen. Dies war möglich, weil die Anfragen im Code festgeschrieben waren: 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 und ob die Anfrage überhaupt zulässig ist. 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:

```
/* kommentar */ SELECT ...
```

Glücklicherweise müssen Sie den Typ der Anfrage nicht im Voraus kennen, um sie korrekt verarbeiten zu können. Das C-API von MySQL ermöglicht die Entwicklung einer allgemeinen Anfrageverarbeitung, 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 erläutern:

1. Setzen Sie die Anfrage ab. Schlägt sie fehl, so sind wir fertig.
2. Ist die Anfrage erfolgreich, rufen wir `mysql_store_result()` auf, um die Zeilen vom Server zu ermitteln und eine Ergebnismenge zu erzeugen.
3. Ist `mysql_store_result()` erfolgreich, dann gibt die Anfrage eine Ergebnismenge zurück. Verarbeiten Sie die Zeilen, indem Sie `mysql_fetch_row()` aufrufen, bis diese Routine `NULL` zurückgibt. Geben Sie dann den Speicher der Ergebnismenge wieder frei.
4. Schlägt `mysql_store_result()` fehl, dann 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:
  - Gibt `mysql_field_count()` den Wert `0` zurück, dann bedeutet dies, dass die Anfrage keine Spalten zurückgegeben hat, d.h., es liegt auch keine Ergebnismenge vor (dies weist darauf hin, dass die Anfrage eine Anweisung wie `INSERT`, `DELETE` oder `UPDATE` war).
  - Ist `mysql_field_count()` ungleich Null, dann ist ein Fehler aufgetreten. Die Anfrage hätte eine Ergebnismenge zurückgeben sollen, hat dies aber nicht getan. Das kann unterschiedliche Ursachen haben. Beispielsweise könnte es sein, dass die Ergebnismenge zu groß war und nicht genügend Speicher zur Verfügung stand oder dass während des Ladens der Zeilen die Netzwerkverbindung zwischen dem Client und dem Server ausgefallen ist.

Das folgende Listing zeigt eine Funktion, die beliebige Anfragen verarbeitet, sofern ein Verbindungs-Handle und ein nullterminierter Anfragestring vorhanden sind:

```
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, "Could not execute query");
        return;
    }

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

    res_set = mysql_store_result (conn);
    if (res_set) /* Ergebnismenge zurückgegeben */
    {
        /* Zeilen verarbeiten und Ergebnismenge freigeben */
        process_result_set (conn, res_set);
        mysql_free_result (res_set);
    }
    else /* 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)
        {
            /*
            * Anfrage hat keine Ergebnismenge zurückgegeben
            * (kein SELECT, SHOW, DESCRIBE, etc.); nur die Anzahl
            * der betroffenen Zeilen ausgeben
            */
            printf ("%lu rows affected\n",
                    (unsigned long) mysql_affected_rows (conn));
        }
        else /* Ein Fehler ist aufgetreten */
        {
            print_error (conn, " Could not retrieve result set");
        }
    }
}
```

Erschwert wird diese Prozedur dadurch, dass es `mysql_field_count()` erst seit MySQL 3.22.24 gibt. In früheren Versionen verwenden Sie stattdessen `mysql_num_fields()`. Um Programme zu entwickeln, die mit allen MySQL-Versionen funktionieren, nehmen Sie den folgenden Codeabschnitt in den Quellcode auf, nachdem Sie `mysql.h` eingebunden haben und bevor Sie `mysql_field_count()` aufrufen:

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

#define konvertiert Aufrufe von mysql_field_count() für MySQL-Versionen vor 3.22.24 in Aufrufe von mysql_num_fields().
```

## 6.5.4 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 abzurufen.
- Wurde keine Ergebnismenge zurückgegeben, dann ermittelt sie mit `mysql_field_count()`, ob ein Fehler aufgetreten oder keine Ergebnismenge zu erwarten ist.

Für alle drei Aspekte der Anfrageverarbeitung gibt es alternative Ansätze:

- Sie könnten statt eines nullterminierten Anfragestrings und `mysql_query()` auch einen gezählten Anfragestring 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.

Als Beispiel finden Sie hier die Funktion `process_real_query()`, die dieselben Aufgaben 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 fehlgeschlagen */
    {
        print_error (conn, "Could not execute query");
        return;
    }

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

    res_set = mysql_use_result (conn);
    if (res_set) /* Ergebnismenge zurückgegeben */
    {
        /* Zeilen verarbeiten und Ergebnismenge freigeben */
```

```
        process_result_set (conn, res_set);
        mysql_free_result (res_set);
    }
    else /* keine Ergebnismenge zurückgegeben */
    {
        /*
         * Bedeutet das Fehlen einer Ergebnismenge, dass ein Fehler
         * aufgetreten ist oder dass es keine Ergebnismenge gibt?
         */
        if (mysql_errno (conn) == 0)
        {
            /*
             * Anfrage hat keine Ergebnismenge zurückgegeben
             * (kein SELECT, SHOW, DESCRIBE, etc.); nur die Anzahl
             * der betroffenen Zeilen ausgeben
             */
            printf ("%lu rows affected\n",
                    (unsigned long) mysql_affected_rows (conn));
        }
        else /* Ein Fehler ist aufgetreten */
        {
            print_error (conn, "Could not retrieve result set");
        }
    }
}
```

### 6.5.5 Ein Vergleich von `mysql_store_result()` und `mysql_use_result()`

Die Funktionen `mysql_store_result()` und `mysql_use_result()` sind sich auf den ersten Blick ganz ähnlich: Beide nehmen einen Verbindungs-Handle als Argument entgegen und geben eine Ergebnismenge zurück. Es gibt jedoch entscheidende Unterschiede. Der wesentliche Unterschied besteht darin, 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. Diese unterschiedlichen Ansätze führen zu allen möglichen Unterschieden zwischen den beiden Funktionen. In diesem Abschnitt sollen die beiden Funktionen verglichen werden, sodass Sie besser entscheiden können, welche Funktion 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()` geben nie einen Fehler zurück, 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.

Im Gegensatz dazu ermittelt `mysql_use_result()` keine Zeilen, sondern initiiert 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 durch `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 zwischen diesen beiden Ergebnissen durch den Aufruf von `mysql_errno()` oder `mysql_error()`.

