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 to establish and terminate a session with a server.
- Routines to 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 other C API calls fail.

This chapter shows how to use the client library to write your own programs. Some of the goals we'll keep in mind are consistency with existing client programs in the MySQL distribution, as well as modularity and reusability of the code. 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, they all have one thing in common: They establish a connection to the server.

We'll build the skeleton in steps:

1. Write some bare-bones connection and disconnection code (`client1`).
2. Add error checking (`client2`).
3. Make the connection code modular and reusable (`client3`).
4. Add the ability to get connection parameters (host, user, password) at runtime (`client4`).

This framework is reasonably generic, and you can use it as the basis for any number of 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 hardcoded SQL statements, then develop code that can be used to process arbitrary statements. After that, we'll add our query-processing code to our client framework to develop another program (`client5`) that's similar to the `mysql` client.

We'll also 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 only 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 for downloading so you can try them directly without typing them in yourself. See Appendix A, "Obtaining and Installing Software," for instructions.

Where to Find Examples

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 contains several client programs (`mysql`, `mysqladmin`, and `mysqldump`, for example), most of which are written in C. Because the distribution is readily available in source form, MySQL itself 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 the `client` directory. The MySQL client programs are in the public domain and you may freely borrow code from them for your own programs.

Between the examples provided in this chapter and the client programs included in the MySQL distribution, you may be able to find something similar to what you want to do when writing your own programs. If you do, you may be able to reuse code by copying an existing program and modifying it. You should read this chapter to gain an understanding of how the client library works. Remember, however, that you don't always need to write everything yourself from the ground up. (You'll notice that code reusability is one of the goals in our discussion of writing programs in this chapter.) If you can avoid a lot of work by building on what someone else has already done, so much the better.

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 almost any client program you write.

Basic System Requirements

When you write a MySQL client program in C, you'll need a C compiler, obviously. The examples shown here use `gcc`. You'll also need the following in addition to your own source files:

- The MySQL header files
- The MySQL client library

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

Compiling and Linking the Client

To compile and link a client program, you must specify where the MySQL header files and client library are located because they usually are not installed in locations that the compiler and linker search by default. For the following example, 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, pass it a `-I/usr/local/include/mysql` argument when you compile a source file into an object file. For example, 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. If the link step results in an error having to do with not being able to find the `floor()` function, link in the math library by adding `-lm` to the end of the command:

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

You might need to add other libraries as well. For example, you'll probably need `-lsocket -lnsl` on Solaris.

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`, comprising two source files, `main.c` and `aux.c`, and a header file, `myclient.h`. A simple `Makefile` to build this program might look like this:

```
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
```

If your system is one for which you need to link in the math library, change the value of `LIBS` to add `-lm` to the end:

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

If you need other libraries, such as `-lsocket` and `-lnsl`, add those to `LIBS`, too.

Using the `Makefile`, you can rebuild your program whenever you modify any of the source files simply by typing "`make`". That's easier and less error prone than typing a long `gcc` command.

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 before you can really do anything with a MySQL database, you must be connected to a server. This is such a common operation that code you develop to establish a connection is code you'll use in every client program you write. Besides, this task gives us something simple to start with. We can flesh out the client later to do something more useful.

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

```
/* client1.c */

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

#define def_host_name   NULL /* host to connect to (default = localhost) */
#define def_user_name   NULL /* user name (default = your login name) */
#define def_password    NULL /* password (default = none) */
#define def_db_name     NULL /* database to use (default = none) */

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

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

    conn = mysql_init (NULL);
    mysql_real_connect (
                conn,           /* pointer to connection handler */
                def_host_name,  /* host to connect to */
                def_user_name,  /* user name */
                def_password,   /* password */
                def_db_name,    /* database to use */
                0,              /* port (use default) */
                NULL,           /* socket (use default) */
                0);             /* flags (none) */
    mysql_close (conn);
    exit (0);
}
```

The source file begins by including `stdio.h` and `mysql.h`. MySQL clients may include other header files, but generally these two are the bare minimum.

The defaults for the hostname, username, password, and database name are hard-wired into the code to keep things simple. Later we'll parameterize these values so you can specify them in option files or on the command line.

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. The `MYSQL` data type is a structure containing information about a connection. Variables of this type are called connection handlers. When you pass `NULL` to `mysql_init()`, it allocates a `MYSQL` variable, initializes it, and returns a pointer to it.
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 not be `NULL`; it should be the value returned by `mysql_init()`.

- The server host. If you specify `NULL` or the host `"localhost"`, the client connects to the server running on the local host using a UNIX socket. If you specify a hostname or host IP address, the client connects to the named host using a TCP/IP connection.

  On Windows, the behavior is similar, except that TCP/IP connections are used instead of UNIX sockets. (On Windows NT, the connection is attempted using a named pipe before TCP/IP if the host is `NULL`.)

- The username and password. 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 port number and socket file. These are specified as `0` and `NULL`, to tell the client library to use its default values. By leaving the port and socket unspecified, the defaults are determined according to the host you wish to connect to. The details on this are given in the description of `mysql_real_connect()` in Appendix F.

- The flags value. This is `0` because we aren't using any special connection options. The options that are available for this parameter are discussed in more detail in the entry for `mysql_real_connect()` in Appendix F.

To terminate the connection, pass a pointer to the connection handler to `mysql_close()`. A connection handler that is allocated automatically by `mysql_init()` is de-allocated automatically when you pass it to `mysql_close()` to terminate the connection.

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

```
% client1
```

The 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 you don't really know whether or not it actually works!
- The connection parameters (hostname, username, etc.) are hardwired into the source code. It would be better to allow the user to override them by specifying the parameters in an option file or on the command line.

Neither of these problems is difficult to deal with. We'll address them both in the next few sections.

Client 2—Adding Error Checking

Our second client will be like the first one, but it will be modified to take into account the possibility of errors occurring. 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, I prefer to promote the view that MySQL client programs should test for error conditions and respond to them appropriately. The client library calls that return status values do so for a reason, and you ignore them at your peril. You end up trying to track down obscure problems that occur in your programs due to failure to check for errors, users of your programs wonder why those programs behave erratically, or both.

