7

Writing MySQL Programs Using C

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

- Connection management routines that open and close a session with a server
- Routines that construct SQL statements, send them to the server, and process the results
- Status-checking and error-reporting functions that determine the reason for an error when an API call fails
- Routines that process options given in option files or on the command line

This chapter shows how to use the C client library to write your own programs, using conventions that are reasonably consistent with those used by the client programs included in the MySQL distribution.

The first part of this chapter develops a series of short programs. The series culminates in a simple program that serves as the framework for a client skeleton that does nothing but connect to and disconnect from the server. The reason for this is that although MySQL client programs are written for different purposes, one thing all have in common is that they must establish a connection to the server.

The resulting skeleton program is generic, but it also includes code to process options and handle errors, so it is usable as the basis for any number of other client programs. After developing it, we'll pause to consider how to execute various kinds of SQL statements. Initially, we'll discuss how to handle specific hardcoded statements, and then develop code that can be used to process arbitrary statements. After that, we'll add some statement-processing code to the skeleton to develop another program that's similar to the `mysql` client and that can be used to issue statements interactively.

The chapter then demonstrates several other activities that the client library supports:

- Writing client programs that communicate with the server over secure connections using the Secure Sockets Layer (SSL) protocol
- Sending multiple statements to the server at once and processing the result sets that come back
- Using server-side prepared statements

This chapter discusses only those functions and data types from the client library needed for the example programs. For a comprehensive listing of functions and types, see Appendix G, "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 so that you can try them directly without typing them in yourself. For instructions on obtaining the `sampdb` distribution that contains these programs, see Appendix A, "Software Required to Use This Book." You'll find the programs under the `capi` directory of that distribution.

Where to Find Example Programs

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

7.1 Compiling and Linking Client Programs

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

When you write a MySQL client program in C, you'll need a C compiler, obviously. The examples shown here use `gcc`, the most common compiler used on Unix. You'll also need the MySQL header files and client library.

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

To compile and link a client program, you must specify where the MySQL header files and client library are located, if they are not installed in locations that the compiler and linker search by default. For the following examples, suppose that the header file and client library locations are `/usr/local/include/mysql` and `/usr/local/lib/mysql`. Modify the pathnames as appropriate for your own system.

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

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

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

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

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

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

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

If the message names the `floor()` function, add `-lm` to link in the math library. You might need to add other libraries as well.

As an alternative to figuring out the proper flags for compiling and linking MySQL programs yourself, you can use the `mysql_config` utility to do it for you. For example, the utility might indicate that the following options are needed:

```
% mysql_config --include
-I'/usr/local/mysql/include'
% mysql_config --libs
-L'/usr/local/mysql/lib' -lmysqlclient -lpthread -lz -lm -lrt -ldl
```

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

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

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

If you don't use `make` to build programs, I suggest you learn how so that you won't have to type a lot of program-building commands manually. Suppose that you have a client program, `myclient`, that comprises two source files, `main.c` and `lib.c`, and a header file, `myclient.h`.

A simple `Makefile` to build this program follows. Indented lines are indented with tabs; spaces will not work.

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

all: myclient

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

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

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

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

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

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

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

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

Another way to change `make` variables other than editing the `Makefile` is to specify them on the command line. For example, if your C compiler is named `cc` rather than `gcc`, you can say so like this:

```
% make CC=cc
```

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

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

When `make` runs, it executes each `mysql_config` command and uses its output to set the corresponding variable value. The `${shell}` construct shown here is supported by GNU `make`;

you might need to use a somewhat different syntax if your version of `make` isn't based on GNU `make`.

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

7.2 Connecting to the Server

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

Our first client program, `connect1`, consists of a single source file, `connect1.c`:

```
/*
 * connect1.c - connect to and disconnect from MySQL server
 */

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

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

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

int
main (int argc, char *argv[])
{
  MY_INIT (argv[0]);
  /* initialize client library */
  if (mysql_library_init (0, NULL, NULL))
  {
    fprintf (stderr, "mysql_library_init() failed\n");
    exit (1);
  }
  /* initialize connection handler */
  conn = mysql_init (NULL);
  if (conn == NULL)
```

```
  {
    fprintf (stderr, "mysql_init() failed (probably out of memory)\n");
    exit (1);
  }
  /* connect to server */
  if (mysql_real_connect (conn, opt_host_name, opt_user_name, opt_password,
      opt_db_name, opt_port_num, opt_socket_name, opt_flags) == NULL)
  {
    fprintf (stderr, "mysql_real_connect() failed\n");
    mysql_close (conn);
    exit (1);
  }
  /* disconnect from server, terminate client library */
  mysql_close (conn);
  mysql_library_end ();
  exit (0);
}
```

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

- `my_global.h` includes several other header files that are likely to be generally useful, such as `stdio.h`. It also includes Windows compatibility information if you're compiling the program on Windows. (You might not intend to use Windows yourself, but if you plan to distribute your code, using `my_global.h` will help anyone else who does compile under Windows.)
- `my_sys.h` contains portability macros and definitions for structures and functions used by the client library.
- `mysql.h` defines the primary MySQL-related constants and data structures.

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

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

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

1. Call `mysql_init()` to obtain a connection handler.
2. Call `mysql_real_connect()` to establish a connection to the server.

When you pass NULL to mysql_init(), it automatically allocates a MYSQL structure, initializes it, and returns a pointer to it. The MYSQL data type is a structure containing information about a connection. Variables of this type are called "connection handlers."

Another approach is to pass a pointer to an existing MYSQL structure. In this case, mysql_init() initializes that structure and returns a pointer to it without allocating the structure itself.

mysql_real_connect() takes about a zillion parameters:

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

  On Windows, the behavior is similar, except that for "localhost", a shared-memory or TCP/IP connection is used rather than a Unix socket file connection. On Windows, the connection is attempted to the local server using a named pipe if the host is "." or NULL and the server supports named-pipe connections.
- The username and password for the MySQL account to be used. If the name is NULL, the client library sends your login name to the server (or ODBC, for Windows). If the password is NULL, no password is sent.
- The name of the database to select as the default database after the connection has been established. If this value is NULL, no database is selected.
- The port number. This is used for TCP/IP connections. A value of 0 tells the client library to use the default port number.
- The socket filename. On Unix, the name is used for Unix socket file connections. On Windows, the name is interpreted as the name to use for a pipe connection. A value of NULL tells the client library to use the default socket (or pipe) name.
- A flags value. The connect1 program passes a value of 0 because it uses no special connection options.

For more information about mysql_real_connect(), see Appendix G, "C API Reference." The description there discusses in more detail issues such as how the hostname parameter interacts with the port number and socket filename parameters, and lists the options that can be specified in the flags parameter. The appendix also describes mysql_options(), which you can use to specify other connection-related options prior to calling mysql_real_connect().

To terminate the connection, invoke mysql_close() and pass it a pointer to the connection handler. If you allocated the handler automatically by passing NULL to mysql_init(), mysql_close() automatically deallocates the handler when you terminate the connection. After calling mysql_close(), the handler cannot be used for further communication with the server.

In addition to the connection-establishment code, `connect1.c` uses three other calls:

- `MY_INIT()` is an initialization macro. It sets a global variable to point to the name of your program (which you pass as its argument), for use by MySQL libraries in error messages. It also calls `my_init()` to perform some setup operations.
- `mysql_library_init()` initializes the MySQL client library. Call it before invoking any other `mysql_xxx()` functions.
- `mysql_library_end()` terminates use of the client library and performs any necessary cleanup. Call it when you are done using the client library.