`mysql_store_result()` stellt höhere Anforderungen an den Speicher und die 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 stattdessen `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, sie alle zu verarbeiten. Damit ist `mysql_use_result()` für bestimmte Clienttypen ungeeignet:

- für interaktive Clients, die auf Anforderung des Benutzers zeilenweise fortschreiten (Sie wollen schließlich nicht, dass der Server warten muss, bis er die nächste Zeile senden kann, nur weil der Benutzer gerade eine Kaffeepause macht.)
- für 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, sodass Sie wahlfrei darauf zugreifen können: Mit den Funktionen `mysql_data_seek()`, `mysql_row_seek()` und `mysql_row_tell()` können Sie in beliebiger Reihenfolge auf die Zeilen zugreifen. 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 in der Reihenfolge verarbeiten wollen, in der sie vom Server zurückgegeben werden, müssen Sie `mysql_store_result()` verwenden. Wenn Sie beispielsweise eine Anwendung haben, mit der sich der Benutzer vorwärts und rückwärts durch die Zeilen bewegen kann, die eine Anfrage ausgewählt hat, dann 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_width` 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 (Sie können dies verhindern, indem Sie vor dem Absetzen der zweiten Anfrage die Funktion `mysql_free_result()` aufrufen, denn diese holt alle wartenden Zeilen für Sie ab und verwirft sie.) Das heißt aber auch, dass Sie bei diesem Verarbeitungsmodell mit `mysql_use_result()` immer nur mit einer einzigen Ergebnismenge arbeiten können.

Synchronisierungsfehler treten bei `mysql_store_result()` nicht auf, 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 `mytbl` gibt, könnten Sie die folgende Anfrage ausführen:

```
SHOW TABLES LIKE 'mytbl'
```

Falls der Wert von `mysql_num_rows()` nach dem Aufruf von `mysql_store_result()` ungleich Null ist, gibt es die Tabelle. `mysql_fetch_row()` muss nicht aufgerufen werden.

Ergebnismengen, die mit `mysql_store_result()` erzeugt werden, sollten mit `mysql_free_result()` freigegeben werden, aber das muss nicht unbedingt sein, bevor Sie eine andere Anfrage absetzen. Das wiederum bedeutet, dass Sie – im Gegensatz zur Verwendung von `mysql_use_result()`, wo Sie immer nur ein Ergebnis auf einmal verwenden können – mehrere Ergebnismengen erstellen und mit diesen gleichzeitig arbeiten können.

Wenn Sie eine maximale Flexibilität bieten wollen, müssen Sie es den Benutzern ermöglichen, die Verarbeitungsmethode für die Ergebnismenge selbst auszuwählen. 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()`.

### 6.5.6 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_lengths()` ermittelt werden.
- Informationen über jede Spalte, beispielsweise ihren Namen und ihren Typ, die maximale Breite aller Spaltenwerte sowie die Tabelle, aus der die Spalte stammt. Diese Information wird in `MYSQL_FIELD`-Strukturen gespeichert, die normalerweise mit `mysql_fetch_field()` ermittelt werden. In Anhang F, »C-API-Referenz«, ist die Struktur `MYSQL_FIELD` beschrieben und sind alle Funktionen aufgelistet, 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()` und 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 wohlformatierte Ausgabe mit Spaltentiteln und vertikaler Ausrichtung.
- Mithilfe des Spaltenzählers ermitteln Sie, wie oft Sie eine Schleife zur Verarbeitung aufeinander folgender Spaltenwerte aus Datenzeilen durchlaufen müssen.
- Sie können anhand der Zeilen- oder Spaltenzähler Datenstrukturen reservieren, für die Sie die Anzahl der Zeilen oder Spalten in der Ergebnismenge kennen müssen.
- Sie können den Datentyp einer Spalte ermitteln. Auf diese Weise können Sie erkennen, ob eine Spalte eine Zahl, binäre Daten etc. repräsentiert.

In Abschnitt 6.5.2, »Verarbeitung von Anfragen mit einer 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 – etwa wenn Sie die Daten beispielsweise in eine Tabellenkalkulation importieren wollen –, aber für Ausdrücke 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 H.W. Milton MA
Bush George W. New Haven CT
Carter James E. Plains GA
...
```

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 H.W. | Milton | MA |
| Bush | George W. | New Haven | CT |
| Carter | James E. | Plains | GA |
...
+-----+-----+-----+-----+

```

Der Anzeigalgorithmus geht wie folgt vor:

1. Er ermittelt die Anzeigebreite aller Spalten.
2. Er gibt eine Zeile mit eingerahmten Spaltenüberschriften (d.h. durch vertikale Striche voneinander abgetrennt und zwischen Zeilen mit Trennstrichen eingerahmt) aus.
3. Er gibt die Werte jeder Zeile der Ergebnismenge in den eingerahmten Spalten (abgetrennt durch vertikale Striche) vertikal ausgerichtet aus. Darüber hinaus werden Zahlen rechtsbündig ausgerichtet, und für NULL-Werte wird das Wort NULL ausgegeben.
4. Am Ende wird die 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 identisch, 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 die einzelnen Spaltenwerte dazu nicht herangezogen werden:

```

MYSQL_FIELD *field;
unsigned long col_len;
unsigned int i;

/* Spaltenanzeigebreite ermitteln -- erfordert die Erzeugung */
/* der Ergebnismenge mit mysql_store_result(), nicht mit */
/* mysql_use_result() */

```

```
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 Wortes "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 Höchstwert dreier Werte, die sich jeweils aus den Metadaten der Struktur mit den Spalteninformationen ergeben:

- `field->name` (Länge des Spaltentitels)
- `field->max_length` (Länge des längsten Datenwerts in der Spalte)
- Länge des Strings `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, das ein Element einer Struktur ist, die wir aus der Clientbibliothek erhalten. Ist das erlaubt, oder sollte der Inhalt der `MYSQL_FIELD`-Struktur schreibgeschützt sein? Normalerweise würde ich »schreibgeschützt« antworten, aber einige der Clientprogramme in der MySQL-Distribution ändern den Wert von `max_length` auf ähnliche Weise. Ich nehme also an, dass das so in Ordnung ist (wenn Sie einen alternativen Ansatz vorziehen, bei dem `max_length` nicht geändert wird, weisen Sie einem Array `unsigned int`-Werte zu und legen die berechneten Breiten in diesem Array ab).

Bei der Berechnung der Anzeigebreiten müssen Sie auf folgenden kritischen Punkt achten: 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.<sup>2</sup>

Wenn 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 |", (int) field->max_length, field->name);
```

2. 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.