Consider our program, `client1`. How do you know whether or not 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:

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

Alternatively, you might see an `Access denied` message instead:

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

This message indicates that no connection was established at all. 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. In any case, you certainly shouldn't have to look in the log to find out whether or not you were able to connect to the server! Let's fix that right away.

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 older code that tests a return value 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())            /* this test is correct */
    fprintf (stderr, "something bad happened\n");
```

or like this:

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

The two tests are equivalent. If you look through the source code for MySQL itself, you'll find that generally it uses the first form of the test, which is shorter to write.

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 fails and you need more information about the failure, two calls in the API are useful. `mysql_error()` returns a string containing an error message, and `mysql_errno()` returns a numeric error code. You should call them right after an error occurs because 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. If you report only one of the two, I suggest it be the string. For completeness, the examples in this chapter report both values.

Taking the preceding discussion into account, we'll write our second client, `client2`. It is similar to `client1`, but with proper error-checking code added. The source file, `client2.c`, looks like this:

```
/* client2.c */

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

#define def_host_name   NULL /* host to connect to (default = localhost) */
#define def_user_name   NULL /* user name (default = your login name) */
#define def_password    NULL /* password (default = none) */
#define def_db_name     NULL /* database to use (default = none) */

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

int
main (int argc, char *argv[])
{
    conn = mysql_init (NULL);
    if (conn == NULL)
    {
        fprintf (stderr, "mysql_init() failed (probably out of memory)\n");
        exit (1);
    }
```

```
    if (mysql_real_connect (
                conn,           /* pointer to connection handler */
                def_host_name,  /* host to connect to */
                def_user_name,  /* user name */
                def_password,   /* password */
                def_db_name,    /* database to use */
                0,              /* port (use default) */
                NULL,           /* socket (use default) */
                0)              /* flags (none) */
            == NULL)
    {
        fprintf (stderr, "mysql_real_connect() failed:\nError %u (%s)\n",
                         mysql_errno (conn), mysql_error (conn));
        exit (1);
    }
    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 doesn't reflect a valid connection, but does contain error information that can be passed to the error-reporting functions. (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: 'paul@localhost' (Using password: NO))
```

This output indicates no connection was established, and lets you know why. It also means that our 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. That's why you should always test API function return values.

Answers to questions on the MySQL mailing list often have to do with error checking. 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. Don't make the mistake of assuming that every client library call succeeds.

The rest of the examples in this chapter perform error checking, and you 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 Chapters 7, "The Perl DBI API," and 8, "The PHP API."

Now, suppose you did see an `Access denied` message when you ran the `client2` program. How can you fix the problem? One possibility is to change the `#define` lines for the hostname, username, and password 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 hardcoded 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 form, but it's not hidden at all if someone can run `strings` on the program. (Not to mention the fact that anyone with read access to your source file can get the password with no work at all.)

We'll deal with the access problem in the section "Client 4—Getting Connection Parameters at Runtime." First, I want to show some other ways of writing your connection code.

Client 3—Making the Connection Code Modular

For our third client, `client3`, we will make the connection and disconnection code more modular by encapsulating it into functions `do_connect()` and `do_disconnect()`, which can be used easily by multiple client programs. This provides an alternative to embedding the connection code literally into your `main()` function. That's a good idea anyway for any code that's stereotypical across applications. Put it in a function that you can access from multiple programs rather than writing it out in each one. If you fix a bug in or make an improvement to the function, you can change it once and all the programs that use the function can be fixed or take advantage of the improvement just by being recompiled. Also, some client programs are written such that they may connect and disconnect several times during the course of their execution. It's a lot easier to write such a client if you make the code modular by putting your setup and teardown machinery in connect and disconnect functions.

The encapsulation strategy works like this:

1. Split out common code into wrapper functions in a separate source file, `common.c`.
2. Provide a header file, `common.h`, containing prototypes for the common routines.
3. Include `common.h` in client source files that use the common routines.
4. Compile the common source into an object file.
5. Link that common object file into your client program.

With that strategy in mind, let's construct `do_connect()` and `do_disconnect()`.

`do_connect()` replaces the calls to `mysql_init()` and `mysql_real_connect()`, as well as the error-printing code. You call it just like `mysql_real_connect()`, except that you don't pass any connection handler. Instead, `do_connect()` allocates and initializes the handler itself, then returns a pointer to it after connecting. If `do_connect()` fails, it returns `NULL` after printing an error message. (That way, any program that calls `do_connect()` and gets a `NULL` return value can simply exit without worrying about printing a message itself.)

`do_disconnect()` takes a pointer to the connection handler and calls `mysql_close()`.

Here is the code for `common.c`:

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

MYSQL *
do_connect (char *host_name, char *user_name, char *password, char *db_name,
            unsigned int port_num, char *socket_name, unsigned int flags)
{
MYSQL   *conn;  /* pointer to connection handler */

    conn = mysql_init (NULL);   /* allocate, initialize connection handler */
    if (conn == NULL)
    {
        fprintf (stderr, "mysql_init() failed\n");
        return (NULL);
    }
    if (mysql_real_connect (conn, host_name, user_name, password,
                        db_name, port_num, socket_name, flags) == NULL)
    {
        fprintf (stderr, "mysql_real_connect() failed:\nError %u (%s)\n",
                           mysql_errno (conn), mysql_error (conn));
        return (NULL);
    }
    return (conn);          /* connection is established */
}

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

`common.h` declares the prototypes for the routines in `common.c`:

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

To access the common routines, include `common.h` in your source files. Note that `common.c` includes `common.h` as well. That way, you get a compiler warning immediately if the function definitions in `common.c` don't match the declarations in the header file. Also, if you change a calling sequence in `common.c` without making the corresponding change to `common.h`, the compiler will warn you when you recompile `common.c`.

It's reasonable to ask why anyone would invent a wrapper function, `do_disconnect()`, that does so little. It's true that `do_disconnect()` and `mysql_close()` are equivalent. But suppose sometime down the road you decide there is some additional cleanup you'd like to perform whenever you disconnect. By calling a wrapper function that you have complete control over, you can modify the wrapper to do what you like and the change takes effect uniformly for any disconnect operation you do. You can't do this if you invoke `mysql_close()` directly.

Earlier, I asserted that it's beneficial to modularize commonly used code by encapsulating it in a function that can be used by multiple programs, or from multiple places within a single program. The preceding paragraph gives one reason why, and the following two examples provide some additional justification.

- **Example 1.** In versions of MySQL prior to the 3.22 series, the `mysql_real_connect()` call was slightly different than it is now: There was no database name parameter. If you want to use `do_connect()` with an older MySQL client library, it won't work. However, it's possible to modify `do_connect()` so that it will work on pre-3.22 installations. This means that by modifying `do_connect()`, you can increase the portability of all programs that use it. If you embed the connect code literally in every client, you must modify each of them individually.

  To fix `do_connect()` so that it can deal with the older form of `mysql_real_connect()`, use the `MYSQL_VERSION_ID` macro that contains the current MySQL version number. The modified `do_connect()` tests the value of `MYSQL_VERSION_ID` and uses the proper form of `mysql_real_connect()`:

```
MYSQL *
do_connect (char *host_name, char *user_name, char *password, char *db_name,
            unsigned int port_num, char *socket_name, unsigned int flags)
{
MYSQL   *conn;  /* pointer to connection handler */

    conn = mysql_init (NULL);   /* allocate, initialize connection handler */
    if (conn == NULL)
    {
        fprintf (stderr, "mysql_init() failed\n");
        return (NULL);
    }
#if defined(MYSQL_VERSION_ID) && MYSQL_VERSION_ID >= 32200  /* 3.22 and up */
    if (mysql_real_connect (conn, host_name, user_name, password,
```

```
                    db_name, port_num, socket_name, flags) == NULL)
    {
        fprintf (stderr, "mysql_real_connect() failed:\nError %u (%s)\n",
                        mysql_errno (conn), mysql_error (conn));
        return (NULL);
    }
#else                           /* pre-3.22 */
    if (mysql_real_connect (conn, host_name, user_name, password,
                    port_num, socket_name, flags) == NULL)
    {
        fprintf (stderr, "mysql_real_connect() failed:\nError %u (%s)\n",
                        mysql_errno (conn), mysql_error (conn));
        return (NULL);
    }
    if (db_name != NULL)        /* simulate effect of db_name parameter */
    {
        if (mysql_select_db (conn, db_name) != 0)
        {
            fprintf (stderr, "mysql_select_db() failed:\nError %u (%s)\n",
                            mysql_errno (conn), mysql_error (conn));
            mysql_close (conn);
            return (NULL);
        }
    }
#endif
    return (conn);          /* connection is established */
}
```

The modified version of `do_connect()` is identical to the previous version in all respects except two:

- It doesn't pass a `db_name` parameter to the older form of `mysql_real_connect()` because that version has no such parameter.
- If the database name is non-`NULL`, `do_connect()` calls `mysql_select_db()` to make the named database current. (This simulates the effect of the missing `db_name` parameter). If the database cannot be selected, `do_connect()` prints an error message, closes the connection, and returns `NULL` to indicate failure.

- **Example 2.** This example builds on the changes made to `do_connect()` for the first example. Those changes result in three sets of calls to the error functions `mysql_errno()` and `mysql_error()`, and it's really tiresome write those out each time the code needs to squawk about a problem. Besides, the error printing code is visually noisy and difficult to read. It's easier to read something like this:

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

So let's encapsulate error printing in a `print_error()` function. We can write it to do something sensible even if `conn` is `NULL`. That way, we can use `print_error()` if the `mysql_init()` call fails, and we don't have a mix of calls (some to `fprintf()` and some to `print_error()`).

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 making your code difficult to read on purpose just so your encapsulation example looks better. And you wouldn't really write out all that error-printing code anyway; you'd write it once, then use copy and paste when you need it again." Those are valid points, but I would address the objections like this:

- 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. Putting the error-reporting code in a wrapper 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.

Here's our error-reporting function `print_error()`:

```
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));
    }
}
```

`print_error()` is in `common.c`, so we add a prototype for it to `common.h`:

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

Now `do_connect()` can be modified to use `print_error()`:

```
MYSQL *
do_connect (char *host_name, char *user_name, char *password, char *db_name,
            unsigned int port_num, char *socket_name, unsigned int flags)
{
MYSQL   *conn;  /* pointer to connection handler */

    conn = mysql_init (NULL);  /* allocate, initialize connection handler */
    if (conn == NULL)
    {
        print_error (NULL, "mysql_init() failed (probably out of memory)");
        return (NULL);
    }
#if defined(MYSQL_VERSION_ID) && MYSQL_VERSION_ID >= 32200  /* 3.22 and up
*/
    if (mysql_real_connect (conn, host_name, user_name, password,
                    db_name, port_num, socket_name, flags) == NULL)
    {
        print_error (conn, "mysql_real_connect() failed");
        return (NULL);
    }
#else                           /* pre-3.22 */
    if (mysql_real_connect (conn, host_name, user_name, password,
                    port_num, socket_name, flags) == NULL)
    {
        print_error (conn, "mysql_real_connect() failed");
        return (NULL);
    }
    if (db_name != NULL)        /* simulate effect of db_name parameter */
    {
        if (mysql_select_db (conn, db_name) != 0)
        {
            print_error (conn, "mysql_select_db() failed");
            mysql_close (conn);
            return (NULL);
        }
    }
#endif
    return (conn);          /* connection is established */
}
```

Our main source file, `client3.c`, is like `client2.c`, but has all the embedded connect and disconnect code removed and replaced with calls to the wrapper functions. It looks like this:

```
/* client3.c */

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

#define def_host_name   NULL /* host to connect to (default = localhost) */
#define def_user_name   NULL /* user name (default = your login name) */
#define def_password    NULL /* password (default = none) */
#define def_port_num    0    /* use default port */
#define def_socket_name NULL /* use default socket name */
#define def_db_name     NULL /* database to use (default = none) */

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

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

    /* do the real work here */

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

Client 4—Getting Connection Parameters at Runtime

Okay, now that we have our easily modifiable and bullet-proof-in-case-an-error-occurs connection code, we're ready to figure out how to do something smarter than using `NULL` connection parameters—like letting the user specify those values at runtime.