To try `connect1`, compile and link it using the instructions given earlier in the chapter for building client programs, and then run it. Under Unix, run the program like this:

```
% ./connect1
```

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

```
% connect1
```

If `connect1` produces no output, it connected successfully. On the other hand, you might see something like this:

```
% ./connect1
mysql_real_connect() failed
```

This output indicates that no connection was established, but it doesn't tell you why. Very likely the reason for the failure is that the default connection parameters (hostname, username, and so on) are unsuitable. Assuming that is so, one way to fix the problem is to recompile the program after editing the initializers for the parameter variables and changing them to values that enable 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 program still would contain hardcoded values, which isn't very flexible if other people are to use it. It's also insecure because it exposes your password. You might think that the password becomes hidden when you compile your program into binary executable form, but it's not hidden at all if someone can run the `strings` utility on the binary. Also, anyone with read access to the source file can get the password with no work at all.

The preceding paragraph highlights two significant shortcomings of `connect1`:

- The error output isn't very informative about specific causes of problems.
- There is no flexible way for the user who runs the program to specify connection parameters. They are hardwired into the source code. It would be better to enable the user to override the parameters on the command line or in an option file.

The next section addresses these problems.

7.3 Handling Errors and Processing Command Options

Our next client, `connect2`, is similar to `connect1` in the sense that it connects to the MySQL server, disconnects, and exits. However, `connect2` has two important improvements:

- It does a better job of error reporting, using MySQL client library functions that return specific information about the causes of errors.
- It provides default connection parameters but enables the user to override them with options on the command line or in an option file.

7.3.1 Checking for Errors

Let's consider the topic of error-handling first. To start off, I want to emphasize that it's important to check for errors whenever you invoke a MySQL function that can fail. It seems to be fairly common in programming texts to say "Error checking is left as an exercise for the reader." I suppose that this is because checking for errors is—let's face it—such a bore. Nevertheless, it is necessary to test for error conditions and respond to them appropriately. The client library functions that return status values do so for a reason, and you ignore them at your peril. For example, if a function returns a pointer to a data structure or `NULL` to indicate an error, you'd better check the return value. Attempts to use `NULL` later in the program when a pointer to a valid data structure is expected will lead to strange results or crash your program.

Failure to check return values is an unnecessary cause of programming difficulties and is a phenomenon that plays itself out frequently on the MySQL mailing lists. Typical questions are "Why does my program crash when it issues this statement?" or "How come my query doesn't return anything?" In many cases, the program in question didn't check whether the connection was established successfully before issuing the statement or didn't check to make sure the server successfully executed the statement before trying to retrieve the results.

Don't make the mistake of assuming that every client library call succeeds. If you don't check return values, you'll end up trying to track down obscure problems that occur in your programs, or users of your programs will wonder why those programs behave erratically, or both.

Routines in the MySQL client library that return a value generally indicate success or failure in one of two ways, depending on whether the return value is a pointer or an integer.

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

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

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

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

Alternatively, write the test like this, which is equivalent and slightly simpler to write:

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

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

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

When a client library call does fail, three calls in the API tell you why:

- `mysql_error()` returns a string containing an error message.
- `mysql_errno()` returns a MySQL-specific numeric error code.
- `mysql_sqlstate()` returns an SQLSTATE code. The SQLSTATE value is more vendor neutral because it is based on the ANSI SQL and ODBC standards.

The argument to each function is a pointer to the connection handler. Call them immediately after an error occurs. If you issue another API call that returns a status, any error information you get from `mysql_error()`, `mysql_errno()`, or `mysql_sqlstate()` apply to the later call instead.

Generally, the user of a program will find an error message more enlightening than either error code, so if you report only one value, I suggest that it be the message. The examples in this chapter report all three values for completeness. However, it's tedious to write three function invocations every place an error might occur. Instead, let's write a utility function, `print_error()`, that prints an error message supplied by us as well as the error values provided by the MySQL client library routines. In other words, we won't write out the calls to the `mysql_errno()` `mysql_error()`, and `mysql_sqlstate()` functions like this each time an error test occurs:

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

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

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

`print_error()` prints the error message and calls the MySQL error functions. The `print_error()` call is simpler than the `fprintf()` call, so it's easier to write and it makes the program easier to read. Also, if `print_error()` is written to do something sensible even when `conn` is `NULL`, we can use it under circumstances such as when `mysql_init()` call fails. Then we won't have a mix of error-reporting calls—some to `fprintf()` and some to `print_error()`.

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

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

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

The following listing shows the definition of `print_error()`, which provides the benefits just discussed:

```
static void
print_error (MYSQL *conn, char *message)
{
```

```
  fprintf (stderr, "%s\n", message);
  if (conn != NULL)
  {
    fprintf (stderr, "Error %u (%s): %s\n",
             mysql_errno (conn), mysql_sqlstate (conn), mysql_error (conn));
  }
}
```

The part of `connect2.c` that needs to check for errors is similar to the corresponding code in `connect1.c`, and looks like this when we use `print_error()`:

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

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

The error-checking logic is based on the fact that both `mysql_init()` and `mysql_real_connect()` return `NULL` if they fail. Note that if `mysql_init()` fails, we pass `NULL` as the first argument to `print_error()`. That causes it not to invoke the MySQL error-reporting functions, because the connection handler passed to those functions cannot be assumed to contain any meaningful information. By contrast, if `mysql_real_connect()` fails, we do pass the connection handler to `print_error()`. The handler won't contain information that corresponds to a valid connection, but it *will* contain diagnostic information that can be extracted by the error-reporting functions. The handler also can be passed to `mysql_close()` to release any memory that may have been allocated automatically for it by `mysql_init()`. (Don't pass the handler to any other client routines, though! Because most of them assume a valid connection, your program may crash.)

The rest of the programs in this chapter perform error checking, and your own programs should, too. It's less work in the long run because you spend less time tracking down subtle problems.

7.3.2 Getting Connection Parameters at Runtime

Now we're ready to tackle the task of enabling users to specify connection parameters at runtime rather than using hardwired default parameters. The `connect1` program had a significant shortcoming in that the connection parameters were written literally into the source code. To change any of those values, you'd have to edit the source file and recompile it. That's not very convenient, especially if you intend to make your program available for other people to use. One common way to specify connection parameters at runtime is by using command-line options. For example, the programs in the MySQL distribution accept parameters in either of two forms, as shown in the following table.

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

To be consistent with the standard MySQL clients, our `connect2` client program accepts those same formats. It's easy to do this because the client library supports option processing. In addition, `connect2` has the capability to extract information from option files. This enables you to put connection parameters in `~/.my.cnf` (that is, the `.my.cnf` file in your home directory) or in any global option file. Then you need not specify the options on the command line each time you invoke the program. The client library makes it easy to check for MySQL option files and pull any relevant values from them. By adding only a few lines of code to your programs, you can make them option file-aware, with no need to reinvent the wheel by writing your own code to do it. (For a description of option file syntax, see Section F.2.2, "Option Files.")

Before showing how option processing works in `connect2` itself, we develop two of programs that illustrate the general principles involved. These show how option handling works fairly simply and without the added complication of connecting to the MySQL server and processing statements.

> **Note**
>
> MySQL provides two other options that relate to connection establishment. `--protocol` specifies the connection protocol (TCP/IP, Unix socket file, and so on), and `--shared-memory-base-name` specifies the name of the shared memory to use for shared-memory connections on Windows. This chapter doesn't cover either option, but if you are interested, the `sampdb` distribution contains the source code for a program, `protocol`, that shows how to use them.