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 rechtsbündig ausgerichtet werden). Spaltenwerte werden wie folgt ausgegeben, wobei `row[i]` den Datenwert enthält und `field` auf die Spalteninformation zeigt:

```
if (row[i] == NULL)           /* Wort "NULL" drucken */
    printf (" %-*s |", (int) field->max_length, "NULL");
else if (IS_NUM (field->type)) /* Wert rechtsbündig drucken */
    printf ("%*s |", (int) field->max_length, row[i]);
else                          /* Wert linksbündig drucken */
    printf (" %-*s |", (int) 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 namens `print_dashes()` eingekapselt haben:

```
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_ROW    row;
    MYSQL_FIELD    *field;
    unsigned long col_len;
    unsigned int    i;

    /* Spaltenanzeigebreite ermitteln -- erfordert die Erzeugung */
    /* der Ergebnismenge mit mysql_store_result(), nicht mit */
    /* mysql_use_result() */
```

```

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 Wortes "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 |", (int) 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) /* Wort "NULL" drucken */
            printf (" %-*s |", (int) field->max_length, "NULL");
        else if (IS_NUM (field->type)) /* Wert rechtsbündig drucken */
            printf (" %*s |", (int) field->max_length, row[i]);
        else /* Wert linksbündig drucken */
            printf (" %-*s |", (int) field->max_length, row[i]);
    }
    fputc ('\n', stdout);
}
print_dashes (res_set);
printf ("%lu rows returned\n", (unsigned long) mysql_num_rows (res_set));
}

```

Die MySQL-Clientbibliothek 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 aus `mysql_field_seek()` und `mysql_fetch_field()` stellt jedoch nur eine Methode dar, 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, »C-API-Referenz«.

## 6.6 Client 4: Ein interaktives Anfrageprogramm

Jetzt wollen wir einmal zusammentragen, was wir bisher entwickelt haben, und damit einen einfachen interaktiven Client schreiben: *client4*. Das Programm ermöglicht es Ihnen, Anfragen einzugeben, führt sie mithilfe unserer allgemeinen Anfrageverarbeitung `process_query()` aus und zeigt das Ergebnis mit der im vorigen Abschnitt entwickelten Anzeigeformatierung `process_result_set()` an.

*client4* ist *mysql* in mancher Hinsicht ganz ähnlich, bietet aber natürlich nicht so viele Funktionen. Für die Eingabe in *client4* gibt es einige Einschränkungen:

- Jede Eingabezeile muss eine einzige, vollständige Anfrage enthalten.
- Anfragen sollten nicht durch ein Semikolon oder mit `\g` abgeschlossen werden.
- Die einzigen Nicht-SQL-Befehle, die erkannt werden, sind `quit` und `\q` zum Beenden des Programms. Alternativ können Sie auch die Tastenkombination `[Strg] [D]` verwenden.

Es stellt sich heraus, dass *client4* (mit gerade einem Dutzend neuer Codezeilen) ganz einfach zu realisieren ist. Fast alles, was wir brauchen, ist bereits in unserem Clientprogrammgerüst (*client3.c*) enthalten oder wird durch den anderen bisher entwickelten Code bereitgestellt. Wir brauchen nur noch eine Schleife, die Eingabezeilen entgegennimmt und ausführt.

Sie entwickeln *client4*, indem Sie zunächst das Clientgerüst von *client3.c* in *client4.c* kopieren. Anschließend fügen Sie den Code für `process_query()`, `process_result_set()` und `print_dashes()` ein. Suchen Sie in *client4.c* im Abschnitt `main()` nach der folgenden Zeile:

```

    /* ... hier Anfragen und Prozessergebnisse absetzen ... */

```

Ersetzen Sie sie durch die folgende `while`-Schleife:

```

while (1)
{
    char    buf[10000];

    fprintf (stderr, "query> ");           /* Eingabeaufforderung
    ↵                                       ausgeben */
    if (fgets (buf, sizeof (buf), stdin) == NULL) /* Anfrage lesen */

```

```

        break;
    if (strcmp (buf, "quit\n") == 0 || strcmp (buf, "\\q\n") == 0)
        break;
    process_query (conn, buf);           /* Anfrage ausführen */
}

```

Kompilieren Sie *client4.c*, um *client4.o* zu erzeugen, und linken Sie dann *client4.o* mit *common.o* und der Clientbibliothek, um *client4* zu erzeugen – das war’s! Sie haben ein interaktives MySQL-Clientprogramm, das beliebige Anfragen ausführt und das Ergebnis anzeigt. Das folgende Beispiel zeigt, wie das Programm sowohl für *SELECT*- als auch für andere Anfragetypen sowie für fehlerhafte Anweisungen funktioniert:

```

% ./client4
query> USE sampdb
0 rows affected
query> SELECT DATABASE(), USER()
+-----+-----+
| DATABASE() | USER() |
+-----+-----+
| sampdb     | sampadm@localhost |
+-----+-----+
1 rows returned
query> SELECT COUNT(*) FROM president
+-----+
| COUNT(*) |
+-----+
|      42 |
+-----+
1 rows returned
query> SELECT last_name, first_name FROM president ORDER BY last_name LIMIT 3
+-----+-----+
| last_name | first_name |
+-----+-----+
| Adams    | John      |
| Adams    | John Quincy |
| Arthur   | Chester A. |
+-----+-----+
3 rows returned
query> CREATE TABLE t (i INT)
0 rows affected
query> SELECT j FROM t
Could not execute query
Error 1054 (Unknown column 'j' in 'field list')
query> USE mysql
Could not execute query
Error 1044 (Access denied for user: 'sampadm@localhost' to database 'mysql')

```

## 6.7 Clients mit SSL-Support entwickeln

Mit Version 4 wurde die SSL-Unterstützung in MySQL implementiert, d.h., Sie können nun von Ihren eigenen Programmen aus über sichere Verbindungen auf den Server zugreifen. Um Ihnen zu zeigen, wie das geht, beschreiben wir in diesem Abschnitt den Vorgang zur Modifikation von *client4*, an dessen Ende ein ähnlicher Client namens *sslclient* steht, der nach außen hin mit *client4* weitgehend identisch ist, aber die Herstellung verschlüsselter Verbindungen unterstützt. Damit *sslclient* korrekt funktioniert, muss MySQL mit SSL-Unterstützung kompiliert worden sein. Ferner muss der Server mit den passenden Optionen gestartet werden, die sein Zertifikat und die Schlüsseldateien benennen. Außerdem benötigen Sie auch auf der Clientseite Zertifikate und Schlüsseldateien. Weitere Informationen finden Sie in Kapitel 12.3, »Sichere Verbindungen einrichten«. Außerdem sollten Sie MySQL 4.0.5 oder höher verwenden. Die SSL-Routinen sowie die Routinen zur Optionsverarbeitung verhalten sich bei MySQL-Versionen vor 4.0.x anders als hier beschrieben.

Die *sampdb*-Distribution enthält eine Quellcodedatei namens *sslclient.c*, aus der das Clientprogramm *sslclient* kompiliert werden kann. Der folgende Vorgang beschreibt ausgehend von *client4.c*, wie *sslclient.c* erstellt wird.

1. Kopieren Sie *client4.c* als *sslclient.c*. Die verbleibenden Schritte beziehen sich alle auf *sslclient.c*.
2. Damit der Compiler erkennen kann, ob die SSL-Unterstützung vorhanden ist, definiert die MySQL-Header-Datei *my\_config.h* das Symbol `HAVE_OPENSSL` entsprechend. Das heißt: Wenn Sie Code mit SSL-Bezug schreiben, müssen Sie das folgend. Konstrukt verwenden, damit dieser Code ignoriert wird, wenn SSL nicht zur Verfügung steht:

```
#ifdef HAVE_OPENSSL
    ...SSL-Code...
#endif
```

*my\_config.h* ist in *my\_global.h* enthalten. Letzteres ist bereits in *sslclient.c* eingebunden, d.h., Sie müssen *my\_config.h* nicht explizit einbinden.

3. Erweitern Sie das Array `my_opts`, das Optionsdatenstrukturen enthält, sodass auch standardmäßige SSL-Optionen (`--ssl-ca`, `--ssl-key` usw.) berücksichtigt werden. Die einfachste Möglichkeit, dies zu tun, besteht darin, den Inhalt der Datei *ssloptlongopts.h* mithilfe einer `#include`-Anweisung in das Array `my_opts` einzubinden. Danach sieht `my_opts` wie folgt aus:

```
static struct my_option my_opts[] = /* Optionsdatenstrukturen */
{
    {"help", '?', "Display this help and exit",
     NULL, NULL, NULL,
     GET_NO_ARG, NO_ARG, 0, 0, 0, 0, 0, 0},
    {"host", 'h', "Host to connect to",
     (gptr *) &opt_host_name, NULL, NULL,
     GET_STR_ALLOC, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
```

```

{"password", 'p', "Password",
(gptr *) &opt_password, NULL, NULL,
GET_STR_ALLOC, OPT_ARG, 0, 0, 0, 0, 0, 0},
{"port", 'P', "Port number",
(gptr *) &opt_port_num, NULL, NULL,
GET_UINT, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
{"socket", 'S', "Socket path",
(gptr *) &opt_socket_name, NULL, NULL,
GET_STR_ALLOC, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
{"user", 'u', "User name",
(gptr *) &opt_user_name, NULL, NULL,
GET_STR_ALLOC, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},

#include <sslopt-longopts.h>

{ NULL, 0, NULL, NULL, NULL, NULL, GET_NO_ARG, NO_ARG, 0, 0, 0, 0, 0, 0 }
};

```

*sslopt-longopts.h* ist eine öffentliche MySQL-Header-Datei. Der Inhalt (mit leicht korrigiertem Format) sieht wie folgt aus:

```

#ifdef HAVE_OPENSSL
{"ssl", OPT_SSL_SSL,
"Enable SSL for connection. Disable with --skip-ssl",
(gptr*) &opt_use_ssl, NULL, 0,
GET_BOOL, NO_ARG, 0, 0, 0, 0, 0, 0},
{"ssl-key", OPT_SSL_KEY, "X509 key in PEM format (implies --ssl)",
(gptr*) &opt_ssl_key, NULL, 0,
GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
{"ssl-cert", OPT_SSL_CERT, "X509 cert in PEM format (implies --ssl)",
(gptr*) &opt_ssl_cert, NULL, 0,
GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
{"ssl-ca", OPT_SSL_CA,
"CA file in PEM format (check OpenSSL docs, implies --ssl)",
(gptr*) &opt_ssl_ca, NULL, 0,
GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
{"ssl-capath", OPT_SSL_CAPATH,
"CA directory (check OpenSSL docs, implies --ssl)",
(gptr*) &opt_ssl_capath, NULL, 0,
GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
{"ssl-cipher", OPT_SSL_CIPHER, "SSL cipher to use (implies --ssl)",
(gptr*) &opt_ssl_cipher, NULL, 0,
GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
#endif /* HAVE_OPENSSL */

```

- Die in *sslopt-longopts.h* definierten Optionsstrukturen beziehen sich auf die Werte `OPT_SSL_SSL`, `OPT_SSL_KEY` usw. Diese werden als kurze Optionscodes verwendet und müssen in Ihrem Programm definiert sein; Sie geben beispielsweise die folgenden Zeilen vor der Definition des Arrays `my_opts` ein:

```

#ifdef HAVE_OPENSSL
enum options
{
    OPT_SSL_SSL=256,
    OPT_SSL_KEY,
    OPT_SSL_CERT,
    OPT_SSL_CA,
    OPT_SSL_CAPATH,
    OPT_SSL_CIPHER
};
#endif

```

Wenn Sie eigene Anwendungen schreiben, müssen Sie sich, wenn ein gegebenes Programm auch Codes für andere Optionen definiert, vergewissern, dass die Symbole vom Typ `OPT_SSL_XXX` andere Werte haben als in diesen anderen Codes.

- Die SSL-bezogenen Optionsstrukturen in `sslopt-longopts.h` referenzieren Variablen, die zur Aufnahme der Optionswerte verwendet werden. Um diese Variablen zu deklarieren, binden Sie mit einer `#include`-Anweisung den Inhalt von `sslopt-vars.h` in Ihr Programm ein. Die Einbindung erfolgt vor der Definition des Arrays `my_opts`. `sslopt-vars.h` und sieht wie folgt aus:

```

#ifdef HAVE_OPENSSL
static my_bool opt_use_ssl = 0;
static char *opt_ssl_key = 0;
static char *opt_ssl_cert = 0;
static char *opt_ssl_ca = 0;
static char *opt_ssl_capath = 0;
static char *opt_ssl_cipher = 0;
#endif

```

- In der Routine `get_one_option()` müssen Sie eine Zeile hinzufügen, um die Datei `sslopt-case.h` einzubinden:

```

my_bool
get_one_option (int optid, const struct my_option *opt, char *argument)
{
    switch (optid)
    {
        case '?':
            my_print_help (my_opts); /* Hilfmeldung drucken */
            exit (0);
        case 'p': /* Kennwort */
            if (!argument) /* Kein Wert angegeben, deshalb
                ↪          später anfordern */
                ask_password = 1;
            ↪          else /* Kennwort kopieren, Original
                ↪          löschen */
                {
                    opt_password = strdup (argument);
                    if (opt_password == NULL)

```

```

        {
            print_error (NULL, "could not allocate password buffer");
            exit (1);
        }
        while (*argument)
            *argument++ = 'x';
    }
    break;
#include <sslopt-case.h>
}
return (0);
}

```

*sslopt-case.h* enthält Fälle für die `switch()`-Anweisung, die erkennen, wenn SSL-Optionen angegeben wurden und ggf. die Variable `opt_use_ssl` setzen. Die Datei sieht wie folgt aus:

```

#ifdef HAVE_OPENSSL
    case OPT_SSL_KEY:
    case OPT_SSL_CERT:
    case OPT_SSL_CA:
    case OPT_SSL_CAPATH:
    case OPT_SSL_CIPHER:
    /*
        SSL aktivieren, sofern ssl-Optionen verwendet werden.
        Kann später mit --skip-ssl oder --ssl=0 deaktiviert werden
    */
    opt_use_ssl= 1;
    break;
#endif

```

Die Folge ist, dass man, wenn die Optionsverarbeitung abgeschlossen ist, bestimmen kann, ob der Benutzer eine sichere Verbindung wünscht, indem man den Wert von `opt_use_ssl` überprüft.

Wenn Sie sich an diese Vorgehensweise halten, werden die normalen Routinen `load_defaults()` und `handle_options()` die SSL-bezogenen Optionen analysieren und die Werte automatisch entsprechend einstellen. Jetzt müssen Sie nur noch, bevor Sie die Serververbindung herstellen, die SSL-Optionsdaten an die Clientbibliothek übergeben, sofern diese Optionen die Verwendung einer SSL-Verbindung spezifizieren. Sie tun dies, indem Sie nach `mysql_init()` – aber vor `mysql_real_connect` – die Funktion `mysql_ssl_set()` aufrufen. Das sieht dann so aus:

```

/* Verbindungs-Handle initialisieren */
conn = mysql_init (NULL);
if (conn == NULL)
{
    print_error (NULL, "mysql_init() failed (probably out of memory)");
    exit (1);
}

```

```
#ifndef HAVE_OPENSSL
/* SSL-Informationen an Clientbibliothek übergeben */
if (opt_use_ssl)
    ↪ mysql_ssl_set (conn, opt_ssl_key, opt_ssl_cert, opt_ssl_ca,
                  opt_ssl_capath, opt_ssl_cipher);
#endif

/* Serververbindung aufbauen */
if (mysql_real_connect (conn, opt_host_name, opt_user_name, opt_password,
    ↪ opt_db_name, opt_port_num, opt_socket_name, opt_flags) == NULL)
{
    print_error (conn, "mysql_real_connect() failed");
    mysql_close (conn);
    exit (1);
}
```

Beachten Sie, dass `mysql_ssl_set()` nicht darauf geprüft wird, ob ein Fehler zurückgegeben wird. Alle Probleme, die in Zusammenhang mit den von Ihnen an diese Funktion übergebenen Informationen entstehen, führen zu einem Fehler, wenn `mysql_real_connect()` aufgerufen wird.

Erzeugen Sie nun durch Kompilierung von `sslclient.c` das Programm `sslclient`, und führen Sie es aus. Wenn der `mysql_real_connect()`-Aufruf erfolgreich ist, können Sie Anfragen absetzen. Wenn Sie `sslclient` mit den entsprechenden SSL-Optionen aufrufen, dann sollte die Kommunikation mit dem Server über eine verschlüsselte Verbindung erfolgen. Um festzustellen, ob dies der Fall ist, setzen Sie die folgende Anfrage ab:

```
SHOW STATUS LIKE 'ssl_cipher'
```

Der Wert von `ssl_cipher` ist nicht leer, wenn eine Verschlüsselung eingesetzt wird (um Ihnen die Sache zu erleichtern, setzt das in der `sampdb`-Distribution enthaltene Programm `sslclient` diese Anfrage automatisch ab und zeigt Ihnen das Ergebnis an).

## 6.8 Verwendung des Embedded Server

Mit MySQL 4 wurde auch die Embedded Server-Bibliothek `libmysqld` eingeführt, die den Server in einer Form enthält, die in Anwendungen eingebunden werden kann. Auf diese Weise können Sie MySQL-basierte Anwendungen erstellen, die – anders als Anwendungen, die als Clients über ein Netzwerk eine Verbindung zu einem separaten Serverprogramm herstellen – autark sind.

Um eine Anwendung mit dem Embedded Server schreiben zu können, müssen zwei Anforderungen erfüllt sein. Zunächst einmal muss die Embedded Server-Bibliothek installiert werden:

- Wenn Sie die Bibliothek aus dem Quelltext erstellen, dann aktivieren Sie sie mithilfe der Option `--with-embedded-server`, die bei der Ausführung von `configure` angegeben werden muss.

- Bei Binärdistributionen müssen Sie eine Max-Distribution verwenden, wenn die Nicht-Max-Distribution *libmysqld* nicht enthalten sollte.
- Bei RPM-Installationen müssen Sie sicherstellen, dass der Embedded Server als RPM installiert wird.

Zum Zweiten müssen Sie ein wenig Code in Ihre Anwendung integrieren, um den Server zu starten bzw. zu beenden.

Wenn Sie sich vergewissert haben, dass beide Anforderungen erfüllt sind, müssen Sie die Anwendung nur noch kompilieren und statt in die normale Clientbibliothek in die Embedded-Server-Bibliothek linken (d.h. `-lmysqld` statt `-lmysqlclient`). Die Struktur der Serverbibliothek ist in der Tat dergestalt, dass Sie (wenn Sie eine Anwendung schreiben, die die Bibliothek verwendet) diese Anwendung problemlos als eingebettete oder Client/Server-Version erstellen können, indem Sie einfach nur die richtige Bibliothek verlinken. Dies funktioniert, weil die reguläre Clientbibliothek Schnittstellenfunktionen enthält, die die gleiche Aufrufsequenz aufweisen wie die Aufrufe für Embedded Server. In der Clientbibliothek sind diese Aufrufe aber Dummys, die nichts machen.

## 6.8.1 Anwendungen mit der Embedded-Server-Bibliothek schreiben

Das Schreiben einer Anwendung, die den Embedded Server verwendet, unterscheidet sich nur wenig von Entwickeln einer Anwendung im Client/Server-Kontext. Tatsächlich können Sie ein Programm, das ursprünglich als Client/Server-Anwendung geschrieben wurde, leicht so umgestalten, dass Sie stattdessen den Embedded Server verwenden. Um beispielsweise *client4* so zu ändern, dass daraus eine eingebettete Anwendung namens *embapp* wird, kopieren Sie *client4.c* in *embapp.c* und führen dann die folgenden Schritte durch, um *embapp.c* entsprechend anzupassen:

1. Fügen Sie *mysql\_embed.h* zu der Menge der vom Programm verwendeten MySQL-Header-Dateien hinzu:

```
#include <my_global.h>
#include <mysql.h>
#include <mysql_embed.h>
#include <my_getopt.h>
```

2. Eine eingebettete Anwendung enthält sowohl eine Client- als auch eine Serverseite, d.h., sie kann eine Gruppe von Optionen für den Client und eine andere Gruppe für den Server verarbeiten. Die Anwendung *embapp* könnte etwa die Gruppen `[client]` und `[embapp]` aus den Optionsdateien des Clientteils auslesen. Um dies zu konfigurieren, ändern Sie die Definitionen des Arrays `client_groups` wie folgt ab:

```
static const char *client_groups[] =
{
    "client", "embapp", NULL
};
```

Die Optionen in diesen Gruppen können von `load_defaults()` und `handle_options()` wie gewöhnlich verarbeitet werden. Dann definieren Sie eine andere Liste mit Optionsgruppen zur Verwendung durch die Serverseite. Per Konvention sollte diese Liste die Gruppen `[server]` und `[embedded]` sowie eine Gruppe `[anw_name_SERVER]` enthalten, wobei `anw_name` der Name Ihrer Anwendung ist. Beim Programm *embapp* heißt die anwendungsspezifische Gruppe also `[embapp_SERVER]`; die Liste der Gruppennamen wird demnach wie folgt deklariert:

```
static const char *server_groups[] =
{
    "server", "embedded", "embapp_SERVER", NULL
};
```

3. Rufen Sie `mysql_server_init()` auf, bevor Sie die Kommunikation mit dem Server beginnen. Eine gute Stelle für den Aufruf ist vor `mysql_init()`.
4. Wenn die Arbeit mit dem Server beendet ist, rufen Sie `mysql_server_end()` auf. Dieser Aufruf könnte etwa nach `mysql_close()` stehen.

Wenn Sie diese Änderungen vorgenommen haben, sieht die Funktion `main()` in *embapp.c* wie folgt aus:

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

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

    if ((opt_err = handle_options (&argc, &argv, my_opts, get_one_option)))
        exit (opt_err);

    /* Kennwort ggf. anfordern */
    if (ask_password)
        opt_password = get_tty_password (NULL);

    /* Datenbanknamen holen, soweit in Befehlszeile vorhanden */
    if (argc > 0)
    {
        opt_db_name = argv[0];
        --argc; ++argv;
    }

    /* Embedded Server initialisieren */
    mysql_server_init (0, NULL, (char **) server_groups);

    /* Verbindungs-Handle initialisieren */
    conn = mysql_init (NULL);
    if (conn == NULL)
```

```

    {
        print_error (NULL, "mysql_init() failed (probably out of memory)");
        exit (1);
    }

    /* Serverbindung herstellen */
    if (mysql_real_connect (conn, opt_host_name, opt_user_name, opt_password,
        opt_db_name, opt_port_num, opt_socket_name, opt_flags) == NULL)
    {
        print_error (conn, "mysql_real_connect() failed");
        mysql_close (conn);
        exit (1);
    }

    while (1)
    {
        char    buf[10000];

        fprintf (stderr, "query> ");                /* Eingabeaufforderung
                                                    drucken */
        if (fgets (buf, sizeof (buf), stdin) == NULL) /* Anfrage lesen */
            break;
        if (strcmp (buf, "quit\n") == 0 || strcmp (buf, "\\q\n") == 0)
            break;
        process_query (conn, buf);                  /* Anfrage ausführen */
    }

    /* Serverbindung trennen */
    mysql_close (conn);
    /* Embedded Server beenden */
    mysql_server_end ();
    exit (0);
}

```

## 6.8.2 Ausführbare Binärdatei für die Anwendung erstellen

Um aus der eingebetteten Anwendung *embapp* eine ausführbare Binärdatei zu erstellen, verknüpfen Sie diese statt mit der Bandbreite `-lmysqlclient` mit der Serverbibliothek `-lmysqlld`. Das Dienstprogramm *mysql\_config* ist hier nützlich. Es zeigt Ihnen nicht nur die Flags, die Sie zu Verknüpfung mit der regulären Clientbibliothek verwenden, sondern auch die für den Embedded Server:

```
% mysql_config --libmysqlld-libs
-L'/usr/local/mysql/lib/mysql' -lmysqlld -lz -lm
```

Daher verwenden Sie zur Erstellung einer eingebetteten Version von *embapp* Befehle wie die folgenden:

```
% gcc -c `mysql_config --cflags` embapp.c
% gcc -o embapp embapp.o `mysql_config --libmysqlld-libs`
```

An dieser Stelle verfügen Sie nun über eine eingebettete Anwendung, die alles enthält, was Sie für den Zugriff auf Ihre MySQL-Datenbanken benötigen. Allerdings müssen Sie bei der Ausführung von *embapp* sicherstellen, dass das Programm nicht das gleiche Datenverzeichnis wie ggf. auf dem gleichen Computer laufende eigenständige Server verwendet. Bei Unix ist ferner zu beachten, dass die Anwendung mit Berechtigungen ausgeführt wird, die ihr Zugriff auf das Datenverzeichnis ermöglichen. Sie können *embapp* entweder ausführen, während Sie als Besitzer des Datenverzeichnisses angemeldet sind, oder Sie erstellen ein *setuid*-Programm, das die Benutzererkennung beim Start auf diesen Benutzer setzt. Wenn Sie beispielsweise *embapp* so konfigurieren wollen, dass es mit den Berechtigungen eines Benutzers namens *mysqladm* ausgeführt wird, setzen Sie als *root* die folgenden Befehle ab:

```
# chown mysqladm embapp
# chmod 4755 embapp
```

Wenn Sie eine nicht-eingebettete Version der Anwendung generieren wollen, die in einem Client/Server-Kontext lauffähig ist, dann müssen Sie sie nur mit der regulären Clientbibliothek verknüpfen. Dies können Sie tun, indem Sie etwa wie folgt kompilieren:

```
% gcc -c `mysql_config --cflags` embapp.c
% gcc -o embapp embapp.o `mysql_config --libs`
```

Die reguläre Clientbibliothek enthält Dummy-Versionen der Routinen `mysql_server_init()` und `mysql_server_end()`, die keine Funktion haben, d.h., es treten keine Verknüpfungsfehler auf.