The previous client, `client3`, still has a significant shortcoming in that the connection parameters are hardwired in. 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. The programs in the MySQL distribution accept connection parameters in either of two forms, as shown in Table 6.1.

Table 6.1 **Standard MySQL Command-Line Options**

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

For consistency with the standard MySQL clients, our client will accept those same formats. It's easy to do this because the client library includes a function to perform option parsing.

In addition, our client will have the ability to extract information from option files. This allows you to put connection parameters in `~/.my.cnf` (that is, the `.my.cnf` file in your home directory) so that you don't have to specify them on the command line. 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."

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. Therefore, when you parse the command options, you get the connection parameters as part of your normal option-parsing loop. The options are added to the beginning of `argv[]` rather than at the end so that if connection parameters really are specified on the command line, they occur later than (and thus override) any options added by `load_defaults()`.

Here's a little program, `show_argv`, that shows how to use `load_defaults()` and illustrates how doing so modifies your argument vector:

```
/* show_argv.c */

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

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

int
```

continues

continued

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

    my_init ();

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

    load_defaults ("my", 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 the following:

- `groups[]` is a character string array indicating which option file groups you are interested in. For client programs, you always specify as least `"client"` (for the `[client]` group). The last element of the array must be `NULL`.
- `my_init()` is an initialization routine that performs some setup operations required by `load_defaults()`.
- `load_defaults()` takes four arguments: the prefix of your option files (this should always be `"my"`), the array listing 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 although `argv` is a pointer, you still pass `&argv`, that pointer's address.

`show_argv` prints its arguments twice—first as you specified them on the command line, then as they were modified by `load_defaults()`. To see the effect of `load_defaults()`, make sure you have a `.my.cnf` file in your home directory with some settings specified for the `[client]` group. Suppose `.my.cnf` looks like this:

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

If this is the case, then executing `show_argv` produces 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=paul
arg 2: --password=secret
arg 3: --host=some_host
arg 4: a
arg 5: b
```

It's possible that you'll see some options in the output from `show_argv` that were not on the command line or in your `~/.my.cnf` file. If so, they were probably specified in a system-wide option file. `load_defaults()` actually looks for `/etc/my.cnf` and the `my.cnf` file in the MySQL data directory before reading `.my.cnf` in your home directory. (On Windows, `load_defaults()` searches for `C:\my.cnf`, `C:\mysql\data\my.cnf`, and the `my.ini` file in your Windows system directory.)

Client programs that use `load_defaults()` almost always specify `"client"` in the options group list (so that they get any general client settings from option files), but you can also ask for values that are specific to your own program. Just change the following:

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

to this:

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

Then you can add a `[show_argv]` group to your `~/.my.cnf` file:

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

[show_argv]
host=other_host
```

With these changes, invoking `show_argv` again has a different result, as follows:

```
% 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=paul
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 your option groups are listed in the `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. 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 manage that yourself using `getenv()`. I'm not going to add that capability to our clients, but here's an example showing how to check the values of a couple of the standard MySQL-related environment variables:

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

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

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

Parsing Command-Line Arguments

We can get all the connection parameters into the argument vector now, but we need a way to parse the vector. The `getopt_long()` function is designed for this.

`getopt_long()` is built into the MySQL client library, so you have access to it whenever you link in that library. In your source file, you need to include the `getopt.h` header file. You can copy this header file from the `include` directory of the MySQL source distribution into the directory where you're developing your client program.

> **`load_defaults()` and Security**
>
> You may be wondering about the process-snooping implications of having `load_defaults()` putting the text of passwords in your argument list because programs such as `ps` can display argument lists for arbitrary processes. There is no problem because `ps` displays the original `argv[]` contents. Any password argument created by `load_defaults()` points to an area 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 specified on the command line *does* show up in `ps`, unless you take care to wipe it out. The section "Parsing Command-Line Arguments" shows how to do that.

The following program, `show_param`, uses `load_defaults()` to read option files, then calls `getopt_long()` to parse the argument vector. `show_param` illustrates what happens at each phase of argument processing by performing the following actions:

1. Sets up default values for the hostname, username, and password.
2. Prints the original connection parameter and argument vector values.
3. Calls `load_defaults()` to rewrite the argument vector to reflect option file contents, then prints the resulting vector.
4. Calls `getopt_long()` to process the argument vector, then prints the resulting parameter values and whatever is left in the argument vector.

`show_param` 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. `show_param` is useful for getting a feel for what will happen in our next client program, when we actually hook up this parameter-processing code to our connection function, `do_connect()`.

Here's what `show_param.c` looks like:

```
/* show_param.c */

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

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

struct option long_options[] =
{
    {"host",     required_argument, NULL, 'h'},
    {"user",     required_argument, NULL, 'u'},
    {"password", optional_argument, NULL, 'p'},
    {"port",     required_argument, NULL, 'P'},
    {"socket",   required_argument, NULL, 'S'},
    { 0, 0, 0, 0 }
};

int
main (int argc, char *argv[])
{
char *host_name = NULL;
char *user_name = NULL;
char *password = NULL;
unsigned int port_num = 0;
char *socket_name = NULL;
int i;
int c, option_index;
```

continues

continued