7.3.2.1 Accessing Option File Contents

To read option files for connection parameter values, invoke `load_defaults()`. This function 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). It puts information from those option 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. When you parse the command options, you see the connection parameters in your normal option-processing code. The options are added to `argv[]` immediately after the command name and before any other arguments (rather than at the end), so that any connection parameters specified on the command line occur later than and thus override any options added by `load_defaults()`.

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

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

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

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

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

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

  MY_INIT (argv[0]);
  load_defaults ("my", client_groups, &argc, &argv);

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

  exit (0);
}
```

The option file-processing code involves several components:

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

- `MY_INIT()` is an initialization macro that we have used before. The important point here is that `MY_INIT()` calls `my_init()` to perform some setup operations required by `load_defaults()`.
- `load_defaults()` reads the option files. It takes four arguments: the prefix used in the names of your option files (this should always be `"my"`), the array listing the names of the option groups in which you're interested, and the addresses of your program's argument count and vector. Don't pass the values of the count and vector. Pass their addresses instead because `load_defaults()` needs to change their values. In particular, even though `argv` is already a pointer, you still pass `&argv`, that pointer's address.

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

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

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

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

```
% ./show_argv a b
Original argument vector:
arg 0: ./show_argv
arg 1: a
arg 2: b
Modified argument vector:
arg 0: ./show_argv
arg 1: --user=sampadm
arg 2: --password=secret
arg 3: --host=some_host
arg 4: a
arg 5: b
```

When `show_argv` prints the argument vector the second time, the values in the option file show up as part of the argument list. It's also possible that you'll see some options that were not specified on the command line or in your `~/.my.cnf` file. If this occurs, you will likely find that options for the `[client]` group are listed in a system-wide option file. This can happen because `load_defaults()` actually looks for option files in several locations. (For a list of these locations, see Section F.2.2, "Option Files.")

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

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

Change the line to this:

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

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

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

[show_argv]
host=other_host
```

With these changes, invoking show_argv again produces a result different from before:

```
% ./show_argv a b
Original argument vector:
arg 0: ./show_argv
arg 1: a
arg 2: b
Modified argument vector:
arg 0: ./show_argv
arg 1: --user=sampadm
arg 2: --password=secret
arg 3: --host=some_host
arg 4: --host=other_host
arg 5: a
arg 6: b
```

The order of option values in the argument array is determined by the order in which they are listed in your option file, not the order in which option groups are named in the client_groups[] array. This means you'll probably want to specify program-specific groups after the [client] group in your option file. That way, if you specify an option in both groups, the program-specific value takes precedence over the more general [client] group value. You can see this in the example just shown: The host option is 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. To use the values of environment variables such as MYSQL_TCP_PORT or MYSQL_UNIX_PORT, you must arrange for that yourself by using getenv(). I'm not going to add that capability to our clients, but here's a short code fragment that shows how to check the values of two MySQL-related environment variables:

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

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

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

> **`load_defaults()` and Security**
>
> On multiple-user systems, utilities such as the `ps` program can display argument lists from arbitrary processes, including those being run by other users. Because of this, you might be wondering if there are any process-snooping implications of `load_defaults()` taking passwords that it finds in option files and putting them in your argument list. This actually is not a problem because `ps` displays the original `argv[]` contents. Any password argument created by `load_defaults()` points to an area of memory that it allocates for itself. That area is not part of the original vector, so `ps` never sees it.
>
> On the other hand, a password that is given on the command line ***does*** show up in `ps`, so it's not a good idea to specify passwords that way. One precaution a program can take to help reduce the risk is to remove the password from the argument list as soon as it starts executing. Section 7.3.2.2, "Processing Command-Line Arguments," shows how to do that.

7.3.2.2 Processing Command-Line Arguments

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

Some of the characteristics of the client library option-processing routines are as follows:

- Precise specification of the option type and range of legal values. For example, you can indicate not only that an option must have integer values, but that it must be positive and a multiple of 1024.
- Integration of help text to make it easy to print a help message by calling a standard library function. There is no need to write your own special code to produce a help message.

- Built-in support for the standard `--no-defaults`, `--print-defaults`, `--defaults-file`, and `--defaults-extra-file` options. (These options are described in Section F.2.2, “Option Files.”)
- Support for a standard set of option prefixes, such as `--disable-`, `--enable-`, and `--loose-`, to make it easier to implement boolean (on/off) and ignorable options. (This capability is not used in this chapter, but is described in Section F.2, “Specifying Program Options.”)

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

`show_opt` enables you to experiment with various ways of specifying connection parameters (whether in option files or on the command line), and to see the result by showing you what values would be used to make a connection to the MySQL server. `show_opt` is useful for getting a feel for what will happen in our next client program, `connect2`, which hooks up this option-processing code with code that actually does connect to the server.

To illustrate what happens at each phase of argument processing, `show_opt` performs the following actions:

1. Sets up default values for the hostname, username, password, and other connection parameters.
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 the option processing routine `handle_options()` to process the argument vector, then prints the resulting connection parameter values and whatever is left in the argument vector.

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

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

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

static char *opt_host_name = NULL;    /* server host (default=localhost) */
static char *opt_user_name = NULL;    /* username (default=login name) */
static char *opt_password = NULL;     /* password (default=none) */
```

```
static unsigned int opt_port_num = 0; /* port number (use built-in value) */
static char *opt_socket_name = NULL;  /* socket name (use built-in value) */

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

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

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

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

```
  printf ("Original connection parameters:\n");
  printf ("hostname: %s\n", opt_host_name ? opt_host_name : "(null)");
  printf ("username: %s\n", opt_user_name ? opt_user_name : "(null)");
  printf ("password: %s\n", opt_password ? opt_password : "(null)");
  printf ("port number: %u\n", opt_port_num);
  printf ("socket filename: %s\n",
          opt_socket_name ? opt_socket_name : "(null)");

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

  MY_INIT (argv[0]);
  load_defaults ("my", client_groups, &argc, &argv);

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

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

  printf ("Connection parameters after calling handle_options():\n");
  printf ("hostname: %s\n", opt_host_name ? opt_host_name : "(null)");
  printf ("username: %s\n", opt_user_name ? opt_user_name : "(null)");
  printf ("password: %s\n", opt_password ? opt_password : "(null)");
  printf ("port number: %u\n", opt_port_num);
  printf ("socket filename: %s\n",
          opt_socket_name ? opt_socket_name : "(null)");

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

  exit (0);
}
```

The option-processing approach illustrated by `show_opt.c` involves several aspects that are common to any program that uses the MySQL client library to handle command options. In your own programs, you should do the same things:

1. In addition to the other files that we already have been including, include `my_getopt.h` as well. `my_getopt.h` defines the interface to MySQL's option-processing facilities.
2. Define an array of `my_option` structures. In `show_opt.c`, this array is named `my_opts`. The array should have one structure per option that the program understands. Each structure provides information such as an option's short and long names, its default value, whether the value is a number or string, and so forth.

3. After invoking `load_defaults()` to read the option files and set up the argument vector, call `handle_options()` to process the options. The first two arguments to `handle_options()` are the addresses of your program's argument count and vector. (Just as with `load_options()`, you pass the addresses of these variables, not their values.) The third argument points to the array of `my_option` structures. The fourth argument is a pointer to a helper function. The `handle_options()` routine and the `my_options` structures are designed to make it possible for most option-processing actions to be performed automatically for you by the client library. However, to permit special actions that the library does not handle, your program should also define a helper function for `handle_options()` to call. In `show_opt.c`, this function is named `get_one_option()`.

