6

The MySQL C API

MySQL provides a client library written in the C programming language that you can use to write client programs that access MySQL databases. This library defines an application-programming interface that includes the following facilities:

- Connection management routines that establish and terminate a session with a server
- Routines that construct queries, send them to the server, and process the results
- Status- and error-reporting functions for determining the exact reason for an error when an API call fails
- Routines that help you process options given in option files or on the command line

This chapter shows how to use the client library to write your own programs using conventions that are reasonably consistent with those used by the client programs included in the MySQL distribution. I assume you know something about programming in C, but I've tried not to assume you're an expert.

The chapter develops a series of client programs in a rough progression from very simple to more complex. The first part of this progression develops the framework for a client skeleton that does nothing but connect to and disconnect from the server. (The reason for this is that although MySQL client programs are written for different purposes, one thing they all have in common is that they must establish a connection to the server.) Development of the framework proceeds in the following stages:

- Begin with some bare-bones connection and disconnection code (`client1`).
- Add error checking (`client2`).
- Add the ability to get connection parameters at runtime, such as the hostname, username, and password (`client3`).

The resulting `client3` program is reasonably generic, so you can use it as the basis for any number of other client programs. After developing it, we'll pause to consider how to handle various kinds of queries. Initially, we'll discuss how to handle specific hard-coded SQL statements and then develop code that can be used to process arbitrary statements. After that, we'll add some query-processing code to `client3` to develop another program (`client4`) that's similar to the `mysql` client and can be used to issue queries interactively.

The chapter then shows how to take advantage of two capabilities that are new in MySQL 4:

- How to write client programs that communicate with the server over secure connections using the Secure Sockets Layer (SSL) protocol
- How to write applications that use `libmysqld`, the embedded server library

Finally, we'll consider (and solve) some common problems, such as, "How can I get information about the structure of my tables?" and "How can I insert images in my database?"

This chapter discusses functions and data types from the client library as they are needed. For a comprehensive listing of all functions and types, see Appendix F, "C API Reference." You can use that appendix as a reference for further background on any part of the client library you're trying to use.

The example programs are available online, so you can try them directly without typing them in yourself. They are part of the `sampdb` distribution; you can find them under the `capi` directory of the distribution. See Appendix A, "Obtaining and Installing Software," for downloading instructions.

Where to Find Example Programs

A common question on the MySQL mailing list is "Where can I find some examples of clients written in C?" The answer, of course, is "right here in this book!" But something many people seem not to consider is that the MySQL distribution itself includes several client programs that happen to be written in C (`mysql`, `mysqladmin`, and `mysqldump`, for example). Because the distribution is readily available in source form, it provides you with quite a bit of example client code. Therefore, if you haven't already done so, grab a source distribution sometime and take a look at the programs in its `client` directory.

General Procedure for Building Client Programs

This section describes the steps involved in compiling and linking a program that uses the MySQL client library. The commands to build clients vary somewhat from system to system, and you may need to modify the commands shown here a bit. However, the description is general and you should be able to apply it to most client programs you write.

Basic System Requirements

When you write a MySQL client program in C, you'll obviously need a C compiler. The examples shown here use `gcc`, which is probably the most common compiler used on UNIX. You'll also need the following in addition to the program's own source files:

- The MySQL header files
- The MySQL client library

The header files and client library constitute the basis of MySQL client programming support. If they are not installed on your system already, you'll need to obtain them. If MySQL was installed on your system from a source or binary distribution, client programming support should have been installed as part of that process. If RPM files were used, this support won't be present unless you installed the developer RPM. Should you need to obtain the MySQL header files and library, see Appendix A.

Compiling and Linking Client Programs

To compile and link a client program, you may need to specify where the MySQL header files and client library are located, because often they are not installed in locations that the compiler and linker search by default. For the following examples, suppose the header file and client library locations are `/usr/local/include/mysql` and `/usr/local/lib/mysql`.

To tell the compiler how to find the MySQL header files when you compile a source file into an object file, pass it an `-I` option that names the appropriate directory. For example, to compile `myclient.c` to produce `myclient.o`, you might use a command like this:

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

To tell the linker where to find the client library and what its name is, pass `-L/usr/local/lib/mysql` and `-lmysqlclient` arguments when you link the object file to produce an executable binary, as follows:

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

If your client consists of multiple files, name all the object files on the link command.

The link step may result in error messages having to do with functions that cannot be found. In such cases, you'll need to supply additional `-l` options to name the libraries containing the functions. If you see a message about `compress()` or `uncompress()`, try adding `-lz` or `-lgz` to tell the linker to search the `zlib` compression library:

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

If the message names the `floor()` function, add `-lm` to link in the math library. You might need to add other libraries as well. For example, you'll probably need `-lsocket` and `-lnsl` on Solaris.

As of MySQL 3.23.21, you can use the `mysql_config` utility to determine the proper flags for compiling and linking MySQL programs. For example, the utility might indicate that the following options are needed:

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

To use `mysql_config` directly within your compile or link commands, invoke it within backticks:

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

The shell will execute `mysql_config` and substitute its output into the surrounding command, which automatically provides the appropriate flags for `gcc`.

If you don't use `make` to build programs, I suggest you learn how so that you won't have to type a lot of program-building commands manually. Suppose you have a client program, `myclient`, that comprises two source files, main.c and aux.c, and a header file, myclient.h. You might write a simple Makefile to build this program as follows. Note that indented lines are indented with tabs; if you use spaces, the Makefile will not work.

```
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
```

Using the Makefile, you can rebuild your program whenever you modify any of the source files simply by running make, which displays and executes the necessary commands:

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

That's easier and less error prone than typing long gcc commands. A Makefile also makes it easier to modify the build process. For example, if your system is one for which you need to link in additional libraries, such as the math and compression libraries, edit the LIBS line in the Makefile to add -lm and -lz:

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

If you need other libraries, add them to the LIBS line as well. Thereafter, when you run make, it will use the updated value of LIBS automatically.

Another way to change make variables other than editing the Makefile is to specify them on the command line. For example, if your C compiler is named cc rather than gcc (as is the case for Mac OS X, for example), you can say so as follows:

```
% make CC=cc
```

If mysql_config is available, you can use it to avoid writing literal include file and library directory pathnames in the Makefile. Write the INCLUDES and LIBS lines as follows instead:

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

When make runs, it will execute each mysql_config command and use its output to set the corresponding variable value. The ${shell} construct shown here is supported by GNU make; you may need to use a somewhat different syntax if your version of make isn't based on GNU make.

If you're using an integrated development environment (IDE), you may not see or use a Makefile at all. The details will depend on your particular IDE.

Client 1—Connecting to the Server

Our first MySQL client program is about as simple as can be—it connects to a server, disconnects, and exits. That's not very useful in itself, but you have to know how to do it because you must be connected to a server before you can do anything with a MySQL database. Connecting to a MySQL server is such a common operation that the code you develop to establish the connection is code you'll use in every client program you write. Additionally, this task gives us something simple to start with. The client can be fleshed out later to do something more useful.

The code for our first client program, client1, consists of a single source file, client1.c:

```
/* client1.c - connect to and disconnect from MySQL server */

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

static char *opt_host_name = NULL;      /* server host (default=localhost) */
static char *opt_user_name = NULL;      /* username (default=login name) */
static char *opt_password = NULL;       /* password (default=none) */
static unsigned int opt_port_num = 0;   /* port number (use built-in value) */
static char *opt_socket_name = NULL;    /* socket name (use built-in value) */
static char *opt_db_name = NULL;        /* database name (default=none) */
static unsigned int opt_flags = 0;      /* connection flags (none) */

static MYSQL *conn;                     /* pointer to connection handler */

int
main (int argc, char *argv[])
{
    /* initialize connection handler */
    conn = mysql_init (NULL);
    /* connect to server */
    mysql_real_connect (conn, opt_host_name, opt_user_name, opt_password,
                opt_db_name, opt_port_num, opt_socket_name, opt_flags);
    /* disconnect from server */
    mysql_close (conn);
    exit (0);
}
```

The source file begins by including the header files my_global.h and mysql.h. Depending on what a MySQL client does, it may need include other header files as well, but usually these two are the bare minimum:

- my_global.h takes care of including several other header files that are likely to be generally useful, such as stdio.h. It also includes windows.h for Windows compatibility if you're compiling the program on Windows. (You may not intend to build the program under Windows yourself, but if you plan to distribute your code, having that file included will help anyone else who does compile under Windows.)
- mysql.h defines the primary MySQL-related constants and data structures.

The order of inclusion is important; my_global.h is intended to be included before any other MySQL-specific header files.

Next, the program declares a set of variables corresponding to the parameters that need to be specified when connecting to the server. For this client, the parameters are hardwired to have default values. Later, we'll develop a more flexible approach that allows the defaults to be overridden using values specified either in option files or on the command line. (That's why the names all begin with opt_; the intent is that eventually those variables will become settable through command options.) The program also declares a pointer to a MYSQL structure that will serve as a connection handler.

The main() function of the program establishes and terminates the connection to the server. Making a connection is a two-step process:

1. Call mysql_init() to obtain a connection handler. When you pass NULL to mysql_init(), it automatically allocates a MYSQL structure, initializes it, and returns a pointer to it. The MYSQL data type is a structure containing information about a connection. Variables of this type are called connection handlers.
2. Call mysql_real_connect() to establish a connection to the server. mysql_real_connect() takes about a zillion parameters:

    - *A pointer to the connection handler*—This should be the value returned by mysql_init().
    - *The server host*—This value is interpreted in a platform-specific way. If you specify a string containing a hostname or IP address on UNIX, the client connects to the given host using a TCP/IP connection. If you specify NULL or the host "localhost", the client connects to the server running on the local host using a UNIX socket.

On Windows, the behavior is similar, except that TCP/IP connections are used instead of UNIX sockets. Also, on Windows NT-based systems, the connection is attempted to the local server using a named pipe if the host is `"."` or `NULL`.

- *The username and password for the MySQL account to be used*—If the name is `NULL`, the client library sends your login name to the server. If the password is `NULL`, no password is sent.
- *The name of the database to select as the default database after the connection has been established*—If this value is `NULL`, no database is selected.
- *The port number and socket file*—The port number is used for TCP/IP connections. The socket name is used for UNIX socket connections (on UNIX) or named pipe connections (on Windows). The values `0` and `NULL` for the parameters tell the client library to use the default port number or socket (or pipe) name.
- *A flags value*—The program passes a value of `0` because it isn't using any special connection options.

You can find more information about `mysql_real_connect()` in Appendix F. For example, the description there discusses in more detail how the hostname parameter interacts with the port number and socket name parameters and lists the options that can be specified in the flags parameter. The appendix also describes `mysql_options()`, which you can use to specify other connection-related options prior to calling `mysql_real_connect()`.

To terminate the connection, invoke `mysql_close()` and pass it a pointer to the connection handler. If you allocated the handler automatically by passing `NULL` to `mysql_init()`, `mysql_close()` will automatically de-allocate the handler when you terminate the connection.

To try out `client1`, compile and link it using the instructions given earlier in the chapter for building client programs, and then run it. Under UNIX, run the program as follows:

```
% ./client1
```

The leading "`./`" may be necessary on UNIX if your shell does not have the current directory ("`.`") in its search path. If the directory is in your search path or you are using Windows, you can omit the "`./`" from the command name:

```
% client1
```

The `client1` program connects to the server, disconnects, and exits. Not very exciting, but it's a start. However, it's *just* a start, because there are two significant shortcomings:

- The client does no error checking, so we don't really know whether or not it actually works!
- The connection parameters (hostname, username, and so forth) are hardwired into the source code. It would be better to give the user the ability to override the parameters by specifying them in an option file or on the command line.