```
my_init ();

printf ("Original connection parameters:\n");
printf ("host name: %s\n", host_name ? host_name : "(null)");
printf ("user name: %s\n", user_name ? user_name : "(null)");
printf ("password: %s\n", password ? password : "(null)");
printf ("port number: %u\n", port_num);
printf ("socket name: %s\n", socket_name ? socket_name : "(null)");

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

load_defaults ("my", 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]);

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

argc -= optind; /* advance past the arguments that were processed */
argv += optind; /* by getopt_long() */

printf ("Connection parameters after getopt_long():\n");
printf ("host name: %s\n", host_name ? host_name : "(null)");
printf ("user name: %s\n", user_name ? user_name : "(null)");
printf ("password: %s\n", password ? password : "(null)");
printf ("port number: %u\n", port_num);
printf ("socket name: %s\n", socket_name ? socket_name : "(null)");
```

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

    exit (0);
}
```

To process the argument vector, `show_argv` uses `getopt_long()`, which you typically call in a loop:

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

The first two arguments to `getopt_long()` are your program's argument count and vector. The third argument lists the option letters you want to recognize. These are the short-name forms of your program's options. Option letters may be followed by a colon, a double colon, or no colon to indicate that the option must be followed, may be followed, or is not followed by an option value. The fourth argument, `long_options`, is a pointer to an array of option structures, each of which specifies information for an option you want your program to understand. Its purpose is similar to the options string in the third argument. The four elements of each `long_options[]` structure are as follows:

- **The option's long name.**
- **A value for the option.** The value can be `required_argument`, `optional_argument`, or `no_argument` indicating whether the option must be followed, may be followed, or is not followed by an option value. (These serve the same purpose as the colon, double colon, or no colon in the options string third argument.)
- **A flag argument.** You can use this to store a pointer to a variable. If the option is found, `getopt_long()` stores the value specified by the fourth argument into the variable. If the flag is `NULL`, `getopt_long()` instead sets the `optarg` variable to point to any value following the option, and returns the option's short name. Our `long_options[]` array specifies `NULL` for all options. That way, `getopt_long()` returns each argument as it is encountered so that we can process it in the `switch` statement.
- **The option's short (single-character) name.** The short names specified in the `long_options[]` array *must* match the letters used in the options string that you pass as the third argument to `getopt_long()` or your program will not process command-line arguments properly.

The `long_options[]` array must be terminated by a structure with all elements set to `0`.

The fifth argument to `getopt_long()` is a pointer to an `int` variable. `getopt_long()` stores into this variable the index of the `long_options[]` structure that corresponds to the option last encountered. (`show_param` doesn't do anything with this value.)

Note that the password option (specified as `--password` or as `-p`) may take an optional value. That is, you may specify it 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. The optional nature of the password value is indicated by the double colon after the "`p`" in the options string, and by `optional_argument` in the `long_options[]` array. MySQL clients typically allow you to omit the password value on the command line, then prompt you for it. This allows you to avoid giving the password on the command line, which keeps people from seeing your password via process snooping. When we write our next client, `client4`, we'll add this password-checking behavior to it.

Here is a sample invocation of `show_param` and the resulting output (assuming that `~/.my.cnf` still has the same contents as for the `show_argv` example):

```
% show_param -h yet_another_host x
Original connection parameters:
host name: (null)
user name: (null)
password: (null)
port number: 0
socket name: (null)
Original argument vector:
arg 0: show_param
arg 1: -h
arg 2: yet_another_host
arg 3: x
Modified argument vector after load_defaults():
arg 0: show_param
arg 1: --user=paul
arg 2: --password=secret
arg 3: --host=some_host
arg 4: -h
arg 5: yet_another_host
arg 6: x
Connection parameters after getopt_long():
host name: yet_another_host
user name: paul
password: secret
port number: 0
socket name: (null)
Argument vector after getopt_long():
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. `getopt_long()` correctly parses options whether specified in short-option form (`-h host_name`) or in long-option form (`--user=paul`, `--password=secret`).

Now let's strip out 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, `client4.c`, looks like this:

```
/* client4.c */

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

#define def_host_name   NULL /* host to connect to (default = localhost) */
#define def_user_name   NULL /* user name (default = your login name) */
#define def_password    NULL /* password (default = none) */
#define def_port_num    0    /* use default port */
#define def_socket_name NULL /* use default socket name */
#define def_db_name     NULL /* database to use (default = none) */

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

struct option long_options[] =
{
    {"host",     required_argument, NULL, 'h'},
    {"user",     required_argument, NULL, 'u'},
    {"password", optional_argument, NULL, 'p'},
    {"port",     required_argument, NULL, 'P'},
    {"socket",   required_argument, NULL, 'S'},
    { 0, 0, 0, 0 }
};

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

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

    my_init ();
```

continues

continued

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

while ((c = getopt_long (argc, argv, "h:p::u:P:S:", long_options,
                &option_index)) != EOF)
{
    switch (c)
    {
    case 'h':
        host_name = optarg;
        break;
    case 'u':
        user_name = optarg;
        break;
    case 'p':
        if (!optarg)        /* no value given */
            ask_password = 1;
        else                /* copy password, wipe out original */
        {
                (void) strncpy (passbuf, optarg, sizeof(passbuf)-1);
                passbuf[sizeof(passbuf)-1] = '\0';
                password = passbuf;
                while (*optarg)
                    *optarg++ = ' ';
        }
        break;
    case 'P':
        port_num = (unsigned int) atoi (optarg);
        break;
    case 'S':
        socket_name = optarg;
        break;
    }
}

argc -= optind; /* advance past the arguments that were processed */
argv += optind; /* by getopt_long() */

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

if (ask_password)
    password = get_tty_password (NULL);

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

```
    /* do the real work here */

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

Compared to the programs `client1`, `client2`, and `client3` that we developed earlier, `client4` does a few things we haven't seen before:

- It allows the database name to be specified on the command line, following the options that are parsed by `getopt_long()`. This is consistent with the behavior of the standard clients in the MySQL distribution.
- It wipes out any password value in the argument vector after making a copy of it. This is to minimize 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 *minimized*, not eliminated. Specifying passwords on the command line still is a security risk.)
- If a password option was given without a value, the client prompts the user for a password using `get_tty_password()`. This is a utility routine in the client library that prompts for a password without echoing it on the screen. (The client library is full of goodies like this. It's instructive to read through the source of the MySQL client programs because you find out about these routines and how to use them.) 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.