The `my_option` structure defines the types of information that must be specified for each option that the program understands:

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

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

- `name` is the long option name. This is the `--`*`name`* form of the option, without the leading dashes. For example, if the long option is `--user`, list it as `"user"` in the `my_option` structure.
- `id` is the short (single-letter) option name, or a code value associated with the option if it has no single-letter name. For example, if the short option is `-u`, list it as `'u'` in the `my_option` structure. For options that have only a long name and no corresponding single-character name, you should make up a set of option code values to be used internally for the short names. The values must be unique and different from all the single-character names. (To satisfy the latter constraint, make the codes greater than 255, the largest possible single-character value. For an example, see Section 7.6, "Writing Clients That Include SSL Support.")

- `comment` is an explanatory string that describes the purpose of the option. This is the text to be displayed in a help message.
- `value` is the address of a generic pointer, declared as a `uchar **` value. If the option takes an argument, `value` points to the variable where you want the argument to be stored. After the options have been processed, you can check that variable to see what the option has been set to. The data type of the variable that's pointed to must be consistent with the value of the `var_type` member. If the option takes no argument, `value` is `NULL`.
- `u_max_value` is another address of a generic pointer, but it's used only by the server. For client programs, set `u_max_value` to `NULL`.
- `typelib` currently is unused. In future MySQL releases, it may be used to enable a list of legal values to be specified, in which case any option value given will be required to match one of these values.
- `var_type` indicates what kind of value must follow the option name on the command line. The following table shows these types, their meanings, and the corresponding C type.

`var_type` Value	Meaning	C Type
`GET_NO_ARG`	No value	
`GET_BOOL`	Boolean value	`my_bool`
`GET_INT`	Integer value	`int`
`GET_UINT`	Unsigned integer value	`unsigned int`
`GET_LONG`	Long integer value	`long`
`GET_ULONG`	Unsigned long integer value	`unsigned long`
`GET_LL`	Long long integer value	`long long`
`GET_ULL`	Unsigned long long integer value	`unsigned long long`
`GET_STR`	String value	`char *`
`GET_STR_ALLOC`	String value	`char *`
`GET_DISABLED`	Option is disabled	
`GET_ENUM`	Enumeration value (currently unused)	
`GET_SET`	Set value (currently unused)	
`GET_DOUBLE`	Double-precision (floating-point) value	`double`

The difference between `GET_STR` and `GET_STR_ALLOC` is that for `GET_STR`, the client library sets the option variable to point directly at the value in the argument vector, whereas for `GET_STR_ALLOC`, it makes a copy of the argument and sets the option variable to point to the copy.

The GET_DISABLED type can be used to indicate that an option is no longer available, or that it is available only when the program is built a certain way (for example, with debugging support enabled). To see an example, take a look at the mysql.cc file in a MySQL source distribution.

- arg_type indicates whether a value follows the option name, and may be any of the values shown in the following table. If arg_type is NO_ARG, var_type should be GET_NO_ARG.

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

- def_value is for numeric-valued options. It is the default value to assign to the option if no explicit value is specified in the argument vector.
- min_value is for numeric-valued options. It is the smallest permitted value. Smaller values are bumped up to this value automatically. Use 0 to indicate "no minimum."
- max_value is for numeric-valued options. It is the largest permitted value. Larger values are bumped down to this value automatically. Use 0 to indicate "no maximum."
- sub_size is for numeric-valued options. It is an offset that is used to convert values from the range as given in the argument vector to the range that is used internally. For example, if values are given on the command line in the range from 1 to 256, but the program wants to use an internal range of 0 to 255, set sub_size to 1.
- block_size is for numeric-valued options. This value indicates a block size if it is nonzero. Option values given by the user are rounded down to the nearest multiple of this size if necessary. For example, if values must be even, set the block size to 2; handle_options() rounds odd values down to the nearest even number.
- app_type is reserved for application-specific use.

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

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

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

When `handle_options()` returns, the argument count and vector are reset appropriately to represent an argument list containing only the nonoption arguments.

Here is a sample invocation of `show_opt` and the resulting output (assuming that `~/.my.cnf` still has the same contents as for the final `show_argv` example in Section 7.3.2.1, "Accessing Option File Contents"):

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

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

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

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

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

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

You can add other cases to the `switch()` statement in `get_one_option()` as necessary (and we do so in `connect2` shortly). For example, `get_one_option()` is useful for handling password options. When you specify such an option, the password value may or may not be given, as indicated by `OPT_ARG` in the option information structure. That is, you can specify the option as `--password` or `--password=`*`your_pass`* if you use the long-option form, or as `-p` or `-p`*`your_pass`* if you use the short-option form. MySQL clients typically permit you to omit the password value on the command line, and then prompt you for it. This enables you to avoid giving the password on the command line, which keeps people from seeing your password. In later programs, we use `get_one_option()` to check whether a password value was given. We save the value if so, and otherwise set a flag to indicate that the program should prompt the user for a password before attempting to connect to the server.

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

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

7.3.3 Incorporating Option Processing into a Client Program

Now we're ready to write `connect2.c`. It has the following characteristics:

- It connects to the MySQL server, disconnects, and exits. This is similar to `connect1.c`, but uses the `print_error()` function developed earlier for reporting errors.
- It processes options on the command line or in option files. This is done using code similar to that from `show_opt.c`, modified to prompt the user for a password if necessary.

The resulting source file, `connect2.c`, looks like this:

```
/*
 * connect2.c - connect to MySQL server, using connection parameters
 * specified in an option file or on the command line
 */

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

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

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

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

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

static struct my_option my_opts[] =   /* option information structures */
{
  {"help", '?', "Display this help and exit",
  NULL, NULL, NULL,
  GET_NO_ARG, NO_ARG, 0, 0, 0, 0, 0, 0},
  {"host", 'h', "Host to connect to",
  (uchar **) &opt_host_name, NULL, NULL,
  GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
  {"password", 'p', "Password",
  (uchar **) &opt_password, NULL, NULL,
```

```
  GET_STR, OPT_ARG, 0, 0, 0, 0, 0, 0},
  {"port", 'P', "Port number",
  (uchar **) &opt_port_num, NULL, NULL,
  GET_UINT, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
  {"socket", 'S', "Socket path",
  (uchar **) &opt_socket_name, NULL, NULL,
  GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
  {"user", 'u', "User name",
  (uchar **) &opt_user_name, NULL, NULL,
  GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
  { NULL, 0, NULL, NULL, NULL, NULL, GET_NO_ARG, NO_ARG, 0, 0, 0, 0, 0, 0 }
};

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

static my_bool
get_one_option (int optid, const struct my_option *opt, char *argument)
{
  switch (optid)
  {
  case '?':
    my_print_help (my_opts);  /* print help message */
    exit (0);
  case 'p':                   /* password */
    if (!argument)            /* no value given; solicit it later */
      ask_password = 1;
    else                      /* copy password, overwrite original */
    {
      opt_password = strdup (argument);
      if (opt_password == NULL)
      {
        print_error (NULL, "could not allocate password buffer");
        exit (1);
      }
      while (*argument)
        *argument++ = 'x';
      ask_password = 0;
    }
```

```
    break;
  }
  return (0);
}

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

  MY_INIT (argv[0]);
  load_defaults ("my", client_groups, &argc, &argv);

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

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

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

  /* initialize client library */
  if (mysql_library_init (0, NULL, NULL))
  {
    print_error (NULL, "mysql_library_init() failed");
    exit (1);
  }

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

  /* connect to server */
  if (mysql_real_connect (conn, opt_host_name, opt_user_name, opt_password,
      opt_db_name, opt_port_num, opt_socket_name, opt_flags) == NULL)
  {
    print_error (conn, "mysql_real_connect() failed");
```

```
    mysql_close (conn);
    exit (1);
  }

  /* ... issue statements and process results here ... */

  /* disconnect from server, terminate client library */
  mysql_close (conn);
  mysql_library_end ();
  exit (0);
}
```