Neither of these problems is difficult to deal with. The next few sections address them both.

Client 2—Adding Error Checking

Our second client will be like the first one, but it will be modified to take into account the fact that errors may occur. It seems to be fairly common in programming texts to say "Error checking is left as an exercise for the reader," probably because checking for errors is—let's face it—such a bore. Nevertheless, it is much better for MySQL client programs to test for error conditions and respond to them appropriately. The client library functions that return status values do so for a reason, and you ignore them at your peril; you'll end up trying to track down obscure problems that occur in your programs due to failure to check for errors, or users of your programs will wonder why those programs behave erratically, or both.

Consider our first program, `client1`. How do you know whether it really connected to the server? You could find out by looking in the server log for `Connect` and `Quit` events corresponding to the time at which you ran the program:

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

Alternatively, you might see an `Access denied` message instead, which indicates that no connection was established at all:

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

Unfortunately, `client1` doesn't tell us which of these outcomes occurred. In fact, it can't. It doesn't perform any error checking, so it doesn't even know itself what happened. That is unacceptable. You certainly shouldn't have to look in the server's log to find out whether you were able to connect to it! Let's fix this problem right away by adding some error checking.

Routines in the MySQL client library that return a value generally indicate success or failure in one of two ways:

- Pointer-valued functions return a non-`NULL` pointer for success and `NULL` for failure. (`NULL` in this context means "a C `NULL` pointer," not "a MySQL `NULL` column value.")

  Of the client library routines we've used so far, `mysql_init()` and `mysql_real_connect()` both return a pointer to the connection handler to indicate success and `NULL` to indicate failure.

- Integer-valued functions commonly return 0 for success and non-zero for failure. It's important not to test for specific non-zero values, such as –1. There is no guarantee that a client library function returns any particular value when it fails. On occasion, you may see code that tests a return value from a C API function `mysql_XXX()` incorrectly, like this:

  ```
  if (mysql_XXX() == -1)        /* this test is incorrect */
      fprintf (stderr, "something bad happened\n");
  ```

  This test might work, and it might not. The MySQL API doesn't specify that any non-zero error return will be a particular value other than that it (obviously) isn't zero. The test should be written either like this:

  ```
  if (mysql_XXX() != 0)        /* this test is correct */
      fprintf (stderr, "something bad happened\n");
  ```

  or like this, which is equivalent, and slightly simpler to write:

  ```
  if (mysql_XXX())            /* this test is correct */
      fprintf (stderr, "something bad happened\n");
  ```

If you look through the source code for MySQL itself, you'll find that generally it uses the second form of the test.

Not every API call returns a value. The other client routine we've used, `mysql_close()`, is one that does not. (How could it fail? And if it did, so what? You were done with the connection, anyway.)

When a client library call does fail, two calls in the API are useful for finding out why. `mysql_error()` returns a string containing an error message, and `mysql_errno()` returns a numeric error code. The argument to both functions is a pointer to the connection handler. You should call them right after an error occurs; if you issue another API call that returns a status, any error information you get from `mysql_error()` or `mysql_errno()` will apply to the later call instead.

Generally, the user of a program will find the error string more enlightening than the error code, so if you report only one of the two values, I suggest it be the string. For completeness, the examples in this chapter report both values.

Taking the preceding discussion into account, we can write our second client program, `client2`, which is similar to `client1` but has proper error-checking code added. The source file, `client2.c`, is as follows:

```
/*
 * client2.c - connect to and disconnect from MySQL server,
 * with error-checking
 */

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

static char *opt_host_name = NULL;      /* server host (default=localhost) */
static char *opt_user_name = NULL;      /* username (default=login name) */
static char *opt_password = NULL;       /* password (default=none) */
static unsigned int opt_port_num = 0;   /* port number (use built-in value) */
static char *opt_socket_name = NULL;    /* socket name (use built-in value) */
static char *opt_db_name = NULL;        /* database name (default=none) */
static unsigned int opt_flags = 0;      /* connection flags (none) */

static MYSQL *conn;                     /* pointer to connection handler */

int
main (int argc, char *argv[])
{
    /* initialize connection handler */
    conn = mysql_init (NULL);
    if (conn == NULL)
    {
        fprintf (stderr, "mysql_init() failed (probably out of memory)\n");
        exit (1);
    }
    /* connect to server */
    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);
    }
    /* disconnect from server */
    mysql_close (conn);
    exit (0);
}
```

The error-checking logic is based on the fact that both mysql_init() and mysql_real_connect() return NULL if they fail. Note that although the program checks the return value of mysql_init(), no error-reporting function is called if it fails. That's because the connection handler cannot be assumed to contain any meaningful information when mysql_init() fails. By contrast, if mysql_real_connect() fails, the connection handler still won't contain information that corresponds to a valid connection, but it will contain diagnostic information that can be passed to the error-reporting functions. The handler can also be passed to mysql_close() to release any memory that may have been allocated automatically for it by mysql_init(). (Don't pass the handler to any other client routines, though! Because they generally assume a valid connection, your program may crash.)

Compile and link client2, and then try running it:

```
% ./client2
```

If client2 produces no output (as just shown), it connected successfully. On the other hand, you might see something like this:

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

This output indicates no connection was established, and it lets you know why. It also means that the first program, client1, never successfully connected to the server either. (After all, client1 used the same connection parameters.) We didn't know it then because client1 didn't bother to check for errors. client2 does check, so it can tell us when something goes wrong.

Knowing about problems is better than not knowing, which is why you should test API function return values. Failure to do so is an unnecessary cause of programming difficulties. This phenomenon plays itself out frequently on the MySQL mailing list. Typical questions are "Why does my program crash when it issues this query?" or "How come my query doesn't return anything?" In many cases, the program in question didn't check whether or not the connection was established successfully before issuing the query or didn't check to make sure the server successfully executed the query before trying to retrieve the results. And when a program doesn't check for errors, the programmer ends up confused. Don't make the mistake of assuming that every client library call succeeds.

The rest of the programs in this chapter perform error checking, and your own programs should, too. It might seem like more work, but in the long run it's really less because you spend less time tracking down subtle problems. I'll also take this approach of checking for errors in Chapter 7, "The Perl DBI API," and Chapter 8, "The PHP API."

Now, suppose you do see an Access denied message when you run the client2 program. How can you fix the problem? One possibility is to recompile the program after modifying the source code to change the initializers for the connection parameters to values that allow you to access your server. That might be beneficial in the sense that at least you'd be able to make a connection. But the values would still be hard-coded into your program. I recommend against that approach, especially for the password value. (You might think that the password becomes hidden when you compile your program into binary executable form, but it's not hidden at all if someone can run the strings utility on the binary. Not to mention the fact that anyone with read access to the source file can get the password with no work at all.)

In the next section, we'll develop more flexible methods of indicating how to connect to the server. But first I want to develop a simpler method for reporting errors, because that's something we'll need to be ready to do often. I will continue to use the style of reporting both the MySQL numeric error code and the descriptive error string when errors occur, but I prefer not to write out the calls to the error functions mysql_errno() and mysql_error() like this each time:

```
if (...some MySQL function fails...)
{
    fprintf (stderr, "...some error message...:\nError %u (%s)\n",
                        mysql_errno (conn), mysql_error (conn));
}
```

It's easier to report errors by using a utility function that can be called like this instead:

```
if (...some MySQL function fails...)
{
    print_error (conn, "...some error message...");
}
```

print_error() prints the error message and calls the MySQL error functions automatically. It's easier to write out the print_error() call than a long fprintf() call, and it also makes the program easier to read. Also, if print_error() is written to do something sensible even when conn is NULL, we can use it under circumstances such as when mysql_init() call fails. Then we won't have a mix of error-reporting calls—some to fprintf() and some to print_error(). A version of print_error() that satisfies this description can be written as follows:

```
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));
        }
    }
```

I can hear someone in the back row objecting, "Well, you don't really have to call both error functions every time you want to report an error, so you're deliberately overstating the tedium of reporting errors that way just so your utility function looks more useful. And you wouldn't really write out all that error-printing code a bunch of times anyway; you'd write it once, and then use copy and paste when you need it again." Those are reasonable objections, but I would address them as follows:

- Even if you use copy and paste, it's easier to do so with shorter sections of code.
- Whether or not you prefer to invoke both error functions each time you report an error, writing out all the error-reporting code the long way leads to the temptation to take shortcuts and be inconsistent when you do report errors. Wrapping the error-reporting code in a utility function that's easy to invoke lessens this temptation and improves coding consistency.
- If you ever do decide to modify the format of your error messages, it's a lot easier if you only need to make the change one place rather than throughout your program. Or, if you decide to write error messages to a log file instead of (or in addition to) writing them to `stderr`, it's easier if you only have to change `print_error()`. This approach is less error prone and, again, lessens the temptation to do the job halfway and be inconsistent.
- If you use a debugger when testing your programs, putting a breakpoint in the error-reporting function is a convenient way to have the program break to the debugger when it detects an error condition.

For these reasons, programs in the rest of this chapter will use `print_error()` to report MySQL-related problems.

Client 3—Getting Connection Parameters at Runtime

Now we're ready to figure out how to do something smarter than using hardwired default connection parameters—such as letting the user specify those values at runtime. The previous client programs have a significant shortcoming in that the connection parameters are written literally into the source code. To change any of those values, you have to edit the source file and recompile it. That's not very convenient, especially if you intend to make your program available for other people to use. One common way to specify connection parameters at runtime is by using command line options. For example, the programs in the MySQL distribution accept parameters in either of two forms, as shown in the following table.

Parameter	Long Option Form	Short Option Form
Hostname	`--host=`*host_name*	`-h` *host_name*
Username	`--user=`*user_name*	`-u` *user_name*
Password	`--password` or	`-p` or
	`--password=`*your_pass*	`-p`*your_pass*
Port number	`--port=`*port_num*	`-P` *port_num*
Socket name	`--socket=`*socket_name*	`-S` *socket_name*

For consistency with the standard MySQL clients, our next client program, `client3`, will accept those same formats. It's easy to do this because the client library includes support for option processing. In addition, our client will have the ability to extract information from option files, which allows you to put connection parameters in `~/.my.cnf` (that is, the `.my.cnf` file in your home directory) or in any of the global option files. Then you don't have to specify the options on the command line each time you invoke the program. The client library makes it easy to check for MySQL option files and pull any relevant values from them. By adding only a few lines of code to your program, you can make it option file-aware, and you don't have to reinvent the wheel by writing your own code to do it. (Option file syntax is described in Appendix E, "MySQL Program Reference.")

Before writing `client3` itself, we'll develop a couple programs that illustrate how MySQL's option-processing support works. These show how option handling works fairly simply and without the added complication of connecting to the MySQL server and processing queries.

Accessing Option File Contents

To read option files for connection parameter values, use the `load_defaults()` function. `load_defaults()` looks for option files, parses their contents for any option groups in which you're interested, and rewrites your program's argument vector (the `argv[]` array) to put information from those groups in the form of command line options at the beginning of argv[]. That way, the options appear to have been specified on the command line so that when you parse the command options, you get the connection parameters as part of your normal option-processing code. The options are added to argv[] immediately after the command name and before any other arguments (rather than at the end), so that any connection parameters specified on the command line occur later than and thus override any options added by load_defaults().