`client4` responds according to the options you specify. Assume there is no option file to complicate matters. If you invoke `client4` with no arguments, it connects to `localhost` and passes your UNIX login name and no password to the server. If instead you invoke `client4`, as shown here, then it prompts for a password (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 in:

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

`client4` also passes the database name `some_db` to `do_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.

Earlier, we went on a code-encapsulation binge, creating wrapper functions for disconnecting to and disconnecting from the server. It's reasonable to ask whether or not to put option-parsing stuff in a wrapper function, too. That's possible, I suppose, but I'm not going to do it. Option-parsing code isn't as consistent across programs as connection code: Programs often support other options in addition to the standard ones we've just looked for, and different programs are likely to support different sets of additional options. That makes it difficult to write a function that standardizes the option-processing loop. Also, unlike connection establishment, which a program may

wish to do multiple times during the course of its execution (and thus is a good candidate for encapsulation), option parsing is typically done just once at the beginning of the program.

The work we've done so far accomplishes something that's necessary for every MySQL client: connecting to the server using appropriate parameters. You need to know how to connect, of course. But now you do know how, and the details of that process are implemented by the client skeleton (`client4.c`), so you no longer need to think about them. That means you can concentrate 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 `do_connect()` and `do_disconnect()` 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 this:

1. Make a copy of `client4.c`.
2. Modify the option-processing loop, if you accept additional options other than the standard ones that `client4.c` knows about.
3. Add your own application-specific code between the connect and disconnect calls.

And you're done.

The point of going through the discipline of constructing the client program skeleton was to come up with something that you can use easily to set up and tear down a connection so that you could focus on what you really want to do. Now you're free to do that, demonstrating the principle that from discipline comes freedom.

Processing Queries

Now that we know how to begin and end a conversation with the server, it's time to see how to conduct the conversation while it's going on. 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 or not it contains binary data.
2. **Issue the query by sending it to the server for execution.**
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 a counted

string, its contents may be anything, including binary data or null bytes. The query is not treated as a null-terminated string.

The other query-issuing function, `mysql_query()`, is more restrictive in what it allows in the query string but often is easier to use. Queries that you pass to `mysql_query()` should be null-terminated strings, which means they cannot contain null bytes in the text of the query. (The presence of null bytes within the query causes 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. You do 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 "Encoding Problematic Data in Queries."

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 the data referenced by the query.

Queries may be grouped into two broad categories: those that do not return a result and those that do. Queries for statements such as `INSERT`, `DELETE`, and `UPDATE` fall into the "no result 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 returned" category; after all, the purpose of issuing those statements is to get something back. The set of rows produced by a query that returns data is called the result set. This is represented in MySQL by the MYSQL_RES data type, 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). An empty result set (that is, one that contains zero rows) is distinct from "no result."

Handling Queries That Return No Result Set

To process a query that does not return a result set, issue the query with mysql_query() or mysql_real_query(). If the query succeeds, you can find 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 ("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 consideration 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_rows_affected() 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 get this behavior by asking for it when you connect to the server. Pass a flags value of CLIENT_FOUND_ROWS to mysql_real_connect(). You can pass CLIENT_FOUND_ROWS as the flags argument to do_connect(), too; it will pass along the value 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. `SHOW`, `DESCRIBE`, and `EXPLAIN` do so as well. For all of these statements, you must perform additional row-handling processing after you issue the query.

Handling a result set involves these steps:

- **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 returns the rows from the server immediately and stores them in the client.
- **Call `mysql_fetch_row()` for each row of the result set.** This function returns a `MYSQL_ROW` value, which 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. `mysql_fetch_row()` returns `NULL` when there are no more rows left in the result set.
- **When you are done with the result set, call `mysql_free_result()` to deallocate 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 mysql") != 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);
    }
}
```

We cheated a little here by calling a function `process_result_set()` to handle each row. We haven't defined that function yet, so we need to do so. Generally, result set-handling functions are based on a loop that looks like this:

```
MYSQL_ROW row;

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

The `MYSQL_ROW` return value from `mysql_fetch_row()` is a pointer to an array of values, so accessing each value is simply a matter of accessing `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 variables of that type as `MYSQL_ROW row`, not as `MYSQL_ROW *row`.
- The strings in a `MYSQL_ROW` array are null-terminated. However, if a column may contain binary data, it may 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.
- Values for all data types, even numeric types, are returned as strings. If you want to treat a value as a number, you must convert the string yourself.
- `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 or not values for that column are `NULL` pointers.

Your applications can do whatever they like with the contents of each row. For purposes of illustration, let's just print the rows with column values separated by tabs. To do that, we need an additional function, `mysql_num_fields()`, from the client library; this function tells us how many values (columns) the row contains.

Here's 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()` prints each row in tab-delimited format (displaying `NULL` values as the word "`NULL`"), followed by 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 cast that value to `unsigned long` and use a '`%lu`' format to print it.

The row-fetching loop is followed by an error test. 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. The test simply allows `process_result_set()` to detect errors, no matter how you create your result set.

This version of `process_result_set()` takes a rather minimalist approach to printing column values, an approach that has certain shortcomings. For example, suppose you execute this query:

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

You will receive the following output:

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

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 section "Using Result Set Metadata."

Printing Binary Data

Column values that contain 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. You could use `fwrite()` or `putc()`, for example.

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 don't always know what kind of statement the query represents. For example, if you execute a query that you read from the keyboard or from a file, it might be any arbitrary statement. You won't know ahead of time whether or not to expect it to return rows. 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 look at the first word because the query 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.