Compared to the `connect1` and `show_opt` programs developed earlier, `connect2` does a few new things:

- It permits a default database to be specified as a command-line argument. This is consistent with the behavior of the standard clients in MySQL distributions.
- If a password value is present in the argument vector, `get_one_option()` makes a copy of it and overwrites the original. This minimizes the time window during which a password specified on the command line is visible to `ps` or to other system status programs. (The window is only *minimized*, not eliminated. Specifying passwords on the command line still is a security risk.)
- If a password option is given with no value, `get_one_option()` sets a flag to indicate that the program should prompt the user for a password. That's done in `main()` after all options have been processed, using the `get_tty_password()` function. This is a utility routine in the client library that prompts for a password without echoing it on the screen. You may ask, "Why not just call `getpass()`?" The answer is that not all systems have that function (for example, Windows does not). `get_tty_password()` is portable across systems because it's configured to adjust to system idiosyncrasies.

Compile and link `connect2`, then try running it:

```
% ./connect2
```

If `connect2` produces no output (as just shown), it connected successfully. On the other hand, you might see something like this:

```
% ./connect2
mysql_real_connect() failed:
Error 1045 (28000): Access denied for user 'sampadm'@'localhost'
(using password: NO)
```

This output indicates no connection was established, and it says why. In this case, `Access denied` means that you need to supply appropriate connection parameters. With `connect1`, there was no way to do so short of editing and recompiling. `connect2` connects to the MySQL server according to the options you specify. Assume that there is no option file to complicate matters. If you invoke `connect2` with no arguments, it connects to `localhost` and passes your

Unix login name and no password to the server. If instead you invoke `connect2` as shown in the following command, it prompts for a password (because there is no password value immediately following `-p`), connects to `some_host`, and passes the username `some_user` to the server as well as the password you enter when prompted:

```
% ./connect2 -h some_host -p -u some_user some_db
```

`connect2` also passes the database name `some_db` to `mysql_real_connect()` to make that the default database. If there is an option file, `connect2` processes its contents and modifies the connection parameters accordingly.

Let's step back for a moment and consider what's been achieved so far. The work that has gone into producing `connect2` accomplishes something that's necessary for every MySQL client: connecting to the server using appropriate parameters. It also does a good job of reporting errors if the connection attempt fails. What we have now serves as a framework that can be used as the basis for many different client programs. To write a new client, do this:

1. Make a copy of `connect2.c`.
2. If the program accepts additional options other than the standard ones that `connect2.c` knows about, add them to the `my_opts` array and modify the option-processing loop.
3. Add your own application-specific code between the connect and disconnect calls.

And you're done.

All the real action for your application will take place between the `mysql_real_connect()` and `mysql_close()` calls, but having a reusable skeleton means that you can concentrate more on what you're really interested in—accessing the content of your databases.

7.4 Processing SQL Statements

The purpose of connecting to the server is to conduct a conversation with it by executing statements while the connection is open. This section shows how to do that. Each statement execution involves these steps:

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

The MySQL client library includes two sets of routines for statement execution. The first set sends each statement to the server as a string and returns the results for all columns in string format. The second set uses a binary protocol that enables nonstring data values to be sent and returned in native format without conversion to and from string format.

This section discusses the original method for processing SQL statements. Section 7.8, "Using Server-Side Prepared Statements," covers the binary protocol.

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

The other statement-issuing function, `mysql_query()`, is more restrictive in what it permits in the statement string but often is easier to use. Any statement passed to `mysql_query()` should be a null-terminated string. This means the statement text cannot contain null bytes because those would cause it to be interpreted erroneously as shorter than it really is. Generally speaking, if your statement 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 statements using standard C library string functions that you're probably already familiar with, such as `strcpy()` and `sprintf()`.

Another factor to consider in constructing statements is whether you need to perform any character-escaping operations. This is necessary if you want to construct statements using values that contain binary data or other troublesome characters, such as quotes or backslashes. This is discussed in Section 7.4.7.1, "Working with Strings That Contain Special Characters."

A simple outline of statement handling looks like this:

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

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

A statement may fail for many reasons. Common causes of failure include the following:

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

Statements may be grouped into two broad categories: those that modify rows and those that return a result set (a set of rows). Statements such as `INSERT`, `DELETE`, and `UPDATE` modify rows and return a count to indicate the number of affected rows.

Statements such as `SELECT` and `SHOW` return a result set. In the MySQL C API, the result set returned by such statements is represented by the `MYSQL_RES` data type. This is a structure that contains the data values for the rows, and also metadata about the values (such as the column names and data value lengths). It is permitted for a result set to be empty (that is, to contain zero rows).

7.4.1 Handling Statements That Modify Rows

To process a statement that modifies rows, issue it with `mysql_query()` or `mysql_real_query()`. If the statement 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 statement that modifies rows:

```
if (mysql_query (conn, "INSERT INTO my_tbl SET name = 'My Name'") != 0)
{
  print_error (conn, "INSERT statement failed");
}
else
{
  printf ("INSERT statement succeeded; number of rows affected: %lu\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 may not work on all systems. Casting the value to `unsigned long` and using a print format of `%lu` solves the problem. The same principle applies to any other functions that return `my_ulonglong` values, such as `mysql_num_rows()` and `mysql_insert_id()`. If you want your client programs to be portable across different systems, keep this in mind.

`mysql_affected_rows()` returns the number of rows affected by the statement, but the meaning of “rows affected” depends on the type of statement. 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, request that behavior when you connect to the server by passing a value of `CLIENT_FOUND_ROWS` in the flags parameter to `mysql_real_connect()`.

7.4.2 Handling Statements That Return a Result Set

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

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

```
MYSQL_RES *res_set;

if (mysql_query (conn, "SHOW TABLES FROM sampdb") != 0)
  print_error (conn, "mysql_query() failed");
else
{
  res_set = mysql_store_result (conn);  /* generate result set */
  if (res_set == NULL)
    print_error (conn, "mysql_store_result() failed");
  else
  {
    /* process result set, then deallocate it */
    process_result_set (conn, res_set);
    mysql_free_result (res_set);
  }
}
```

The example hides the details of result set processing within another function, `process_result_set()`, which we have not yet defined. Generally, operations that handle a result set are based on a loop that looks something like this:

```
MYSQL_ROW row;

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

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

- `MYSQL_ROW` is a pointer type, so declare a variable of that type as `MYSQL_ROW row`, not as `MYSQL_ROW *row`.
- Values for all data types, even numeric types, are returned in the `MYSQL_ROW` array as strings. To treat a value as a number, you must convert the string yourself.
- The strings in a `MYSQL_ROW` array are null-terminated. However, if a column can contain binary data, it might contain null bytes, so do not treat the value as a null-terminated string. Get the column length to find out how long the column value is. (Section 7.4.6, "Using Result Set Metadata," discusses how to determine column lengths.)
- SQL `NULL` values are represented by C `NULL` pointers in the `MYSQL_ROW` array. Unless you know that a column is declared `NOT NULL`, you should always check whether values for the column are `NULL`, or your program may crash as a result of attempting to dereference a `NULL` pointer.