The following is a little program, show_argv, that demonstrates how to use load_defaults() and illustrates how it modifies your argument vector:

```
/* show_argv.c - show effect of load_defaults() on argument vector */

#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);
}
```

The option file-processing code involves several components:

- client_groups[] is an array of character strings indicating the names of the option file groups from which you want to obtain options. Client programs normally include at least "client" in the list (which represents the [client] group), but you can list as many groups as you want. The last element of the array must be NULL to indicate where the list ends.
- my_init() is an initialization routine that performs some setup operations required by load_defaults().

- `load_defaults()` reads the option files. It takes four arguments: the prefix used in the names of your option files (this should always be `"my"`), the array listing the names of the option groups in which you're interested, and the addresses of your program's argument count and vector. Don't pass the values of the count and vector. Pass their addresses instead because `load_defaults()` needs to change their values. Note in particular that even though `argv` is already a pointer, you still pass `&argv`, that pointer's address.

`show_argv` prints its arguments twice to show the effect that `load_defaults()` has on the argument array. First it prints the arguments as they were specified on the command line, and then it calls `load_defaults()` and prints the argument array again.

To see how `load_defaults()` works, make sure you have a `.my.cnf` file in your home directory with some settings specified for the `[client]` group. (On Windows, you can use the `C:\my.cnf` file.) Suppose the file looks like this:

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

If that is the case, executing `show_argv` should produce output like this:

```
% ./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
```

When `show_argv` prints the argument vector the second time, the values in the option file show up as part of the argument list. It's also possible that you'll see some options that were not specified on the command line or in your `~/.my.cnf` file. If this occurs, you will likely find that options for the `[client]` group are listed in a system-wide option file. This can happen because `load_defaults()` actually looks in several option files. On UNIX, it looks in `/etc/my.cnf` and in the `my.cnf` file in the MySQL data directory before reading `.my.cnf` in your home directory. On Windows, load_defaults() reads the my.ini file in your Windows system directory, C:\my.cnf, and the my.cnf file in the MySQL data directory.

Client programs that use load_defaults() almost always specify "client" in the list of option group names (so that they get any general client settings from option files), but you can set up your option file processing code to obtain options from other groups as well. Suppose you want show_argv to read options in both the [client] and [show_argv] groups. To accomplish this, find the following line in show_argv.c:

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

Change the line to this:

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

Then recompile show_argv, and the modified program will read options from both groups. To verify this, add a [show_argv] group to your ~/.my.cnf file:

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

[show_argv]
host=other_host
```

With these changes, invoking show_argv again will produce a different result than before:

```
% ./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
```

The order in which option values appear in the argument array is determined by the order in which they are listed in your option file, not the order in which option group names are listed in the client_groups[] array. This means you'll probably want to specify program-specific groups after the [client] group in your option file. That way, if you specify an option in both groups, the program-specific value will take precedence over the more general [client] group value. You can see this in the example just shown; the host option was specified in both the [client] and [show_argv] groups, but because the [show_argv] group appears last in the option file, its host setting appears later in the argument vector and takes precedence.

load_defaults() does not pick up values from your environment settings. If you want to use the values of environment variables, such as MYSQL_TCP_PORT or MYSQL_UNIX_PORT, you must arrange for that yourself by using getenv(). I'm not going to add that capability to our clients, but what follows is a short code fragment that shows how to check the values of a couple of the standard MySQL-related environment variables:

```
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 the standard MySQL clients, environment variable values have lower precedence than values specified in option files or on the command line. If you check environment variables in your own programs and want to be consistent with that convention, check the environment before (not after) calling load_defaults() or processing command line options.

load_defaults() and Security

On multiple-user systems, utilities such as the ps program can display argument lists for arbitrary processes, including those being run by other users. Because of this, you may be wondering if there are any process-snooping implications of load_defaults() taking passwords that it finds in option files and putting them in your argument list. This actually is not a problem because ps displays the original argv[] contents. Any password argument created by load_defaults() points to an area of memory that it allocates for itself. That area is not part of the original vector, so ps never sees it.

On the other hand, a password that is given on the command line *does* show up in ps. This is one reason why it's not a good idea to specify passwords that way. One precaution a program can take to help reduce the risk is to remove the password from the argument list as soon as it starts executing. The next section, "Processing Command-Line Arguments," shows how to do that.

Processing Command-Line Arguments

Using `load_defaults()`, we can get all the connection parameters into the argument vector, but now we need a way to process the vector. The `handle_options()` function is designed for this. `handle_options()` is built into the MySQL client library, so you have access to it whenever you link in that library.

The option-processing methods described here were introduced in MySQL 4.0.2. Before that, the client library included option-handling code that was based on the `getopt_long()` function. If you're writing MySQL-based programs using the client library from a version of MySQL earlier than 4.0.2, you can use the version of this chapter from the first edition of this book, which describes how to process command options using `getopt_long()`. The first-edition chapter is available online in PDF format at the book's companion Web site at `http://www.kitebird.com/mysql-book/`.

The `getopt_long()`-based code has now been replaced with a new interface based on `handle_options()`. Some of the improvements offered by the new option-processing routines are:

- **More precise specification of the type and range of legal option values**. For example, you can indicate not only that an option must have integer values but that it must be positive and a multiple of 1024.
- **Integration of help text, to make it easy to print a help message by calling a standard library function**. There is no need to write your own special code to produce a help message.
- **Built in support for the standard `--no-defaults`, `--print-defaults`, `--defaults-file`, and `--defaults-extra-file` options**. These options are described in the "Option Files" section in Appendix E.
- **Support for a standard set of option prefixes, such as `--disable-` and `--enable-`, to make it easier to implement boolean (on/off) options**. These capabilities are not used in this chapter, but are described in the option-processing section of Appendix E.

Note: The new option-processing routines appeared in MySQL 4.0.2, but it's best to use 4.0.5 or later. Several problems were identified and fixed during the initial shaking-out period from 4.0.2 to 4.0.5.

To demonstrate how to use MySQL's option-handling facilities, this section describes a `show_opt` program that invokes `load_defaults()` to read option files and set up the argument vector and then processes the result using `handle_options()`.

show_opt allows you to experiment with various ways of specifying connection parameters (whether in option files or on the command line) and to see the result by showing you what values would be used to make a connection to the MySQL server. show_opt is useful for getting a feel for what will happen in our next client program, client3, which hooks up this option-processing code with code that actually does connect to the server.

show_opt illustrates what happens at each phase of argument processing by performing the following actions:

1. Set up default values for the hostname, username, password, and other connection parameters.
2. Print the original connection parameter and argument vector values.
3. Call load_defaults() to rewrite the argument vector to reflect option file contents and then print the resulting vector.
4. Call the option processing routine handle_options() to process the argument vector and then print the resulting connection parameter values and whatever is left in the argument vector.

The following discussion explains how show_opt works, but first take a look at its source file, show_opt.c:

```
/*
 * show_opt.c - demonstrate option processing with load_defaults()
 * and handle_options()
 */

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

static char *opt_host_name = NULL;      /* server host (default=localhost) */
static char *opt_user_name = NULL;      /* username (default=login name) */
static char *opt_password = NULL;       /* password (default=none) */
static unsigned int opt_port_num = 0;   /* port number (use built-in value) */
static char *opt_socket_name = NULL;    /* socket name (use built-in value) */

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

static struct my_option my_opts[] =     /* option information structures */
{
    {"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);    /* print help message */
        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);
}
```

The option-processing approach illustrated by `show_opt.c` involves the following aspects, which will be common to any program that uses the MySQL client library to handle command options:

1. In addition to the `my_global.h` and `mysql.h` header files, include `my_getopt.h` as well. `my_getopt.h` defines the interface to MySQL's option-processing facilities.
2. Define an array of `my_option` structures. In `show_opt.c`, this array is named `my_opts`. The array should have one structure per option that the program understands. Each structure provides information such as an option's short and long names, its default value, whether the value is a number or string, and so on. Details on members of the `my_option` structure are provided shortly.
3. After calling `load_defaults()` to read the option files and set up the argument vector, process the options by calling `handle_options()`. The first two arguments to `handle_options()` are the addresses of your program's argument count and vector. (Just as with `load_options()`, you pass the addresses of these variables, not their values.) The third argument points to the array of `my_option` structures. The fourth argument is a pointer to a helper function. The `handle_options()` routine and the `my_options` structures are designed to make it possible for most option-processing actions to be performed automatically for you by the

client library. However, to allow for special actions that the library does not handle, your program should also define a helper function for `handle_options()` to call. In `show_opt.c`, this function is named `get_one_option()`. The operation of the helper function is described shortly.

The `my_option` structure defines the types of information that must be specified for each option that the program understands. It looks like this:

```
struct my_option
{
  const char *name;               /* option's long name */
  int        id;                  /* option's short name or code */
  const char *comment;            /* option description for help message */
  gptr       *value;              /* pointer to variable to store value in */
  gptr       *u_max_value;        /* The user defined max variable value */
  const char **str_values;        /* array of legal option values (unused) */
  enum get_opt_var_type var_type; /* option value's type */
  enum get_opt_arg_type arg_type; /* whether option value is required */
  longlong   def_value;           /* option's default value */
  longlong   min_value;           /* option's minimum allowable value */
  longlong   max_value;           /* option's maximum allowable value */
  longlong   sub_size;            /* amount to shift value by */
  long       block_size;          /* option value multiplier */
  int        app_type;            /* reserved for application-specific use */
};
```

The members of the `my_option` structure are used as follows:

- `name`

  The long option name. This is the `--`*`name`* form of the option, without the leading dashes. For example, if the long option is `--user`, list it as `"user"` in the `my_option` structure.

- `id`

  The short (single-letter) option name, or a code value associated with the option if it has no single-letter name. For example, if the short option is `-u`, list it as `'u'` in the `my_option` structure. For options that have only a long name and no corresponding single-character name, you should make up a set of option code values to be used internally for the short names. The values must be unique and different than all the single-character names. (To satisfy the latter constraint, make the codes greater than 255, the largest possible single-character value. An example of this technique is shown in "Writing Clients That Include SSL Support" section later in this chapter.)

- comment

  An explanatory string that describes the purpose of the option. This is the text that you want displayed in a help message.

- value

  This is a gptr (generic pointer) value. It points to the variable where you want the option's argument to be stored. After the options have been processed, you can check that variable to see what the option's value has been set to. If the option takes no argument, value can be NULL. Otherwise, the data type of the variable that's pointed to must be consistent with the value of the var_type member.

- u_max_value

  This is another gptr value, but it's used only by the server. For client programs, set u_max_value to NULL.

- str_values

  This member currently is unused. In future MySQL releases, it might be used to allow a list of legal values to be specified, in which case any option value given will be required to match one of these values.

- var_type

  This member indicates what kind of value must follow the option name on the command line and can be any of the following:

var_type Value	**Meaning**
GET_NO_ARG	No value
GET_BOOL	Boolean value
GET_INT	Integer value
GET_UINT	Unsigned integer value
GET_LONG	Long integer value
GET_ULONG	Unsigned long integer value
GET_LL	Long long integer value
GET_ULL	Unsigned long long integer value
GET_STR	String value
GET_STR_ALLOC	String value

  The difference between GET_STR and GET_STR_ALLOC is that for GET_STR, the option variable will be set to point directly at the value in the argument vector, whereas for GET_STR_ALLOC, a copy of the argument will be made and the option variable will be set to point to the copy.