## 6.9 Verschiedene Themen

In diesem Abschnitt geht es um verschiedene Aspekte in Zusammenhang mit der Verarbeitung von Anfragen, die nicht so recht in andere Abschnitte dieses Kapitels passen:

- Verwendung der Daten aus der Ergebnismenge, um ein Ergebnis zu berechnen, nachdem mithilfe der Metadaten aus der Ergebnismenge sichergestellt wurde, dass die Daten für Ihre Berechnungen geeignet sind
- Umgang mit Daten, deren Eingabe in Anfragen problematisch ist
- Umgang mit Binärdaten
- Ermittlung von Informationen über die Struktur Ihrer Tabellen
- Häufige Fehler bei der MySQL-Programmierung (und wie Sie sie vermeiden)

## 6.9.1 Berechnungen anhand von Ergebnismengen

Bislang haben wir die Metadaten aus den Ergebnismengen hauptsächlich für die Ausgabe von Zeilendaten verwendet, aber es gibt natürlich Situationen, in denen man mehr mit den Daten machen will, als sie nur auszugeben. Sie können beispielsweise statistische Daten basierend auf Datenwerten berechnen, wobei Sie Metadaten verwenden, um sicherzustellen, dass die Daten den verbindlichen Anforderungen entsprechen. Was für Anforderungen könnten das sein? Nun, wenn Sie etwa eine Spalte für numerische Berechnungen verwenden wollen, dann sollte doch zumindest sichergestellt sein, dass diese Spalte Zahlenwerte enthält.

Das folgende Listing zeigt eine einfache Funktion namens `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 innerhalb des Bereichs der Spaltenanzahl liegen; dieser Bereich selbst liegt zwischen 0 und dem Ergebnis der Funktion `mysql_num_fields()` minus 1).
- Die Spalte muss numerische Werte enthalten.

Treffen diese Bedingungen nicht zu, dann 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: Spalte muss */
    /* im zulässigen Wertebereich liegen und numerisch sein */
    if (col_num < 0 || col_num >= mysql_num_fields (res_set))
    {
        print_error (NULL, "illegal column number");
        return;
    }
    mysql_field_seek (res_set, col_num);
    field = mysql_fetch_field (res_set);
    if (!IS_NUM (field->type))
    {
        print_error (NULL, "column is not numeric");
        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]); /* String in Zahl konvertieren */
        sum += val;
        sum_squares += val * val;
    }
}
if (n == 0)
    printf ("No observations\n");
else
{
    printf ("Number of observations: %lu\n", n);
    printf ("Missing observations: %lu\n", missing);
    printf ("Sum: %g\n", sum);
    printf ("Mean: %g\n", sum / n);
    printf ("Sum of squares: %g\n", sum_squares);
    var = ((n * sum_squares) - (sum * sum)) / (n * (n - 1));
    printf ("Variance: %g\n", var);
    printf ("Standard deviation: %g\n", sqrt (var));
}
}
```

Beachten Sie den Aufruf von `mysql_data_seek()`, der der Schleife für `mysql_fetch_row()` vorangeht. Er soll es 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 Standardstatistiken wie etwa einen *t*-Test.

## 6.9.2 Kodierung problematischer Daten in Anfragen

Die direkte Eingabe von Datenwerten, die Anführungszeichen, Nullen oder Backslashes enthalten, kann Probleme bei der Ausführung einer Anfrage verursachen. In diesem Abschnitt wollen wir erläutern, warum das so ist und wie man dieses Problem löst.

Angenommen, Sie wollen eine `SELECT`-Anfrage entwickeln, die den Inhalt des nullterminierten Strings verwendet, auf den die Variable `name` zeigt:

```
char query[1024];
```

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

Falls der Wert von `name` etwa `O'Malley, Brian` lautet, entsteht eine fehlerhafte Anfrage, weil innerhalb eines Strings in Anführungszeichen ein weiteres Anführungszeichen erscheint:

```
SELECT * FROM mytbl 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 des Strings doppelt angeben. Das ist die Konvention in ANSI SQL. MySQL versteht diese Konvention, erlaubt es aber auch, dem Anführungszeichen einen Backslash voranzustellen. Sie können die Anfrage also wie folgt formulieren:

```
SELECT * FROM mytbl WHERE name='O''Malley, Brian'
SELECT * FROM mytbl 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 (darunter Anführungszeichen und Backslashes), ist es nicht sicher, ihn ohne Überprüfung in Anfragen zuzulassen.

Um dieses Problem zu lösen, verwenden Sie `mysql_real_escape_string()`. Diese Routine kodiert Sonderzeichen, sodass Sie sie in Strings einsetzen können, die in Anführungszeichen eingeschlossen sind. `mysql_real_escape_string()` berücksichtigt die folgenden Sonderzeichen: Nullzeichen, einfaches Anführungszeichen, doppeltes Anführungszeichen, Backslash, Zeilenwechsel, Wagenrücklauf und die Tastenkombination `[Strg]+[Z]`. (Letztere ist ein Windows-Sonderzeichen und bezeichnet häufig das Dateiende.)

Wann sollten Sie `mysql_real_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 Strings arbeiten, in denen korrekte Telefonnummern abgelegt sind, die nur aus Ziffern und Trennstrichen bestehen, brauchen Sie `mysql_real_escape_string()` nicht aufzurufen. In vielen anderen Fällen jedoch ist dies eine empfehlenswerte Maßnahme.

`mysql_real_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 und kein Null-Wert ist. Backslash, einfache Anführungszeichen und doppelte Anführungszeichen werden zu `\\`, `\'` und `\"`.

`mysql_real_escape_string()` wird wie folgt aufgerufen:

```
to_len = mysql_real_escape_string (conn, to_str, from_str, from_len);
```

`mysql_real_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 den resultierenden String dann auch in Funktionen wie `strcpy()` und `strlen()` verwenden können.

*from\_str* zeigt auf einen `char`-Puffer mit dem zu kodierenden String. Dieser String kann beliebigen Inhalt haben, auch binäre Daten. *to\_str* zeigt auf einen existierenden `char`-Puffer, in den der kodierte String geschrieben werden soll. Übergeben Sie keinen nicht-initialisierten oder `NULL`-Zeiger, weil Sie damit rechnen, dass `mysql_real_escape_string()` Speicher für Sie reserviert. Die Länge des Puffers, auf den *to\_str* zeigt, muss mindestens  $(from\_len \times 2) + 1$  Bytes betragen (es ist möglich, 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 long`-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 nullterminierter String behandelt werden kann. *to\_len*, der Rückgabewert von `mysql_real_escape_string()`, ist die tatsächliche Länge des resultierenden kodierten Strings, abzüglich der abschließenden Null.

Das von `mysql_real_escape_string()` kodierte und in *to\_str* abgelegte Ergebnis kann als nullterminierter String behandelt werden, weil alle Nullen aus *from\_str* als die druckbare Folge `\0` kodiert wurden.

Um den Code zur Formulierung 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 mytbl WHERE name=");
p += strlen (p);
p += mysql_real_escape_string (conn, p, name, strlen (name));
*p++ = '\'';
*p = '\0';
```