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

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 ("Number of rows returned: %lu\n",
            (unsigned long) mysql_num_rows (res_set));
}
```

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

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

The version of `process_result_set()` just shown takes a rather minimalist approach to printing column values—one that has certain shortcomings. Suppose that you execute this query:

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

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

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

We can 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 must 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 statement handler a bit, we'll write a nicer display formatter in Section 7.4.6, "Using Result Set Metadata."

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

7.4.3 A General-Purpose Statement Handler

The preceding statement-handling examples use knowledge of whether the statement should return any data. That was possible because the statements 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 might not always know what kind of statement a given statement represents. For example, if you execute a statement that you read from the keyboard or from a file, it might be anything. You won't know ahead of time whether to expect it to return rows, or even whether it's legal. What then? You certainly don't want to try to parse the statement to determine what kind of statement it is. That's the server's job, anyway.

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

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

   If `mysql_field_count()` returns 0, it means the statement returned no columns, and thus no result set. (This indicates that it was a statement such as `INSERT`, `DELETE`, or `UPDATE`.)

   If `mysql_field_count()` returns a nonzero value, it means that an error occurred, because the statement should have returned a result set but didn't. This can happen for various reasons. For example, the result set may have been so large that memory allocation failed, or a network outage between the client and the server may have occurred while fetching rows.

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

```
void
process_statement (MYSQL *conn, char *stmt_str)
{
MYSQL_RES *res_set;

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

  /* the statement succeeded; determine whether it returned data */
  res_set = mysql_store_result (conn);
  if (res_set)      /* a result set was returned */
  {
    /* process rows and free the result set */
    process_result_set (conn, res_set);
    mysql_free_result (res_set);
  }
  else              /* no result set was returned */
  {
    /*
     * does the lack of a result set mean that the statement didn't
     * return one, or that it should have but an error occurred?
     */
    if (mysql_field_count (conn) == 0)
    {
      /*
       * statement generated no result set (it was not a SELECT,
       * SHOW, DESCRIBE, etc.); just report rows-affected value.
       */
      printf ("Number of rows affected: %lu\n",
              (unsigned long) mysql_affected_rows (conn));
    }
    else  /* an error occurred */
    {
      print_error (conn, "Could not retrieve result set");
    }
  }
}
```

7.4.4 Alternative Approaches to Statement Processing

The version of `process_statement()` just shown has these three properties:

- It uses `mysql_query()` to issue the statement.
- 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 those aspects of statement handling:

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

Any or all of these approaches can be used instead of those used in `process_statement()`. Here is a `process_real_statement()` function analogous to `process_statement()` that uses all three alternatives:

```
void
process_real_statement (MYSQL *conn, char *stmt_str, unsigned int len)
{
MYSQL_RES *res_set;

  if (mysql_real_query (conn, stmt_str, len) != 0) /* the statement failed */
  {
    print_error (conn, "Could not execute statement");
    return;
  }

  /* the statement succeeded; determine whether it returned data */
  res_set = mysql_use_result (conn);
  if (res_set)      /* a result set was returned */
  {
    /* process rows and free the result set */
    process_result_set (conn, res_set);
    mysql_free_result (res_set);
  }
  else              /* no result set was returned */
  {
    /*
     * does the lack of a result set mean that the statement didn't
     * return one, or that it should have but an error occurred?
```

```
      */
    if (mysql_errno (conn) == 0)
    {
      /*
       * statement generated no result set (it was not a SELECT,
       * SHOW, DESCRIBE, etc.); just report rows-affected value.
       */
      printf ("Number of rows affected: %lu\n",
              (unsigned long) mysql_affected_rows (conn));
    }
    else  /* an error occurred */
    {
      print_error (conn, "Could not retrieve result set");
    }
  }
}
```

7.4.5 `mysql_store_result()` Versus `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. 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 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 buffers them on the client side. Subsequent calls to `mysql_fetch_row()` never return an error because they simply pull a row out of a data structure that already holds the result set. Consequently, a `NULL` return from `mysql_fetch_row()` always means you've reached the end of the result set.

By contrast, `mysql_use_result()` retrieves no rows itself. Instead, it simply initiates a row-by-row retrieval, which you must complete yourself by calling `mysql_fetch_row()` for each row. In this case, although a `NULL` return from `mysql_fetch_row()` normally still means the end of the result set has been reached, it may mean instead that an error occurred while communicating with the server. You can distinguish the two outcomes by calling `mysql_errno()` or `mysql_error()`.

`mysql_store_result()` has greater memory and processing requirements than `mysql_use_result()` because the entire result set is maintained in the client. The overhead for memory allocation and data structure setup is greater, and a client that retrieves large result sets runs the risk of running out of memory. If you're going to retrieve a lot of rows in a single result set, you might 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 waiting 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, particularly if you are using a storage engine like MyISAM that uses table locks: Tables from which you retrieve data are read-locked for the duration of the query. Other clients that are trying to update those tables will be blocked.

Offsetting the additional memory requirements incurred by `mysql_store_result()` are certain benefits of having access to the entire result set at once. All rows of the set are available, so you have random access into them: The `mysql_data_seek()`, `mysql_row_seek()`, and `mysql_row_tell()` functions enable 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 enables 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 access certain types of column information that are unavailable when you use `mysql_use_result()`. The number of rows in the result set is obtained by calling `mysql_num_rows()`. The maximum widths of the values in each column are stored in the `max_width` member of the `MYSQL_FIELD` column information structures. With `mysql_use_result()`, `mysql_num_rows()` doesn't return the correct value until you've fetched all the rows; similarly, `max_width` is unavailable because it can be calculated only after every row's data have been seen.

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

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

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

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

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

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 the `--quick` option is given.

7.4.6 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 following:

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

Metadata availability depends partially on your result set processing method. As indicated in Section 7.4.5, "`mysql_store_result()` Versus `mysql_use_result()`," if you want to use the row count or maximum column length values, you must create the result set with `mysql_store_result()`, not with `mysql_use_result()`.

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

- Column names and widths are useful for producing nicely formatted output that lines up vertically and has column titles.
- The column count indicates how many times to iterate through a loop that processes successive column values for data rows.
- The row and column counts help you allocate data structures that depend on the dimensions of the result set.
- The data type of a column enables you to tell whether a column represents a number, whether it might contain binary data, and so forth.

Earlier, in Section 7.4.2, "Handling Statements That Return a Result Set," we wrote a version of `process_result_set()` that printed columns from result set rows in tab-delimited format. That's good for certain purposes (such as when you want to import the data into a spreadsheet), but it's not a nice display format for visual inspection or for printouts. Recall that our earlier version of `process_result_set()` produced this output:

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

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

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

The display algorithm performs these steps:

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. Print numbers right justified and the word "NULL" for `NULL` values.
4. At the end, print a count of the rows retrieved.

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

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

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

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

/* determine column display widths; requires result set to be */
/* generated with mysql_store_result(), not mysql_use_result() */
mysql_field_seek (res_set, 0);
for (i = 0; i < mysql_num_fields (res_set); i++)
{
  field = mysql_fetch_field (res_set);
  col_len = strlen (field->name);
  if (col_len < field->max_length)
    col_len = field->max_length;
  if (col_len < 4 && !IS_NOT_NULL (field->flags))
    col_len = 4;  /* 4 = length of the word "NULL" */
  field->max_length = col_len;  /* reset column info */
}
```

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

- The length of field->name, the column title.
- field->max_length, the length of the longest data value in the column.
- The length of the string “NULL”, if field->flags indicates that the column can contain NULL values.