- `arg_type`

  The `arg_type` value indicates whether a value follows the option name and can be any of the following:

arg_type Value	**Meaning**
NO_ARG	Option takes no following argument
OPT_ARG	Option may take a following argument
REQUIRED_ARG	Option requires a following argument

  If `arg_type` is `NO_ARG`, then `var_type` should be set to `GET_NO_ARG`.

- `def_value`

  For numeric-valued options, the option will be assigned this value by default if no explicit value is specified in the argument vector.

- `min_value`

  For numeric-valued options, this is the smallest value that can be specified. Smaller values are bumped up to this value automatically. Use 0 to indicate "no minimum."

- `max_value`

  For numeric-valued options, this is the largest value that can be specified. Larger values are bumped down to this value automatically. Use 0 to indicate "no maximum."

- `sub_size`

  For numeric-valued options, `sub_size` is an offset that is used to convert values from the range as given in the argument vector to the range that is used internally. For example, if values are given on the command line in the range from 1 to 256, but the program wants to use an internal range of 0 to 255, set `sub_size` to 1.

- `block_size`

  For numeric-valued options, if this value is non-zero, it indicates a block size. Option values will be rounded down to the nearest multiple of this size if necessary. For example, if values must be even, set the block size to 2; `handle_options()` will round odd values down to the nearest even number.

- `app_type`

  This is reserved for application-specific use.

The my_opts array should have a my_option structure for each valid option, followed by a terminating structure that is set up as follows to indicate the end of the array:

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

When you invoke handle_options() to process the argument vector, it skips over the first argument (the program name) and then processes option arguments—that is, arguments that begin with a dash. This continues until it reaches the end of the vector or encounters the special "end of options" argument ('--' by itself). As it moves through the argument vector, handle_options() calls the helper function once per option to allow that function to perform any special processing. handle_options() passes three arguments to the helper function—the short option value, a pointer to the option's my_option structure, and a pointer to the argument that follows the option in the argument vector (which will be NULL if the option is specified without a following value).

When handle_options() returns, the argument count and vector will have been reset appropriately to represent an argument list containing only the non-option arguments.

The following is a sample invocation of show_opt and the resulting output (assuming that ~/.my.cnf still has the same contents as for the final show_argv example in the "Accessing Option File Contents" section earlier in this chapter):

```
% ./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
```

The output shows that the hostname is picked up from the command line (overriding the value in the option file) and that the username and password come from the option file. `handle_options()` correctly parses options whether specified in short-option form (such as `-h yet_another_host`) or in long-option form (such as `--user=bill`).

The `get_one_option()` helper function is used in conjunction with `handle_options()`. For `show_opt`, it is fairly minimal and takes no action except for the `--help` or `-?` options (for which `handle_options()` passes an `optid` value of `'?'`):

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

`my_print_help()` is a client library routine that automatically produces a help message for you, based on the option names and comment strings in the `my_opts` array. To see how it works, try the following command; the final part of the output will be the help message:

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

You can add other cases to `get_one_option()` as necessary. For example, this function is useful for handling password options. When you specify such an option, the password value may or may not be given, as indicated by `OPT_ARG` in the option information structure. (That is, you can specify the option as `--password` or `--password=`*`your_pass`* if you use the long-option form or as `-p` or `-p`*`your_pass`* if you use the short-option form.) MySQL clients typically allow you to omit the password value on the command line and then prompt you for it. This allows you to avoid giving the password on the command line, which keeps people from seeing your password. In later programs, we'll use `get_one_option()` to check whether or not a password value was given. We'll save the value if so, and, otherwise, set a flag to indicate that the program should prompt the user for a password before attempting to connect to the server.

You may find it instructive to modify the option structures in `show_opt.c` to see how your changes affect the program's behavior. For example, if you set the minimum, maximum, and block size values for the `--port` option to 100, 1000, and 25, you'll find after recompiling the program that you cannot set the port number to a value outside the range from 100 to 1000 and that values get rounded down automatically to the nearest multiple of 25.

The option processing routines also handle the `--no-defaults`, `--print-defaults`, `--defaults-file`, and `--defaults-extra-file` options automatically. Try invoking `show_opt` with each of these options to see what happens.

Incorporating Option Processing into a MySQL Client Program

Now let's strip out from `show_opt.c` the stuff that's purely illustrative of how the option-handling routines work and use the remainder as a basis for a client that connects to a server according to any options that are provided in an option file or on the command line. The resulting source file, `client3.c`, is as follows:

```
/*
 * client3.c - connect to MySQL server, using connection parameters
 * specified in an option file or on the command line
 */

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

static char *opt_host_name = NULL;      /* server host (default=localhost) */
static char *opt_user_name = NULL;      /* username (default=login name) */
static char *opt_password = NULL;       /* password (default=none) */
static unsigned int opt_port_num = 0;   /* port number (use built-in value) */
static char *opt_socket_name = NULL;    /* socket name (use built-in value) */
static char *opt_db_name = NULL;        /* database name (default=none) */
static unsigned int opt_flags = 0;      /* connection flags (none) */

static int ask_password = 0;            /* whether to solicit password */

static MYSQL *conn;                     /* pointer to connection handler */
```

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

static struct my_option my_opts[] =     /* option information structures */
{
    {"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);    /* print help message */
        exit (0);
    case 'p':                       /* password */
        if (!argument)              /* no value given, so solicit it later */
            ask_password = 1;
        else                        /* copy password, wipe out original */
        {
            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);

    /* solicit password if necessary */
    if (ask_password)
        opt_password = get_tty_password (NULL);

    /* get database name if present on command line */
    if (argc > 0)
    {
        opt_db_name = argv[0];
        --argc; ++argv;
    }

    /* initialize connection handler */
    conn = mysql_init (NULL);
    if (conn == NULL)
    {
        print_error (NULL, "mysql_init() failed (probably out of memory)");
        exit (1);
    }

    /* connect to server */
    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);
    }
```

```
    /* ... issue queries and process results here ... */

    /* disconnect from server */
    mysql_close (conn);
    exit (0);
}
```

Compared to the client1, client2, and show_opt programs that we developed earlier, client3 does a few new things:

- It allows a database to be selected on the command line; just specify the database after the other arguments. This is consistent with the behavior of the standard clients in the MySQL distribution.
- If a password value is present in the argument vector, get_one_option() makes a copy of it and then wipes out the original. This minimizes the time window during which a password specified on the command line is visible to ps or to other system status programs. (The window is only *minimized*, not eliminated. Specifying passwords on the command line still is a security risk.)
- If a password option was given without a value, get_one_option() sets a flag to indicate that the program should prompt the user for a password. That's done in main() after all options have been processed, using the get_tty_password() function. This is a utility routine in the client library that prompts for a password without echoing it on the screen. You may ask, "Why not just call getpass()?" The answer is that not all systems have that function—Windows, for example. get_tty_password() is portable across systems because it's configured to adjust to system idiosyncrasies.

client3 connects to the MySQL server according to the options you specify. Assume there is no option file to complicate matters. If you invoke client3 with no arguments, it connects to localhost and passes your UNIX login name and no password to the server. If instead you invoke client3 as shown in the following command, it prompts for a password (because there is no password value immediately following -p), connects to some_host, and passes the username some_user to the server as well as the password you type:

```
% ./client3 -h some_host -p -u some_user some_db
```

client3 also passes the database name some_db to mysql_real_connect() to make that the current database. If there is an option file, its contents are processed and used to modify the connection parameters accordingly.

The work we've done so far to produce `client3` accomplishes something that's necessary for every MySQL client—connecting to the server using appropriate parameters. The process is implemented by the client skeleton, `client3.c`, which you can use as the basis for other programs. Copy it and add to it any application-specific details. That means you can concentrate more on what you're really interested in—being able to access the content of your databases. All the real action for your application will take place between the `mysql_real_connect()` and `mysql_close()` calls, but what we have now serves as a basic framework that you can use for many different clients. To write a new program, just do the following:

1. Make a copy of `client3.c`.
2. Modify the option-processing loop if you accept additional options other than the standard ones that `client3.c` knows about.
3. Add your own application-specific code between the connect and disconnect calls.

And you're done.

Processing Queries

The purpose of connecting to the server is to conduct a conversation with it while the connection is open. This section shows how to communicate with the server to process queries. Each query you run involves the following steps:

1. **Construct the query.** The way you do this depends on the contents of the query—in particular, whether it contains binary data.
2. **Issue the query by sending it to the server.** The server will execute the query and generate a result.
3. **Process the query result.** This depends on what type of query you issued. For example, a `SELECT` statement returns rows of data for you to process. An `INSERT` statement does not.

One factor to consider in constructing queries is which function to use for sending them to the server. The more general query-issuing routine is `mysql_real_query()`. With this routine, you provide the query as a counted string (a string plus a length). You must keep track of the length of your query string and pass that to `mysql_real_query()`, along with the string itself. Because the query is treated as a counted string rather than as a null-terminated string, it can contain anything, including binary data or null bytes.

The other query-issuing function, `mysql_query()`, is more restrictive in what it allows in the query string but often is easier to use. Any query passed to `mysql_query()` should be a null-terminated string, which means it cannot contain null bytes in the text of the query. (The presence of null bytes within the query string will cause it to be interpreted erroneously as shorter than it really is.) Generally speaking, if your query can contain arbitrary binary data, it might contain null bytes, so you shouldn't use `mysql_query()`. On the other hand, when you are working with null-terminated strings, you have the luxury of constructing queries using standard C library string functions that you're probably already familiar with, such as `strcpy()` and `sprintf()`.

Another factor to consider in constructing queries is whether or not you need to perform any character-escaping operations. This is necessary if you want to construct queries using values that contain binary data or other troublesome characters, such as quotes or backslashes. This is discussed in the "Encoding Problematic Data in Queries" section later in this chapter.

A simple outline of query handling looks like this:

```
if (mysql_query (conn, query) != 0)
{
    /* failure; report error */
}
else
{
    /* success; find out what effect the query had */
}
```

`mysql_query()` and `mysql_real_query()` both return zero for queries that succeed and non-zero for failure. To say that a query "succeeded" means the server accepted it as legal and was able to execute it. It does not indicate anything about the effect of the query. For example, it does not indicate that a `SELECT` query selected any rows or that a `DELETE` statement deleted any rows. Checking what effect the query actually had involves additional processing.

A query may fail for a variety of reasons. Some common causes include the following:

- It contains a syntax error.
- It's semantically illegal—for example, a query that refers to a non-existent column of a table.
- You don't have sufficient privileges to access a table referred to by the query.

Queries can be grouped into two broad categories—those that do not return a result set (a set of rows) and those that do. Queries for statements such as INSERT, DELETE, and UPDATE fall into the "no result set returned" category. They don't return any rows, even for queries that modify your database. The only information you get back is a count of the number of rows affected.

Queries for statements such as SELECT and SHOW fall into the "result set returned" category; after all, the purpose of issuing those statements is to get something back. In the MySQL C API, the result set returned by such statements is represented by the MYSQL_RES data type. This is a structure that contains the data values for the rows and also metadata about the values (such as the column names and data value lengths). Is it legal for a result set to be empty (that is, to contain zero rows).

Handling Queries That Return No Result Set

To process a query that does not return a result set, issue it with mysql_query() or mysql_real_query(). If the query succeeds, you can determine out how many rows were inserted, deleted, or updated by calling mysql_affected_rows().

The following example shows how to handle a query that returns no result set:

```
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));
}
```

Note how the result of mysql_affected_rows() is cast to unsigned long for printing. This function returns a value of type my_ulonglong, but attempting to print a value of that type directly does not work on some systems. (For example, I have observed it to work under FreeBSD but to fail under Solaris.) Casting the value to unsigned long and using a print format of %lu solves the problem. The same principle applies to any other functions that return my_ulonglong values, such as mysql_num_rows() and mysql_insert_id(). If you want your client programs to be portable across different systems, keep this in mind.

mysql_affected_rows() returns the number of rows affected by the query, but the meaning of "rows affected" depends on the type of query. For

INSERT, REPLACE, or DELETE, it is the number of rows inserted, replaced, or deleted. For UPDATE, it is the number of rows updated, which means the number of rows that MySQL actually modified. MySQL does not update a row if its contents are the same as what you're updating it to. This means that although a row might be selected for updating (by the WHERE clause of the UPDATE statement), it might not actually be changed.

This meaning of "rows affected" for UPDATE actually is something of a controversial point because some people want it to mean "rows matched"—that is, the number of rows selected for updating, even if the update operation doesn't actually change their values. If your application requires such a meaning, you can request that behavior when you connect to the server by passing a value of CLIENT_FOUND_ROWS in the flags parameter to mysql_real_connect().

Handling Queries That Return a Result Set

Queries that return data do so in the form of a result set that you deal with after issuing the query by calling mysql_query() or mysql_real_query(). It's important to realize that in MySQL, SELECT is not the only statement that returns rows. Statements such as SHOW, DESCRIBE, EXPLAIN, and CHECK TABLE do so as well. For all of these statements, you must perform additional row-handling processing after issuing the query.

Handling a result set involves the following steps:

1. **Generate the result set by calling mysql_store_result() or mysql_use_result().** These functions return a MYSQL_RES pointer for success or NULL for failure. Later, we'll go over the differences between mysql_store_result() and mysql_use_result(), as well as the conditions under which you would choose one over the other. For now, our examples use mysql_store_result(), which retrieves the rows from the server immediately and stores them in the client.
2. **Call mysql_fetch_row() for each row of the result set.** This function returns a MYSQL_ROW value or NULL when there are no more rows. A MYSQL_ROW value is a pointer to an array of strings representing the values for each column in the row. What you do with the row depends on your application. You might simply print the column values, perform some statistical calculation on them, or do something else altogether.

3. **When you are done with the result set, call `mysql_free_result()` to de-allocate the memory it uses.** If you neglect to do this, your application will leak memory. It's especially important to dispose of result sets properly for long-running applications; otherwise, you will notice your system slowly being taken over by processes that consume ever-increasing amounts of system resources.

The following example outlines how to process a query that returns a result set:

```
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);    /* generate result set */
    if (res_set == NULL)
            print_error (conn, "mysql_store_result() failed");
    else
    {
        /* process result set, then deallocate it */
        process_result_set (conn, res_set);
        mysql_free_result (res_set);
    }
}
```

The example hides the details of result set processing within another function, `process_result_set()`. We haven't defined that function yet, so we need to do so. Generally, operations that handle a result set are based on a loop that looks something like this:

```
MYSQL_ROW row;

while ((row = mysql_fetch_row (res_set)) != NULL)
{
    /* do something with row contents */
}
```

`mysql_fetch_row()` returns a `MYSQL_ROW` value, which is a pointer to an array of values. If the return value is assigned to a variable named `row`, each value within the row can be accessed as `row[i]`, where *`i`* ranges from 0 to the number of columns in the row minus one. There are several important points about the `MYSQL_ROW` data type to note:

- `MYSQL_ROW` is a pointer type, so you declare a variable of that type as `MYSQL_ROW row`, not as `MYSQL_ROW *row`.

- Values for all data types, even numeric types, are returned in the MYSQL_ROW array as strings. If you want to treat a value as a number, you must convert the string yourself.
- The strings in a MYSQL_ROW array are null-terminated. However, if a column may contain binary data, it can contain null bytes, so you should not treat the value as a null-terminated string. Get the column length to find out how long the column value is. (The "Using Result Set Metadata" section later in this chapter discusses how to determine column lengths.)
- NULL values are represented by NULL pointers in the MYSQL_ROW array. Unless you have declared a column NOT NULL, you should always check whether values for that column are NULL or your program may crash by attempting to dereference a NULL pointer.

What you do with each row will depend on the purpose of your application. For purposes of illustration, let's just print the rows with column values separated by tabs. To do that, it's necessary to know how many columns values rows contain. That information is returned by another client library function, mysql_num_fields().

The following is the code for 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() displays the contents of each row in tab-delimited format (displaying NULL values as the word "NULL"), and then prints a count of the number of rows retrieved. That count is available by calling mysql_num_rows(). Like mysql_affected_rows(), mysql_num_rows() returns a my_ulonglong value, so you should cast its value to unsigned long and use a %lu format to print it. But note that unlike mysql_affected_rows(), which takes a connection handler argument, mysql_num_rows() takes a result set pointer as its argument.

The code that follows the loop includes an error test. That's just a precautionary measure. If you create the result set with mysql_store_result(), a NULL return value from mysql_fetch_row() always means "no more rows." However, if you create the result set with mysql_use_result(), a NULL return value from mysql_fetch_row() can mean "no more rows" or that an error occurred. Because process_result_set() has no idea whether its caller used mysql_store_result() or mysql_use_result() to generate the result set, the error test allows it to detect errors properly either way.

The version of process_result_set() just shown takes a rather minimalist approach to printing column values—one that has certain shortcomings. For example, suppose you execute the following query:

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

You will receive the following output, which is not so easy to read:

```
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
...
```

We could make the output prettier by providing information such as column labels and making the values line up vertically. To do that, we need the labels, and we need to know the widest value in each column. That information is available, but not as part of the column data values—it's part of the result set's metadata (data about the data). After we generalize our query handler a bit, we'll write a nicer display formatter in the "Using Result Set Metadata" section later in this chapter.

> **Printing Binary Data**
> Column values containing binary data that may include null bytes will not print properly using the `%s` `printf()` format specifier; `printf()` expects a null-terminated string and will print the column value only up to the first null byte. For binary data, it's best to use the column length so that you can print the full value. For example, you could use `fwrite()`.

A General Purpose Query Handler

The preceding query-handling examples were written using knowledge of whether or not the statement should return any data. That was possible because the queries were hardwired into the code; we used an `INSERT` statement, which does not return a result set, and a `SHOW TABLES` statement, which does.

However, you may not always know what kind of statement a given query represents. For example, if you execute a query that you read from the keyboard or from a file, it might be anything. You won't know ahead of time whether or not to expect it to return rows, or even whether it's legal. What then? You certainly don't want to try to parse the query to determine what kind of statement it is. That's not as simple as it might seem, anyway. It's not sufficient to see if the first word is `SELECT`, because the statement might begin with a comment, as follows:

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

Fortunately, you don't have to know the query type in advance to be able to handle it properly. The MySQL C API makes it possible to write a general purpose query handler that correctly processes any kind of statement, whether or not it returns a result set, and whether or not it executes successfully. Before writing the code for this handler, let's outline the procedure that it implements:

1. Issue the query. If it fails, we're done.
2. If the query succeeds, call `mysql_store_result()` to retrieve the rows from the server and create a result set.
3. If `mysql_store_result()` succeeds, the query returned a result set. Process the rows by calling `mysql_fetch_row()` until it returns `NULL`, and then free the result set.
4. If `mysql_store_result()` fails, it could be that the query does not return a result set, or that it should have but an error occurred while trying to retrieve the set. You can distinguish between these outcomes by

passing the connection handler to `mysql_field_count()` and checking its return value:

- If `mysql_field_count()` returns 0, it means the query returned no columns, and thus no result set. (This indicates the query was a statement such as `INSERT`, `DELETE`, or `UPDATE`).
- If `mysql_field_count()` returns a non-zero value, it means that an error occurred, because the query should have returned a result set but didn't. This can happen for various reasons. For example, the result set may have been so large that memory allocation failed, or a network outage between the client and the server may have occurred while fetching rows.

The following listing shows a function that processes any query, given a connection handler and a null-terminated query string:

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

    if (mysql_query (conn, query) != 0) /* the query failed */
    {
        print_error (conn, "Could not execute query");
        return;
    }

    /* the query succeeded; determine whether or not it returns data */

    res_set = mysql_store_result (conn);
    if (res_set)            /* a result set was returned */
    {
        /* process rows, then free the result set */
        process_result_set (conn, res_set);
        mysql_free_result (res_set);
    }
    else                    /* no result set was returned */
    {
        /*
         * does the lack of a result set mean that the query didn't
         * return one, or that it should have but an error occurred?
         */
        if (mysql_field_count (conn) == 0)
        {
            /*
             * query generated no result set (it was not a SELECT, SHOW,
             * DESCRIBE, etc.), so just report number of rows affected
             */
```

```
                printf ("%lu rows affected\n",
                            (unsigned long) mysql_affected_rows (conn));
            }
            else    /* an error occurred */
            {
                print_error (conn, "Could not retrieve result set");
            }
        }
    }
```

A slight complication to this procedure is that `mysql_field_count()` doesn't exist prior to MySQL 3.22.24. The workaround for earlier versions is to call `mysql_num_fields()` instead. To write programs that work with any version of MySQL, include the following code fragment in your source file after including `mysql.h` and before invoking `mysql_field_count()`:

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

The `#define` converts calls to `mysql_field_count()` into invocations of `mysql_num_fields()` for versions of MySQL earlier than 3.22.24.

Alternative Approaches to Query Processing

The version of `process_query()` just shown has the following three properties:

- It uses `mysql_query()` to issue the query.
- It uses `mysql_store_query()` to retrieve the result set.
- When no result set is obtained, it uses `mysql_field_count()` to distinguish occurrence of an error from a result set not being expected.

Alternative approaches are possible for all three of these aspects of query handling:

- You can use a counted query string and `mysql_real_query()` rather than a null-terminated query string and `mysql_query()`.
- You can create the result set by calling `mysql_use_result()` rather than `mysql_store_result()`.
- You can call `mysql_error()` or `mysql_errno()` rather than `mysql_field_count()` to determine whether result set retrieval failed or whether there was simply no set to retrieve.

Any or all of these approaches can be used instead of those used in `process_query()`. The following is a `process_real_query()` function that is analogous to `process_query()` but that uses all three alternatives:

```
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)   /* the query failed */
    {
        print_error (conn, "Could not execute query");
        return;
    }

    /* the query succeeded; determine whether or not it returns data */

    res_set = mysql_use_result (conn);
    if (res_set)            /* a result set was returned */
    {
        /* process rows, then free the result set */
        process_result_set (conn, res_set);
        mysql_free_result (res_set);
    }
    else                    /* no result set was returned */
    {
        /*
         * does the lack of a result set mean that the query didn't
         * return one, or that it should have but an error occurred?
         */
        if (mysql_errno (conn) == 0)
        {
            /*
             * query generated no result set (it was not a SELECT, SHOW,
             * DESCRIBE, etc.), so just report number of rows affected
             */
            printf ("%lu rows affected\n",
                        (unsigned long) mysql_affected_rows (conn));
        }
        else    /* an error occurred */
        {
            print_error (conn, "Could not retrieve result set");
        }
    }
}
```

mysql_store_result() and mysql_use_result() Compared

The `mysql_store_result()` and `mysql_use_result()` functions are similar in that both take a connection handler argument and return a result set. However, the differences between them actually are quite extensive. The primary difference between the two functions lies in the way rows of the result set are retrieved from the server. `mysql_store_result()` retrieves all the rows immediately when you call it. `mysql_use_result()` initiates the retrieval but doesn't actually get any of the rows. These differing approaches to row retrieval give rise to all other differences between the two functions. This section compares them so you'll know how to choose the one that's most appropriate for a given application.