Zugegeben: 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_real_escape_string (conn, buf, name, strlen (name));
sprintf (query, "SELECT * FROM mytbl WHERE name='%s'", buf);
```

Bei MySQL-Versionen vor 3.23.14 ist `mysql_real_escape_string()` noch nicht vorhanden. Als Workaround verwenden Sie hier `mysql_escape_string()`:

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

Der Unterschied besteht darin, dass `mysql_real_escape_string()` den Zeichensatz der aktuellen Verbindung zur Kodierung verwendet, `mysql_escape_string()` hingegen den Standardzeichensatz (weswegen es auch keinen Verbindungs-Handle übernimmt). Wenn Sie einen Quellcode schreiben wollen, der bei jeder MySQL-Version funktioniert, fügen Sie das folgende Codefragment in Ihre Datei ein:

```
#if !defined(MYSQL_VERSION_ID) || (MYSQL_VERSION_ID<32314)
#define mysql_real_escape_string(conn,to_str,from_str,len) \
    ↪      mysql_real_escape_string(to_str,from_str,len)
#endif
```

Schreiben Sie dann Ihren Code unter Verwendung von `mysql_real_escape_string()`; steht diese Funktion nicht zur Verfügung, dann wird sie per `#define` auf `mysql_escape_string()` umgelegt.

### 6.9.3 Bilddaten

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

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

```
CREATE TABLE picture
(
    pict_id      INT NOT NULL PRIMARY KEY,
    pict_data   BLOB
);
```

Um ein Bild aus einer Datei in die Tabelle *picture* zu laden, rufen Sie die Funktion `load_image()` auf. Übergeben Sie ihr dazu eine Kennung 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 long  from_len;
```

```
int          status;

    sprintf (query,
            "INSERT INTO picture (pict_id,pict_data) 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_real_escape_string (conn, p, buf, from_len);
    }
    *p++ = '\\';
    *p++ = ')';
    status = mysql_real_query (conn, query, (unsigned long) (p - 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 (und anderen binären Daten), die Sie aus der Datenbank zurückladen wollen, ist nicht annähernd so kompliziert wie das Ablegen in der Datenbank, 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 zählbare und nicht als nullterminierte Strings.

## 6.9.4 Tabelleninformation ermitteln

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

```
SHOW COLUMNS FROM tbl_name;
SHOW FIELDS FROM tbl_name;
DESCRIBE tbl_name;
EXPLAIN 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 Ergebniszeilen, um die gewünschten Informationen zu extrahieren. Wenn Sie beispielsweise im `mysql`-Client die Anweisung `DESCRIBE president` ausführen, erhalten Sie die folgende Information:

```
mysql> DESCRIBE president;
```

```

+-----+-----+-----+-----+-----+-----+
| Field      | Type      | Null | Key | Default      | Extra |
+-----+-----+-----+-----+-----+-----+
| last_name  | varchar(15) |      |     |               |       |
| first_name | varchar(15) |      |     |               |       |
| suffix     | varchar(5)  | YES  |     | NULL          |       |
| city       | varchar(20) |      |     |               |       |
| state      | char(2)     |      |     |               |       |
| birth      | date        |      |     | 0000-00-00   |       |
| death     | date        | 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:

```

mysql> DESCRIBE president birth;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default      | Extra |
+-----+-----+-----+-----+-----+-----+
| birth | date |      |     | 0000-00-00   |       |
+-----+-----+-----+-----+-----+-----+

```

### 6.9.5 Häufige Fehler bei der Clientprogrammierung

Dieser Abschnitt beschreibt einige Fehler, die häufig bei der Programmierung des C-API von MySQL auftreten, und zeigt, wie man sie vermeidet. Diese Probleme tauchen immer wieder auf der MySQL-Mailingliste auf; ich habe sie nicht erfunden.

#### Fehler 1: Verwendung nicht-initialisierter Zeiger auf Verbindungs-Handles

In den Beispiel in diesem Kapitel haben wir die Funktion `mysql_init()` aufgerufen und ihr das Argument `NULL` übergeben. `mysql_init()` alloziert und initialisiert eine `MYSQL`-Struktur und gibt einen Zeiger darauf zurück. Ein anderer Ansatz bestünde darin, 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 tückischen Problemen führen kann. Die folgenden Beschreibungen sprechen einige Aspekte 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 sonstwohin zeigen kann, wenn die Ausführung von `main()` beginnt. `mysql_init()` verwendet also den Zeiger und gelangt damit in einen willkürlichen Speicherbereich. Wenn Sie Glück haben, zeigt `conn` auf eine Position außerhalb des Adressraums Ihres Programms, das System beendet das Programm 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 folgt noch ein ähnlicher problematischer Codeabschnitt:

```
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 als 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 Funktion des C-APIs von MySQL übergeben, die einen Verbindungs-Handle ungleich `NULL` benötigt, stürzt Ihr Programm ab. Eine Lösung für beide Codeabschnitte bestünde darin 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 jedoch, `mysql_init()` explizit `NULL` zu übergeben, es der Funktion selbst 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 (siehe auch Fehler 2).

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

Denken Sie daran, den Status von Aufrufen zu überprüfen, die fehlschlagen könnten. 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)
{
    /* verarbeite Zeile */
}
```

Wenn `mysql_store_result()` fehlschlägt, ist `res_set` gleich `NULL`, und die `while`-Schleife wird nicht ausgeführt (wird `NULL` an `mysql_fetch_row()` übergeben, dann stürzt das Programm wahrscheinlich ab). Überprüfen Sie den Rückgabewert von Funktionen, die Ergebnismengen zurückgeben, um sicherzustellen, dass wirklich etwas vorliegt, mit dem Sie arbeiten können.

Das gleiche Prinzip gilt auch für alle Funktionen, die fehlschlagen können. Wenn der einer Funktion folgende Code vom Erfolg der Funktion selbst abhängt, dann müssen Sie den Rückgabewert überprüfen und geeignete Schritte für den Fall implementieren, dass die Funktion fehlschlägt. Wenn Sie davon ausgehen, dass die Funktion immer erfolgreich ist, dann sind Probleme unvermeidlich.

### 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 Rechnern 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, sodass 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 Ergebnisbuffer übergeben**

Funktionen aus der Clientbibliothek, für die Sie Puffer bereitstellen müssen, erwarten normalerweise, dass diese Puffer bereits existieren. Der folgende Code stellt einen Verstoß gegen dieses Prinzip dar:

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

len = mysql_real_escape_string (conn, to_str, from_str, strlen (from_str));
```

Wo liegt das Problem? Nun, `to_str` muss auf einen existierenden Puffer verweisen. In diesem Beispiel ist das nicht der Fall; mangels Initialisierung zeigt die Variable auf irgendeine willkürliche Position. Übergeben Sie `mysql_real_escape_string()` keinen nicht initialisierten Zeiger als `to_str`-Argument, falls Sie nicht wirklich beabsichtigen, Ihren Arbeitsspeicher völlig durcheinander zu bringen.