Before writing the code for the query handler, let's outline how this works:

- Issue the query. If it fails, we're done.
- If the query succeeds, call `mysql_store_result()` to retrieve the rows from the server and create a result set.
- If `mysql_store_result()` fails, it could be that the query does not return a result set, or that 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 value, as follows:
    - If `mysql_field_count()` is non-zero, it indicates an error: 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 too large and memory allocation failed, or a network outage between the client and the server may have occurred while fetching rows.

      A slight complication to this procedure is that `mysql_field_count()` doesn't exist prior to MySQL 3.22.24. In earlier versions, you use `mysql_num_fields()` instead. To write programs that work with any version of MySQL, include the following code fragment in any file that calls `mysql_field_count()`:

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

      This causes any calls to `mysql_field_count()` to be treated as calls to `mysql_num_fields()` for versions of MySQL earlier than 3.22.24.

    - If `mysql_field_count()` returns `0`, it means the query returned no result set. (This indicates the query was a statement such as `INSERT`, `DELETE`, or `UPDATE`).
- If `mysql_store_result()` succeeds, the query returned a result set. Process the rows by calling `mysql_fetch_row()` until it returns `NULL`.

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

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

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

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

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

    res_set = mysql_store_result (conn);
    if (res_set == NULL)    /* no result set was returned */
    {
        /*
         * does the lack of a result set mean that an error
         * occurred or that no result set was returned?
         */
        if (mysql_field_count (conn) > 0)
        {
            /*
             * a result set was expected, but mysql_store_result()
             * did not return one; this means an error occurred
             */
            print_error (conn, "Problem processing result set");
        }
        else
        {
            /*
             * no result set was returned; query returned no data
             * (it was not a SELECT, SHOW, DESCRIBE, or EXPLAIN),
             * so just report number of rows affected by query
             */
            printf ("%lu rows affected\n",
                        (unsigned long) mysql_affected_rows (conn));
```

continues

continued

```
            }
        }
        else    /* a result set was returned */
        {
            /* process rows, then free the result set */
            process_result_set (conn, res_set);
            mysql_free_result (res_set);
        }
    }
```

Alternative Approaches to Query Processing

The version of `process_query()` just shown has these 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()` 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()`. Here 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, "process_real_query () failed");
        return;
    }

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

    res_set = mysql_use_result (conn);
    if (res_set == NULL)    /* no result set was returned */
    {
```

```
        /*
         * does the lack of a result set mean that an error
         * occurred or that no result set was returned?
         */
        if (mysql_errno (conn) != 0) /* an error occurred */
            print_error (conn, "Problem processing result set");
        else
        {
            /*
             * no result set was returned; query returned no data
             * (it was not a SELECT, SHOW, DESCRIBE, or EXPLAIN),
             * so just report number of rows affected by query
             */
            printf ("%lu rows affected\n",
                        (unsigned long) mysql_affected_rows (conn));
        }
    }
    else    /* a result set was returned */
    {
        /* process rows, then free the result set */
         process_result_set (conn, res_set);
         mysql_free_result (res_set);
     }
}
```

A Comparison of `mysql_store_result()` and `mysql_use_result()`

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. Instead, it assumes you will call `mysql_fetch_row()` to retrieve the records later. 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. 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's

also possible 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 at once, 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 are 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 can obtain 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, and `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. Otherwise, any remaining records

in the set become part of the next query's result set and an "out of sync" error occurs. This does not happen with `mysql_store_result()` because when that function returns, all rows have already been fetched. In fact, with `mysql_store_result()`, you need not call `mysql_fetch_row()` yourself at all. This can be useful for queries for which all that you're interested in is whether you got a non-empty result, not what the result contains. For example, to find out whether or not a table `my_tbl` exists, you can execute this query:

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

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. (You still need to call `mysql_free_result()`, of course.)

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 a 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:

- The column name and width information is useful for producing nicely formatted output that has column titles and 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 number of rows or columns in 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 may contain binary data, and so forth.

Earlier, in the section "Handling Queries That Return Data," 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 output like this:

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

Let's make some changes to `process_result_set()` to produce tabular output by titling and "boxing" each column. The revised 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 W.    | Milton      | MA    |
| Carter     | James E., Jr.| Plains      | GA    |
| Cleveland  | Grover       | Caldwell    | NJ    |
| ...        | ...          | ...         | ...   |
+------------+--------------+-------------+-------+
```

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. To display output as just described, we need to know quite a number of things about the result set other than just the values of the data contained in the 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 the revised `process_result_set()`. They're not the same, and you may find it instructive to compare two approaches to the same problem.

First, we need 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 they make no reference whatsoever to the row values:

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

/* determine column display widths */
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 */
}
```

Column widths are calculated by iterating through the `MYSQL_FIELD` structures for the columns in the result set. We position to the first structure by calling `mysql_fetch_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 int` 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()`.[1]

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

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

The final code to display the result set looks like this. Note that because we're printing lines of dashes multiple times, code to do that is encapsulated into its own function, `print_dashes()`:

```
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);
}
```

[1] The `length` member of the `MYSQL_FIELD` structure tells you the maximum length that column values can be. This may be a useful workaround if you're using `mysql_use_result()` rather than `mysql_store_result()`.

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

    /* determine column display widths */
    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 |", field->max_length, field->name);
    }
    fputc ('\n', stdout);
    print_dashes (res_set);

    while ((row = mysql_fetch_row (res_set)) != NULL)
    {
        mysql_field_seek (res_set, 0);
        fputc ('|', stdout);
        for (i = 0; i < mysql_num_fields (res_set); i++)
        {
            field = mysql_fetch_field (res_set);
            if (row[i] == NULL)
                printf (" %-*s |", field->max_length, "NULL");
            else if (IS_NUM (field->type))
                printf (" %*s |", field->max_length, row[i]);
            else
                printf (" %-*s |", field->max_length, row[i]);
        }
        fputc ('\n', stdout);
    }
    print_dashes (res_set);
    printf ("%lu 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 5—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. It 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.