When `mysql_store_result()` retrieves a result set from the server, it fetches the rows, allocates memory for them, and stores them in the client. Subsequent calls to `mysql_fetch_row()` never return an error because they simply pull a row out of a data structure that already holds the result set. Consequently, a `NULL` return from `mysql_fetch_row()` always means you've reached the end of the result set.

By contrast, `mysql_use_result()` doesn't retrieve any rows itself. Instead, it simply initiates a row-by-row retrieval, which you must complete yourself by calling `mysql_fetch_row()` for each row. In this case, although a `NULL` return from `mysql_fetch_row()` normally still means the end of the result set has been reached, it may mean instead that an error occurred while communicating with the server. You can distinguish the two outcomes by calling `mysql_errno()` or `mysql_error()`.

`mysql_store_result()` has higher memory and processing requirements than does `mysql_use_result()` because the entire result set is maintained in the client. The overhead for memory allocation and data structure setup is greater, and a client that retrieves large result sets runs the risk of running out of memory. If you're going to retrieve a lot of rows in a single result set, you may want to use `mysql_use_result()` instead.

`mysql_use_result()` has lower memory requirements because only enough space to handle a single row at a time need be allocated. This can be faster because you're not setting up as complex a data structure for the result set. On the other hand, `mysql_use_result()` places a greater burden on the server, which must hold rows of the result set until the client sees fit to retrieve all of them. This makes `mysql_use_result()` a poor choice for certain types of clients:

- Interactive clients that advance from row to row at the request of the user. (You don't want the server having to wait to send the next row just because the user decides to take a coffee break.)
- Clients that do a lot of processing between row retrievals.

In both of these types of situations, the client fails to retrieve all rows in the result set quickly. This ties up the server and can have a negative impact on other clients because tables from which you retrieve data are read-locked for the duration of the query. Any clients that are trying to update those tables or insert rows into them will be blocked.

Offsetting the additional memory requirements incurred by `mysql_store_result()` are certain benefits of having access to the entire result set at once. All rows of the set are available, so you have random access into them; the `mysql_data_seek()`, `mysql_row_seek()`, and `mysql_row_tell()` functions allow you to access rows in any order you want. With `mysql_use_result()`, you can access rows only in the order in which they are retrieved by `mysql_fetch_row()`. If you intend to process rows in any order other than sequentially as they are returned from the server, you must use `mysql_store_result()` instead. For example, if you have an application that allows the user to browse back and forth among the rows selected by a query, you'd be best served by using `mysql_store_result()`.

With `mysql_store_result()`, you have access to certain types of column information that are unavailable when you use `mysql_use_result()`. The number of rows in the result set is obtained by calling `mysql_num_rows()`. The maximum widths of the values in each column are stored in the `max_width` member of the `MYSQL_FIELD` column information structures. With `mysql_use_result()`, `mysql_num_rows()` doesn't return the correct value until you've fetched all the rows; similarly, `max_width` is unavailable because it can be calculated only after every row's data have been seen.

Because `mysql_use_result()` does less work than `mysql_store_result()`, it imposes a requirement that `mysql_store_result()` does not; the client must call `mysql_fetch_row()` for every row in the result set. If you fail to do this before issuing another query, any remaining records in the current result set become part of the next query's result set and an "out of sync" error occurs. (You can avoid this by calling `mysql_free_result()` before issuing the second query. `mysql_free_result()` will fetch and discard any pending rows for you.) One implication of this processing model is that with `mysql_use_result()` you can work only with a single result set at a time.

Sync errors do not happen with `mysql_store_result()` because when that function returns, there are no rows yet to be fetched from the server. In fact, with `mysql_store_result()`, you need not call `mysql_fetch_row()` explicitly at all. This can sometimes be useful if all that you're interested in is whether you got a non-empty result rather than what the result contains. For example, to find out whether a table `mytbl` exists, you can execute the following query:

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

If, after calling `mysql_store_result()`, the value of `mysql_num_rows()` is non-zero, the table exists. `mysql_fetch_row()` need not be called.

Result sets generated with `mysql_store_result()` should be freed with `mysql_free_result()` at some point, but this need not necessarily be done before issuing another query. This means that you can generate multiple result sets and work with them simultaneously, in contrast to the "one result set at a time" constraint imposed when you're working with `mysql_use_result()`.

If you want to provide maximum flexibility, give users the option of selecting either result set processing method. `mysql` and `mysqldump` are two programs that do this. They use `mysql_store_result()` by default but switch to `mysql_use_result()` if you specify the `--quick` option.

Using Result Set Metadata

Result sets contain not only the column values for data rows but also information about the data. This information is called the result set metadata, which includes:

- The number of rows and columns in the result set, available by calling `mysql_num_rows()` and `mysql_num_fields()`.
- The length of each column value in the current row, available by calling `mysql_fetch_lengths()`.
- Information about each column, such as the column name and type, the maximum width of each column's values, and the table the column comes from. This information is stored in `MYSQL_FIELD` structures, which typically are obtained by calling `mysql_fetch_field()`. Appendix F describes the `MYSQL_FIELD` structure in detail and lists all functions that provide access to column information.

Metadata availability is partially dependent on your result set processing method. As indicated in the previous section, if you want to use the row count or maximum column length values, you must create the result set with `mysql_store_result()`, not with `mysql_use_result()`.

Result set metadata is helpful for making decisions about how to process result set data:

- Column names and widths are useful for producing nicely formatted output that has column titles and that lines up vertically.
- You use the column count to determine how many times to iterate through a loop that processes successive column values for data rows.
- You can use the row or column counts if you need to allocate data structures that depend on knowing the dimensions of the result set.
- You can determine the data type of a column. This allows you to tell whether a column represents a number, whether it contains binary data, and so forth.

Earlier, in the "Handling Queries That Return Data" section, we wrote a version of `process_result_set()` that printed columns from result set rows in tab-delimited format. That's good for certain purposes (such as when you want to import the data into a spreadsheet), but it's not a nice display format for visual inspection or for printouts. Recall that our earlier version of `process_result_set()` produced this output:

```
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
...
```

Let's write a different version of `process_result_set()` that produces tabular output instead by titling and "boxing" each column. This version will display those same results in a format that's easier to look at:

```
+------------+--------------+---------------------+-------+
| 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    |
...
+------------+--------------+---------------------+-------+
```

The general outline of the display algorithm is as follows:

1. Determine the display width of each column.
2. Print a row of boxed column labels (delimited by vertical bars and preceded and followed by rows of dashes).
3. Print the values in each row of the result set, with each column boxed (delimited by vertical bars) and lined up vertically. In addition, print numbers right justified and print the word "NULL" for `NULL` values.
4. At the end, print a count of the number of rows retrieved.

This exercise provides a good demonstration of the use of result set metadata because it requires knowledge of quite a number of things about the result set other than just the values of the data contained in its rows.

You may be thinking to yourself, "Hmm, that description sounds suspiciously similar to the way `mysql` displays its output." Yes, it does, and you're welcome to compare the source for `mysql` to the code we end up with for `process_result_set()`. They're not the same, and you may find it instructive to compare the two approaches to the same problem.

First, it's necessary to determine the display width of each column. The following listing shows how to do this. Observe that the calculations are based entirely on the result set metadata and make no reference whatsoever to the row values:

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

/* determine column display widths -- requires result set to be */
/* generated with mysql_store_result(), not 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 = length of the word "NULL" */
    field->max_length = col_len;    /* reset column info */
}
```

This code calculates column widths by iterating through the `MYSQL_FIELD` structures for the columns in the result set. We position to the first structure by calling `mysql_field_seek()`. Subsequent calls to `mysql_fetch_field()` return pointers to the structures for successive columns. The width of a column for display purposes is the maximum of three values, each of which depends on metadata in the column information structure:

- The length of `field->name`, the column title.
- `field->max_length`, the length of the longest data value in the column.
- The length of the string "NULL" if the column can contain `NULL` values. `field->flags` indicates whether or not the column can contain `NULL`.

Notice that after the display width for a column is known, we assign that value to `max_length`, which is a member of a structure that we obtain from the client library. Is that allowable, or should the contents of the `MYSQL_FIELD` structure be considered read-only? Normally, I would say "read-only," but some of the client programs in the MySQL distribution change the `max_length` value in a similar way, so I assume it's okay. (If you prefer an alternative approach that doesn't modify `max_length`, allocate an array of `unsigned long` values and store the calculated widths in that array.)

The display width calculations involve one caveat. Recall that `max_length` has no meaning when you create a result set using `mysql_use_result()`. Because we need `max_length` to determine the display width of the column values, proper operation of the algorithm requires that the result set be generated using `mysql_store_result()`. In programs that use `mysql_use_result()` rather than `mysql_store_result()`, one possible workaround is to use the `length` member of the `MYSQL_FIELD` structure, which tells you the maximum length that column values can be.

When we know the column widths, we're ready to print. Titles are easy to handle; for a given column, we simply use the column information structure pointed to by field and print the `name` member, using the width calculated earlier:

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

For the data, we loop through the rows in the result set, printing column values for the current row during each iteration. Printing column values from the row is a bit tricky because a value might be `NULL`, or it might represent a number (in which case we print it right justified). Column values are printed as follows, where `row[i]` holds the data value and `field` points to the column information:

```
if (row[i] == NULL)             /* print the word "NULL" */
    printf (" %-*s |", (int) field->max_length, "NULL");
else if (IS_NUM (field->type))  /* print value right-justified */
    printf (" %*s |", (int) field->max_length, row[i]);
else                            /* print value left-justified */
    printf (" %-*s |", (int) field->max_length, row[i]);
```

The value of the IS_NUM() macro is true if the column type indicated by field->type is one of the numeric types, such as INT, FLOAT, or DECIMAL.

The final code to display the result set is as follows. Note that because we're printing lines of dashes multiple times, it's easier to write a print_dashes() function to do so rather than to repeat the dash-generation code several places:

```
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;

    /* determine column display widths -- requires result set to be */
    /* generated with mysql_store_result(), not 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 = length of the word "NULL" */
        field->max_length = col_len;    /* reset column info */
    }