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 permitted, 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 that it’s okay. (If you prefer an alternative approach that doesn’t modify max_length, allocate an array of unsigned long values and store the calculated widths in that array.)

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

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

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

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

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

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

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

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

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

```
  fputc ('\n', stdout);
}

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

  /* determine column display widths; requires result set to be */
  /* generated with mysql_store_result(), not mysql_use_result() */
  mysql_field_seek (res_set, 0);
  for (i = 0; i < mysql_num_fields (res_set); i++)
  {
    field = mysql_fetch_field (res_set);
    col_len = strlen (field->name);
    if (col_len < field->max_length)
      col_len = field->max_length;
    if (col_len < 4 && !IS_NOT_NULL (field->flags))
      col_len = 4;  /* 4 = length of the word "NULL" */
    field->max_length = col_len;  /* reset column info */
  }

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

  while ((row = mysql_fetch_row (res_set)) != NULL)
  {
    mysql_field_seek (res_set, 0);
    fputc ('|', stdout);
    for (i = 0; i < mysql_num_fields (res_set); i++)
    {
      field = mysql_fetch_field (res_set);
      if (row[i] == NULL)       /* print the word "NULL" */
        printf (" %-*s |", (int) field->max_length, "NULL");
      else if (IS_NUM (field->type))  /* print value right-justified */
        printf (" %*s |", (int) field->max_length, row[i]);
```

```
      else              /* print value left-justified */
        printf (" %-*s |", (int) field->max_length, row[i]);
    }
    fputc ('\n', stdout);
  }
  print_dashes (res_set);
  printf ("Number of rows returned: %lu\n",
          (unsigned long) mysql_num_rows (res_set));
}
```

The MySQL client library provides several ways to access 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 combination of `mysql_field_seek()` and `mysql_fetch_field()` is only one way of getting `MYSQL_FIELD` structures. For other ways, see the descriptions of the `mysql_fetch_fields()` and `mysql_fetch_field_direct()` functions in Appendix G, "C API Reference."

> **Use the `metadata` Program to Display Result Set Metadata**
>
> The `sampdb` distribution contains the source for a program named `metadata` that you can compile and run to see what metadata different kinds of statements produce. It prompts for and executes SQL statements, but displays result set metadata rather than result set contents. For comparison, invoke `mysql` with the `--column-type-info` option and execute the same statements.

7.4.7 Encoding Special Characters and Binary Data

Programs that execute statements must take care with certain characters. For example, to include a quote character within a quoted string, either double the quote or precede it by a backslash:

```
'O''Malley'
'O\'Malley'
```

This section describes how to handle quoting issues in string values and how to work with binary data.

7.4.7.1 Working with Strings That Contain Special Characters

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

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

```
char stmt_buf[1024];

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

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

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

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

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

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

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

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

To use `mysql_real_escape_string()`, invoke it like this:

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

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

from_str points to a char buffer containing the string to be encoded. This string 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_real_escape_string() to allocate space for you. The length of the buffer pointed to by to_str must be at least (from_len*2)+1 bytes long. (It's possible that every character in from_str needs encoding with two characters; the extra byte is for the terminating null.)

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

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

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

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

p = strcpy (stmt_buf, "SELECT * FROM mytbl WHERE name='");
p += strlen (p);
p += mysql_real_escape_string (conn, p, name_val, strlen (name_val));
*p++ = '\'';
*p = '\0';
```

Yes, that's ugly. To simplify the code a bit, at the cost of using a second buffer, do this instead:

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

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

It's important to make sure that the buffers you pass to mysql_real_escape_string() really exist. Consider the following example, which violates that principle:

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

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

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

7.4.7.2 Working with Binary Data

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

`mysql_real_escape_string()` is essential for working with binary data. This section shows how to do so, using image data read from a file. The discussion applies to any other form of binary data as well.

Suppose that you want to read images from files and store them in a table named `picture`, along with a unique identifier. The `MEDIUMBLOB` type is a good choice for binary values less than 16MB in size, so you could use a table specification like this:

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

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

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

  /* begin creating an INSERT statement, adding the id value */
  sprintf (stmt_buf,
           "INSERT INTO picture (pict_id,pict_data) VALUES (%d,'",
           id);
  p = stmt_buf + strlen (stmt_buf);
  /* read data from file in chunks, encode each */
  /* chunk, and add to end of statement */
  while ((from_len = fread (buf, 1, sizeof (buf), f)) > 0)
  {
    /* don't overrun end of statement buffer! */
    if (p + (2*from_len) + 3 > stmt_buf + sizeof (stmt_buf))
    {
      print_error (NULL, "image is too big");
      return (1);
    }
    p += mysql_real_escape_string (conn, p, buf, from_len);
  }
```

```
  *p++ = '\'';
  *p++ = ')';
  status = mysql_real_query (conn, stmt_buf, (unsigned long) (p - stmt_buf));
  return (status);
}
```

`load_image()` doesn't allocate a very large statement buffer (1MB), so it works only for relatively small images. In a real-world application, you might allocate the buffer dynamically based on the size of the image file.

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

7.5 An Interactive Statement-Execution Program

We are now in a position to put together much of what we've developed so far to write a simple interactive statement-execution client, `exec_stmt`. This program lets you enter statements, executes them using our general-purpose statement handler `process_statement()`, and displays the results using the `process_result_set()` display formatter developed earlier.

`exec_stmt` is similar in some ways to `mysql`, although of course with not as many features. There are several restrictions on what `exec_stmt` permits as input:

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

It turns out that `exec_stmt` is almost completely trivial to write (about a dozen lines of new code). Almost everything we need is provided by our client program skeleton (`connect2.c`) and by other functions that we have written already. The only thing we need to add is a loop that collects input lines and executes them.

To construct `exec_stmt`, begin by copying the client skeleton `connect2.c` to `exec_stmt.c`. Then add to that the code for the `process_statement()`, `process_result_set()`, and `print_dashes()` functions. Finally, in `exec_stmt.c`, look for the line in `main()` that says this:

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

Replace that line with this `while` loop:

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

  fprintf (stderr, "query> ");                  /* print prompt */
```

```
  if (fgets (buf, sizeof (buf), stdin) == NULL) /* read statement */
    break;
  if (strcmp (buf, "quit\n") == 0 || strcmp (buf, "\\q\n") == 0)
    break;
  process_statement (conn, buf);                /* execute it */
}
```

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

```
% ./exec_stmt
query> USE sampdb
Number of rows affected: 0
query> SELECT DATABASE(), USER()
+------------+-------------------+
| DATABASE() | USER()            |
+------------+-------------------+
| sampdb     | sampadm@localhost |
+------------+-------------------+
Number of rows returned: 1
query> SELECT COUNT(*) FROM president
+----------+
| COUNT(*) |
+----------+
|       43 |
+----------+
Number of rows returned: 1
query> SELECT last_name, first_name FROM president ORDER BY last_name LIMIT 3
+-----------+-------------+
| last_name | first_name  |
+-----------+-------------+
| Adams     | John        |
| Adams     | John Quincy |
| Arthur    | Chester A.  |
+-----------+-------------+
Number of rows returned: 3
query> CREATE TABLE t (i INT)
Number of rows affected: 0
query> SELECT j FROM t
Could not execute statement
Error 1054 (42S22): Unknown column 'j' in 'field list'
query> USE mysql
Could not execute statement
Error 1044 (42000): Access denied for user 'sampadm'@'localhost' to
database 'mysql'
```

7.6 Writing Clients That Include SSL Support

MySQL includes SSL support, and you can use it to write your own programs that access the server over secure connections. To show how this is done, this section describes the process of modifying `exec_stmt` to produce a similar client named `exec_stmt_ssl` that outwardly is much the same but enables encrypted connections to be established. For `exec_stmt_ssl` to work properly, MySQL must have been built with SSL support, and the server must be started with the proper options that identify its certificate and key files. You'll also need certificate and key files on the client end. For more information, see Section 13.5, "Setting Up Secure Connections Using SSL."