`client5` will be similar in some ways to `mysql`, although of course not with as many features. There are several restrictions on what `client5` will allow as input:

- Each input line must contain a single complete query.
- Queries should not be terminated by a semicolon or by '`\g`'.
- Commands such as `quit` are not recognized; instead, use Control-D to terminate the program.

It turns out that `client5` is almost completely trivial to write (fewer than 10 lines of new code). Almost everything we need is provided by our client program skeleton (`client4.c`) and by other code that we have written already. The only thing we need to add is a loop that collects input lines and executes them.

To construct `client5`, begin by copying the client skeleton `client4.c` to `client5.c`. Then add to that the code for `process_query()`, `process_result_set()`, and `print_dashes()`. Finally, in `client5.c`, look for the line in `main()` that says the following:

```
/* do the real work here */
```

Then replace it with this while loop:

```
while (1)
{
char    buf[1024];
    fprintf (stderr, "query> ");                    /* print prompt */
    if (fgets (buf, sizeof (buf), stdin) == NULL)    /* read query */
```

```
        break;
    process_query (conn, buf);                      /* execute query */
}
```

Compile `client5.c` to produce `client5.o`, link `client5.o` with `common.o` and the client library to produce `client5`, and you're done! You have an interactive MySQL client program that can execute any query and display the results.

Miscellaneous Topics

This section covers several subjects that didn't fit very well into the progression as we went from `client1` to `client5`:

- Using 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 that are troublesome to insert into queries.
- How to work with image 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 row data, but clearly there will be times when you need to do something with your data 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).
- The column must contain numeric values.

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

```
void
summary_stats (MYSQL_RES *res_set, unsigned int col_num)
{
```

continues

continued

```
    MYSQL_FIELD     *field;
    MYSQL_ROW       row;
    unsigned int    n, missing;
    double  val, sum, sum_squares, var;

        /* verify data requirements */
        if (mysql_num_fields (res_set) < col_num)
        {
            print_error (NULL, "illegal column number");
            return;
        }
        mysql_field_seek (res_set, 0);
        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's there to allow you to call `summary_stats()` multiple times for the same result set (in case you want to calculate statistics on several columns). Each time `summary_stats()` is invoked, it "rewinds" to the beginning of the result set. (This assumes 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.

Encoding Problematic Data in Queries

Data values containing quotes, nulls, or backslashes, if inserted literally into a query, 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 `name`:

```
char query[1024];

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

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

```
SELECT * FROM my_tbl 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. One way to do this is to double the quote within the string. That is the ANSI SQL convention. MySQL understands that convention, and also allows the quote to be preceded by a backslash:

```
SELECT * FROM my_tbl WHERE name='O''Malley, Brian'
SELECT * FROM my_tbl 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 may contain any character, it cannot be considered safe to put into a query as is.

To deal with this problem, use `mysql_escape_string()`, which encodes special characters to make them usable in quoted strings. Characters that `mysql_escape_string()` considers special are the null character, single quote, double quote, backslash, newline, carriage return, and Control-Z. (The last one occurs in Windows contexts.)

When should you use `mysql_escape_string()`? The safest answer is "always." However, if you're sure of the form 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_escape_string()`. Otherwise, you probably should.

`mysql_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_escape_string()`, invoke it like this:

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

`mysql_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()` and `strlen()`.

`from_str` points to a `char` buffer containing the string to be encoded. This string may 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_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 2 characters; the extra byte is for the terminating null.)

`from_len` and `to_len` are `unsigned int` 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_escape_string()`, is the actual length of the resulting encoded string, not counting the terminating null.

When `mysql_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 values of names that contain quotes, we could do something like this:

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

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

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

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

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

Working With Image Data

One of the jobs for which `mysql_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, 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 images
(
    image_id INT NOT NULL PRIMARY KEY,
    image_data BLOB
)
```

To actually get an image from a file into the `images` 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 int    from_len;
int             status;

    sprintf (query, "INSERT INTO images VALUES (%d,'", id);
    p = query + strlen (query);
    while ((from_len = fread (buf, 1, sizeof (buf), f)) > 0)
    {
        /* 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_escape_string (p, buf, from_len);
    }
    (void) strcpy (p, "')");
    status = mysql_query (conn, query);
    return (status);
}
```

`load_image()` doesn't allocate a very large query buffer (100K), 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.

Handling image data (or any binary data) that you get back out of a database isn't nearly as much of a problem as putting it in to begin with because the data values are available in raw form in the `MYSQL_ROW` variable, and the lengths are available by calling `mysql_fetch_lengths()`. Just be sure to treat the values as counted strings, not as null-terminated strings.

Getting Table Information

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

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

Both statements are like `SELECT` in that they return 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 images` statement from the `mysql` client, it returns this information:

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

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

If you want information only about a single column, use this query instead:

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

The query will return the same columns, but only one row (or no rows if the column doesn't exist).

Client Programming Mistakes To Avoid

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

Mistake 1—Using Uninitialized Connection Handler Pointers

In the examples shown in this chapter, we've called `mysql_init()` by passing a `NULL` argument to it. This 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 problems. The following discussion points out some problems to watch out for.

If you pass a pointer to `mysql_init()`, it should actually point to something. Consider this 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, `NULL`) before the program starts up. `mysql_init()` sees a `NULL` argument, so it initializes and allocates a new connection handler. Unfortunately, `conn` is still `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`.

Mistake 2—Failing to Test for a Valid Result Set

Remember to check the status of calls from which you expect to get a result set. This 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`, and the `while` loop shouldn't even be executed. Test the return value of functions that return result sets to make sure you actually have something to work with.

Mistake 3—Failing to Account for `NULL` Column Values

Don't forget to check whether or not 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 crashes and your friend concludes you're a lousy programmer. The loop should be written like this 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 or not 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. This code violates that principle:

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

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

What's the problem? `to_str` must point to an existing buffer. In this example, it doesn't—it points to some random location. Don't pass an uninitialized pointer as the `to_str` argument to `mysql_escape_string()` unless you want it to stomp merrily all over some random piece of memory.