    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)             /* print the word "NULL" */
                    printf (" %-*s |", (int) field->max_length, "NULL");
                else if (IS_NUM (field->type))  /* print value right-justified */
                    printf (" %*s |", (int) field->max_length, row[i]);
                else                            /* print value left-justified */
                    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));
    }
```

The MySQL client library provides several ways of accessing the column information structures. For example, the code in the preceding example accesses these structures several times using loops of the following general form:

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

However, the `mysql_field_seek()`/`mysql_fetch_field()` combination is only one way of getting `MYSQL_FIELD` structures. See the entries for the `mysql_fetch_fields()` and `mysql_fetch_field_direct()` functions in Appendix F for other ways of getting column information structures.

Client 4—An Interactive Query Program

Let's put together much of what we've developed so far and use it to write a simple interactive client, `client4`. This program lets you enter queries, executes them using our general purpose query handler `process_query()`, and displays the results using the `process_result_set()` display formatter developed in the preceding section.

`client4` will be similar in some ways to `mysql`, although of course not with as many features. There are several restrictions on what `client4` will allow as input:

- Each input line must contain a single complete statement.
- Statements should not be terminated by a semicolon or by `\g`.
- The only non-SQL commands that are recognized are `quit` and `\q`, which terminate the program. You can also use Ctrl-D to quit.

It turns out that `client4` is almost completely trivial to write (about a dozen lines of new code). Almost everything we need is provided by our client program skeleton (`client3.c`) and by other functions that we have written already. The only thing we need to add is a loop that collects input lines and executes them.

To construct `client4`, begin by copying the client skeleton `client3.c` to `client4.c`. Then add to that the code for the `process_query()`, `process_result_set()`, and `print_dashes()` functions. Finally, in `client4.c`, look for the line in `main()` that says this:

```
/* ... issue queries and process results here ... */
```

Replace that line with the following `while` loop:

```
while (1)
{
    char    buf[10000];

    fprintf (stderr, "query> ");                    /* print prompt */
    if (fgets (buf, sizeof (buf), stdin) == NULL)   /* read query */
        break;
    if (strcmp (buf, "quit\n") == 0 || strcmp (buf, "\\q\n") == 0)
        break;
    process_query (conn, buf);                      /* execute query */
}
```

Compile `client4.c` to produce `client4.o`, link `client4.o` with the client library to produce `client4`, and you're done. You have an interactive MySQL client program that can execute any query and display the results. The following example shows how the program works, both for `SELECT` and non-`SELECT` queries, as well as for statements that are erroneous:

```
% ./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')
```

Writing Clients That Include SSL Support

MySQL 4 includes SSL support, which you can use in your own programs to access the server over secure connections. To show how this is done, this section describes the process of modifying `client4` to produce a similar client named `sslclient` that outwardly is much the same but allows encrypted connections to be established. For `sslclient` to work properly, MySQL must have been built with SSL support, and the server must be started with the proper options that identify its certificate and key files. You'll also need certificate and key files on the client end. For more information, see the "Setting Up Secure Connections" section in Chapter 12, "Security." In addition, you should use MySQL 4.0.5 or later. The SSL and option-handling routines for earlier 4.0.x releases will not behave quite as described here.

The sampdb distribution contains a source file, sslclient.c, from which the client program sslclient can be built. The following procedure describes how sslclient.c is created, beginning with client4.c:

1. Copy client4.c to sslclient.c. The remaining steps apply to sslclient.c.
2. To allow the compiler to detect whether SSL support is available, the MySQL header file my_config.h defines the symbol HAVE_OPENSSL appropriately. This means that when writing SSL-related code, you use the following construct so that the code will be ignored if SSL cannot be used:

   ```
   #ifdef HAVE_OPENSSL
       ...SSL-related code here...
   #endif
   ```

   my_config.h is included by my_global.h. sslclient.c already includes the latter file, so you need not include my_config.h explicitly.
3. Modify the my_opts array that contains option information structures to include entries for the standard SSL-related options as well (--ssl-ca, --ssl-key, and so on). The easiest way to do this is to include the contents of the sslopt-longopts.h file into the my_opts array with an #include directive. After making the change, my_opts looks like this:

   ```
   static struct my_option my_opts[] =     /* option information structures */
   {
       {"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` is a public MySQL header file. Its contents look like this (reformatted slightly):

```
#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 */
```

4. The option structures defined by `sslopt-longopts.h` refer to the values `OPT_SSL_SSL`, `OPT_SSL_KEY`, and so on. These are used for the short option codes and must be defined by your program, which can be done by adding the following lines preceding the definition of the `my_opts` array:

```
#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
```

When writing your own applications, if a given program also defines codes for other options, make sure these `OPT_SSL_XXX` symbols have different values than those codes.

5. The SSL-related option structures in `sslopt-longopts.h` refer to a set of variables that are used to hold the option values. To declare these, use an `#include` directive to include the contents of the `sslopt-vars.h` file into your program preceding the definition of the `my_opts` array. `sslopt-vars.h` looks like this:

```
#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
```

6. In the `get_one_option()` routine, add a line that includes the `sslopt-case.h` file:

```
my_bool
get_one_option (int optid, const struct my_option *opt, char *argument)
{
    switch (optid)
    {
    case '?':
        my_print_help (my_opts);    /* print help message */
        exit (0);
    case 'p':                       /* password */
        if (!argument)              /* no value given, so solicit it later */
            ask_password = 1;
        else                        /* copy password, wipe out original */
        {
            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 includes cases for the switch() statement that detect when any of the SSL options were given and sets the opt_use_ssl variable if so. It looks like this:

```
#ifdef HAVE_OPENSSL
    case OPT_SSL_KEY:
    case OPT_SSL_CERT:
    case OPT_SSL_CA:
    case OPT_SSL_CAPATH:
    case OPT_SSL_CIPHER:
    /*
      Enable use of SSL if we are using any ssl option
      One can disable SSL later by using --skip-ssl or --ssl=0
    */
      opt_use_ssl= 1;
      break;
#endif
```

The effect of this is that after option processing has been done, it is possible to determine whether the user wants a secure connection by checking the value of opt_use_ssl.

If you follow the preceding procedure, the usual load_defaults() and handle_options() routines will take care of parsing the SSL-related options and setting their values for you automatically. The only other thing you need to do is pass SSL option information to the client library before connecting to the server if the options indicate that the user wants an SSL connection. Do this by invoking mysql_ssl_set() after calling mysql_init() and before calling mysql_real_connect(). The sequence looks like this:

```
    /* initialize connection handler */
    conn = mysql_init (NULL);
    if (conn == NULL)
    {
        print_error (NULL, "mysql_init() failed (probably out of memory)");
        exit (1);
    }

#ifdef HAVE_OPENSSL
    /* pass SSL information to client library */
```

```
    if (opt_use_ssl)
        mysql_ssl_set (conn, opt_ssl_key, opt_ssl_cert, opt_ssl_ca,
                        opt_ssl_capath, opt_ssl_cipher);
#endif

    /* connect to server */
    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);
    }
```

Note that you don't test `mysql_ssl_set()` to see if it returns an error. Any problems with the information you supply to that function will result in an error when you call `mysql_real_connect()`.

Produce `sslclient` by compiling `sslclient.c` and then run it. Assuming that the `mysql_real_connect()` call succeeds, you can proceed to issue queries. If you invoke `sslclient` with the appropriate SSL options, communication with the server should occur over an encrypted connection. To determine whether or not that is so, issue the following query:

```
SHOW STATUS LIKE 'Ssl_cipher'
```

The value of `Ssl_cipher` will be non-blank if an encryption cipher is in use. (To make this easier, the version of `sslclient` included in the `sampdb` distribution actually issues the query for you and reports the result.)

Using the Embedded Server Library

MySQL 4 introduces an embedded server library, `libmysqld`, that contains the server in a form that can be linked (embedded) into applications. This allows you to produce MySQL-based applications that stand on their own, as opposed to applications that connect as a client over a network to a separate server program.

To write an embedded server application, two requirements must be satisfied. First, the embedded server library must be installed:

- If you're building from source, enable the library by using the `--with-embedded-server` option when you run `configure`.
- For binary distributions, use a Max distribution if the non-Max distribution doesn't include `libmysqld`.
- For RPM installs, make sure to install the embedded server RPM.

Second, you'll need to include a small amount of code in your application to start up and shut down the server.

After making sure that both requirements are met, it's necessary only to compile the application and link in the embedded server library (-lmysqld) rather than the regular client library (-lmysqlclient). In fact, the design of the server library is such that if you write an application to use it, you can easily produce either an embedded or a client/server version of the application simply by linking in the appropriate library. This works because the regular client library contains interface functions that have the same calling sequence as the embedded server calls but are stubs (dummy routines) that do nothing.

Writing an Embedded Server Application

Writing an application that uses the embedded server is little different than writing one that operates in a client/server context. In fact, if you begin with a program that is written as a client/server application, you can convert it easily to use the embedded server instead. For example, to modify client4 to produce an embedded application named embapp, copy client4.c to embapp.c and then perform the following steps to modify embapp.c:

1. Add mysql_embed.h to the set of MySQL header files used by the program:

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

2. An embedded application includes both a client side and a server side, so it can process one group of options for the client and another group for the server. For example, an application named embapp might read the [client] and [embapp] groups from option files for the client part. To set that up, modify the definition of the client_groups array to look like this:

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

   Options in these groups can be processed by load_defaults() and handle_options() in the usual fashion. Then define another list of option groups for the server side to use. By convention, this list should include the [server] and [embedded] groups and also the [*appname*_SERVER] group, where *appname* is the name of your application. For a program named embapp, the application-specific group will be [embapp_SERVER], so you declare the list of group names as follows:

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

3. Call `mysql_server_init()` before initiating communication with the server. A good place to do this is before you call `mysql_init()`.
4. Call `mysql_server_end()` after you're done using the server. A good place to do this is after you call `mysql_close()`.

After making these changes, the `main()` function in `embapp.c` will look like this:

```
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);

    /* solicit password if necessary */
    if (ask_password)
        opt_password = get_tty_password (NULL);

    /* get database name if present on command line */
    if (argc > 0)
    {
        opt_db_name = argv[0];
        --argc; ++argv;
    }

    /* initialize embedded server */
    mysql_server_init (0, NULL, (char **) server_groups);

    /* initialize connection handler */
    conn = mysql_init (NULL);
    if (conn == NULL)
    {
        print_error (NULL, "mysql_init() failed (probably out of memory)");
        exit (1);
    }

    /* connect to server */
    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> ");                   /* print prompt */
        if (fgets (buf, sizeof (buf), stdin) == NULL)   /* read query */
            break;
        if (strcmp (buf, "quit\n") == 0 || strcmp (buf, "\\q\n") == 0)
            break;
        process_query (conn, buf);                      /* execute query */
    }

    /* disconnect from server */
    mysql_close (conn);
    /* shut down embedded server */
    mysql_server_end ();
    exit (0);
}
```

Producing the Application Executable Binary

To produce the embedded-server executable binary for embapp, link in the -lmysqld library rather than the -lmysqlclient library. The mysql_config utility is useful here. Just as it can show you the flags to use for linking in the regular client library, it also can display the flags necessary for the embedded server:

```
% mysql_config --libmysqld-libs
 -L'/usr/local/mysql/lib/mysql' -lmysqld -lz -lm
```

Thus, to produce an embedded version of embapp, use commands such as these:

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

At this point, you have an embedded application that contains everything you need to access your MySQL databases. However, be sure when you execute embapp that it does not attempt to use the same data directory as any standalone servers that may already be running on the same machine. Also, under UNIX, the application must run with privileges that give it access to the data directory. You can either run embapp while logged in as the user that owns the data directory, or you can make it a setuid program that changes its user ID to that user when it starts up. For example, to set embapp to run with the privileges of a user named mysqladm, issue the following commands as root:

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

Should you decide that you want to produce a non-embedded version of the application that operates in a client/server context, link it against the regular client library. You can do so by building it as follows:

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

The regular client library includes dummy versions of mysql_server_init() and mysql_server_end() that do nothing, so no link errors will occur.

Miscellaneous Topics

This section covers several query-processing subjects that didn't fit very well into earlier sections of this chapter:

- How to use result set data to calculate a result after using result set metadata to help verify that the data are suitable for your calculations
- How to deal with data values that are troublesome to insert into queries
- How to work with binary data
- How to get information about the structure of your tables
- Common MySQL programming mistakes and how to avoid them

Performing Calculations on Result Sets

So far we've concentrated on using result set metadata primarily for printing query rows, but clearly there will be times when you need to do something with a result set besides print it. For example, you can compute statistical information based on the data values, using the metadata to make sure the data conform to requirements you want them to satisfy. What type of requirements? For starters, you'd probably want to verify that a column on which you're planning to perform numeric computations actually contains numbers.

The following listing shows a simple function, summary_stats(), that takes a result set and a column index and produces summary statistics for the values in the column. The function also reports the number of missing values, which it detects by checking for NULL values. These calculations involve two requirements that the data must satisfy, so summary_stats() verifies them using the result set metadata:

- The specified column must exist—that is, the column index must be within range of the number of columns in the result set. This range is from 0 to `mysql_num_fields()`–1.
- The column must contain numeric values.

If these conditions do not hold, `summary_stats()` simply prints an error message and returns. It is implemented as follows:

```
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;

    /* verify data requirements: column must be in range and numeric */
    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;
    }

    /* calculate summary statistics */

    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]);  /* convert string to number */
            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));
    }
}
```

Note the call to `mysql_data_seek()` that precedes the `mysql_fetch_row()` loop. It positions to the first row of the result set, which is useful in case you want to call `summary_stats()` multiple times for the same result set (for example, to calculate statistics on several different columns). The effect is that each time `summary_stats()` is invoked, it "rewinds" to the beginning of the result set. The use of `mysql_data_seek()` requires that you create the result set with `mysql_store_result()`. If you create it with `mysql_use_result()`, you can only process rows in order, and you can process them only once.

`summary_stats()` is a relatively simple function, but it should give you an idea of how you could program more complex calculations, such as a least-squares regression on two columns or standard statistics such as a *t*-test or an analysis of variance.

Encoding Problematic Data in Queries

If inserted literally into a query, data values containing quotes, nulls, or backslashes can cause problems when you try to execute the query. The following discussion describes the nature of the difficulty and how to solve it.

Suppose you want to construct a `SELECT` query based on the contents of the null-terminated string pointed to by the `name_val` variable:

```
char query[1024];

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

If the value of `name_val` is something like `O'Malley, Brian`, the resulting query is illegal because a quote appears inside a quoted string:

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

You need to treat the quote specially so that the server doesn't interpret it as the end of the name. The ANSI SQL convention for doing this is to double the quote within the string. MySQL understands that convention and also allows the quote to be preceded by a backslash, so you can write the query using either of the following formats:

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

Another problematic situation involves the use of arbitrary binary data in a query. This happens, for example, in applications that store images in a database. Because a binary value can contain any character (including quotes or backslashes), it cannot be considered safe to put into a query as is.

To deal with this problem, use `mysql_real_escape_string()`, which encodes special characters to make them usable in quoted strings. Characters that `mysql_real_escape_string()` considers special are the null character, single quote, double quote, backslash, newline, carriage return, and Ctrl-Z. (The last one is special on Windows, where it often signifies end-of-file.)

When should you use `mysql_real_escape_string()`? The safest answer is "always." However, if you're sure of the format of your data and know that it's okay—perhaps because you have performed some prior validation check on it—you need not encode it. For example, if you are working with strings that you know represent legal phone numbers consisting entirely of digits and dashes, you don't need to call `mysql_real_escape_string()`. Otherwise, you probably should.

`mysql_real_escape_string()` encodes problematic characters by turning them into 2-character sequences that begin with a backslash. For example, a null byte becomes '`\0`', where the '`0`' is a printable ASCII zero, not a null. Backslash, single quote, and double quote become '`\\`', '`\'`', and '`\"`'.

To use `mysql_real_escape_string()`, invoke it as follows:

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

`mysql_real_escape_string()` encodes `from_str` and writes the result into `to_str`. It also adds a terminating null, which is convenient because you can use the resulting string with functions such as `strcpy()`, `strlen()`, or `printf()`.

`from_str` points to a `char` buffer containing the string to be encoded. This string can contain anything, including binary data. `to_str` points to an existing `char` buffer where you want the encoded string to be written; do not pass an uninitialized or `NULL` pointer, expecting `mysql_real_escape_string()` to allocate space for you. The length of the buffer pointed to by `to_str` must be at least `(from_len*2)+1` bytes long. (It's possible that every character in from_str will need encoding with two characters; the extra byte is for the terminating null.)

from_len and to_len are unsigned long values. from_len indicates the length of the data in from_str; it's necessary to provide the length because from_str may contain null bytes and cannot be treated as a null-terminated string. to_len, the return value from mysql_real_escape_string(), is the actual length of the resulting encoded string, not counting the terminating null.

When mysql_real_escape_string() returns, the encoded result in to_str can be treated as a null-terminated string because any nulls in from_str are encoded as the printable '\0' sequence.

To rewrite the SELECT-constructing code so that it works even for name values that contain quotes, we could do something like the following:

```
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';
```

Yes, that's ugly. If you want to simplify the code a bit, at the cost of using a second buffer, do the following instead:

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

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

mysql_real_escape_string() is unavailable prior to MySQL 3.23.14. As a workaround, you can use mysql_escape_string() instead:

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

The difference between them is that mysql_real_escape_string() uses the character set for the current connection to perform encoding. mysql_escape_string() uses the default character set (which is why it doesn't take a connection handler argument). To write source that will compile under any version of MySQL, include the following code fragment in your file:

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

Then write your code in terms of `mysql_real_escape_string()`; if that function is unavailable, the `#define` causes it to be mapped to `mysql_escape_string()` instead.

Working with Image Data

One of the jobs for which `mysql_real_escape_string()` is essential involves loading image data into a table. This section shows how to do it. (The discussion applies to any other form of binary data as well.)

Suppose you want to read images from files and store them in a table named `picture` along with a unique identifier. The `BLOB` type is a good choice for binary data, so you could use a table specification like this:

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

To actually get an image from a file into the `picture` table, the following function, `load_image()`, does the job, given an identifier number and a pointer to an open file containing the image data:

```
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)
    {
        /* don't overrun end of query buffer! */
        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()` doesn't allocate a very large query buffer (100KB), so it works only for relatively small images. In a real-world application, you might allocate the buffer dynamically based on the size of the image file.

Getting an image value (or any binary value) back out of a database isn't nearly as much of a problem as putting it in to begin with because the data value is available in raw form in the `MYSQL_ROW` variable, and the length is available by calling `mysql_fetch_lengths()`. Just be sure to treat the value as a counted string, not as a null-terminated string.

Getting Table Information

MySQL allows you to get information about the structure of your tables, using any of the following queries (which are equivalent):

```
SHOW COLUMNS FROM tbl_name;
SHOW FIELDS FROM tbl_name;
DESCRIBE tbl_name;
EXPLAIN tbl_name;
```

Each statement is like `SELECT` in that it returns a result set. To find out about the columns in the table, all you need to do is process the rows in the result to pull out the information you want. For example, if you issue a `DESCRIBE president` statement using the `mysql` client, it returns the following 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       |       |
+------------+-------------+------+-----+------------+-------+
```

If you execute the same query from your own client program, you get the same information (without the boxes).

If you want information only about a single column, add the column name:

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

Client Programming Mistakes to Avoid

This section discusses some common MySQL C API programming errors and how to avoid them. (These problems crop up periodically on the MySQL mailing list; I'm not making them up.)

Mistake 1—Using Uninitialized Connection Handler Pointers

The examples shown earlier in this chapter invoke `mysql_init()` with a `NULL` argument. That tells `mysql_init()` to allocate and initialize a `MYSQL` structure and return a pointer to it. Another approach is to pass a pointer to an existing `MYSQL` structure. In this case, `mysql_init()` will initialize that structure and return a pointer to it without allocating the structure itself. If you want to use this second approach, be aware that it can lead to certain subtle difficulties. The following discussion points out some problems to watch out for.

If you pass a pointer to `mysql_init()`, it must actually point to something. Consider the following piece of code:

```
main ()
{
MYSQL    *conn;

    mysql_init (conn);
    ...
}
```

The problem is that `mysql_init()` receives a pointer, but that pointer doesn't point anywhere sensible. `conn` is a local variable and thus is uninitialized storage that can point anywhere when `main()` begins execution. That means `mysql_init()` will use the pointer and scribble on some random area of memory. If you're lucky, `conn` will point outside your program's address space and the system will terminate it immediately so that you'll realize that the problem occurs early in your code. If you're not so lucky, `conn` will point into some data that you don't use until later in your program, and you won't notice a problem until your program actually tries to use that data. In that case, your problem will appear to occur much farther into the execution of your program than where it actually originates and may be much more difficult to track down.

Here's a similar piece of problematic code:

```
MYSQL    *conn;

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

In this case, conn is a global variable, so it's initialized to 0 (that is, to NULL) before the program starts up. mysql_init() sees a NULL argument, so it initializes and allocates a new connection handler. Unfortunately, the value of conn remains NULL because no value is ever assigned to it. As soon as you pass conn to a MySQL C API function that requires a non-NULL connection handler, your program will crash. The fix for both pieces of code is to make sure conn has a sensible value. For example, you can initialize it to the address of an already-allocated MYSQL structure:

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

However, the recommended (and easier!) solution is simply to pass NULL explicitly to mysql_init(), let that function allocate the MYSQL structure for you, and assign conn the return value:

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

In any case, don't forget to test the return value of mysql_init() to make sure it's not NULL (see Mistake 2).

Mistake 2—Failing to Check Return Values

Remember to check the status of calls that may fail. The following code doesn't do that:

```
MYSQL_RES *res_set;
MYSQL_ROW row;
```

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

Unfortunately, if `mysql_store_result()` fails, `res_set` is `NULL`, in which case the while loop should never even be executed. (Passing `NULL` to `mysql_fetch_row()` likely will crash the program.) Test the return value of functions that return result sets to make sure you actually have something to work with.

The same principle applies to any function that may fail. When the code following a function depends on the success of the function, test its return value and take appropriate action if failure occurs. If you assume success, problems will occur.

Mistake 3—Failing to Account for NULL Column Values

Don't forget to check whether column values in the `MYSQL_ROW` array returned by `mysql_fetch_row()` are `NULL` pointers. The following code crashes on some machines if `row[i]` is `NULL`:

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

The worst part about this mistake is that some versions of `printf()` are forgiving and print "`(null)`" for `NULL` pointers, which allows you to get away with not fixing the problem. If you give your program to a friend who has a less-forgiving `printf()`, the program will crash and your friend will conclude that you're a lousy programmer. The loop should be written as follows instead:

```
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);
```

The only time you need not check whether a column value is `NULL` is when you have already determined from the column's information structure that IS_NOT_NULL() is true.

Mistake 4—Passing Nonsensical Result Buffers

Client library functions that expect you to supply buffers generally want them to really exist. Consider the following example, which violates that principle:

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

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

What's the problem? to_str must point to an existing buffer, and it doesn't—it's not initialized and may point to some random location. Don't pass an uninitialized pointer as the to_str argument to mysql_real_escape_string() unless you want it to stomp merrily all over some random piece of memory.