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

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

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

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

   ```
   static struct my_option my_opts[] =   /* option information structures */
   {
     {"help", '?', "Display this help and exit",
     NULL, NULL, NULL,
     GET_NO_ARG, NO_ARG, 0, 0, 0, 0, 0, 0},
     {"host", 'h', "Host to connect to",
     (uchar **) &opt_host_name, NULL, NULL,
     GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
     {"password", 'p', "Password",
     (uchar **) &opt_password, NULL, NULL,
     GET_STR, OPT_ARG, 0, 0, 0, 0, 0, 0},
     {"port", 'P', "Port number",
     (uchar **) &opt_port_num, NULL, NULL,
   ```

```
  GET_UINT, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
  {"socket", 'S', "Socket path",
  (uchar **) &opt_socket_name, NULL, NULL,
  GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
  {"user", 'u', "User name",
  (uchar **) &opt_user_name, NULL, NULL,
  GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},

#include <sslopt-longopts.h>

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

`sslopt-longopts.h` is a public MySQL header file. Its contents look like this (reformatted slightly):

```
#ifdef HAVE_OPENSSL
  {"ssl", OPT_SSL_SSL,
  "Enable SSL for connection (automatically enabled with other flags).
  Disable with --skip-ssl.",
  (uchar **) &opt_use_ssl, (uchar **) &opt_use_ssl, 0,
  GET_BOOL, NO_ARG, 0, 0, 0, 0, 0, 0},
  {"ssl-ca", OPT_SSL_CA,
  "CA file in PEM format (check OpenSSL docs, implies --ssl).",
  (uchar **) &opt_ssl_ca, (uchar **) &opt_ssl_ca, 0,
  GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
  {"ssl-capath", OPT_SSL_CAPATH,
  "CA directory (check OpenSSL docs, implies --ssl).",
  (uchar **) &opt_ssl_capath, (uchar **) &opt_ssl_capath, 0,
  GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
  {"ssl-cert", OPT_SSL_CERT, "X509 cert in PEM format (implies --ssl).",
  (uchar **) &opt_ssl_cert, (uchar **) &opt_ssl_cert, 0,
  GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
  {"ssl-cipher", OPT_SSL_CIPHER, "SSL cipher to use (implies --ssl).",
  (uchar **) &opt_ssl_cipher, (uchar **) &opt_ssl_cipher, 0,
  GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
  {"ssl-key", OPT_SSL_KEY, "X509 key in PEM format (implies --ssl).",
  (uchar **) &opt_ssl_key, (uchar **) &opt_ssl_key, 0,
  GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
#ifdef MYSQL_CLIENT
  {"ssl-verify-server-cert", OPT_SSL_VERIFY_SERVER_CERT,
   "Verify server's \"Common Name\" in its cert against hostname used
   when connecting. This option is disabled by default.",
   (uchar **) &opt_ssl_verify_server_cert,
   (uchar **) &opt_ssl_verify_server_cert, 0,
   GET_BOOL, NO_ARG, 0, 0, 0, 0, 0, 0},
#endif
#endif /* HAVE_OPENSSL */
```

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

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

When writing your own applications that define codes for other options, make sure that those codes have values different from the `OPT_SSL_`*`XXX`* symbols.

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

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

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

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

```
    exit (0);
  case 'p':                    /* password */
    if (!argument)             /* no value given; solicit it later */
      ask_password = 1;
    else                       /* copy password, overwrite original */
    {
      opt_password = strdup (argument);
      if (opt_password == NULL)
      {
        print_error (NULL, "could not allocate password buffer");
        exit (1);
      }
      while (*argument)
        *argument++ = 'x';
      ask_password = 0;
    }
    break;
#include <sslopt-case.h>
  }
  return (0);
}
```

sslopt-case.h includes additional cases for the switch() statement that detect when any of the SSL options were given and sets the opt_use_ssl variable if so. It looks like this:

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

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

If you use the preceding procedure, the usual load_defaults() and handle_options() routines take care of parsing the SSL-related options and setting their values for you automatically. The only other thing you need to do is pass SSL option information to the client library before connecting to the server if the options indicate that the user wants an SSL connection.

Do this by invoking `mysql_ssl_set()` after calling `mysql_init()` and before calling `mysql_real_connect()`. The sequence looks like this:

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

#ifdef HAVE_OPENSSL
  /* pass SSL information to client library */
  if (opt_use_ssl)
    mysql_ssl_set (conn, opt_ssl_key, opt_ssl_cert, opt_ssl_ca,
                   opt_ssl_capath, opt_ssl_cipher);
  mysql_options (conn,MYSQL_OPT_SSL_VERIFY_SERVER_CERT,
                 (char*)&opt_ssl_verify_server_cert);
#endif

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

This code doesn't test `mysql_ssl_set()` to see whether it returns an error. Any problems with the information you supply to that function results in an error when you call `mysql_real_connect()`.

Compile `exec_stmt_ssl.c` to produce the `exec_stmt_ssl` program and then run it. Assuming that the `mysql_real_connect()` call succeeds, you can proceed to issue statements. If you invoke `exec_stmt_ssl` with the appropriate SSL options, communication with the server should occur over an encrypted connection. To determine whether that is so, issue the following statement:

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

The value of `Ssl_cipher` will be nonblank if an encryption cipher is in use. (To make this easier, the version of `exec_stmt_ssl` included in the `sampdb` distribution issues the statement for you and reports the result.)

7.7 Using Multiple-Statement Execution

The MySQL client library supports multiple-statement execution capability. This enables you to send a string to the server consisting of multiple statements separated by semicolons, then retrieve the result sets one after the other.

Multiple-statement execution is not enabled by default, so you must tell the server when you want to use it. There are two ways to do this. The first is to add the `CLIENT_MULTI_STATEMENTS` option in the flags argument to `mysql_real_connect()` at connect time:

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

The other is to use `mysql_set_server_option()` to enable the capability for an existing connection. For example:

```
if (mysql_set_server_option (conn, MYSQL_OPTION_MULTI_STATEMENTS_ON) != 0)
  print_error (conn, "Could not enable multiple-statement execution");
```

Which method is preferable? If the program does not use stored procedures, either one is suitable. If the program does use stored procedures and invokes a `CALL` statement that returns a result set, use the first method. That's because `CLIENT_MULTI_STATEMENTS` also turns on the `CLIENT_MULTI_RESULTS` option, which must be enabled or an error occurs if a stored procedure attempts to return a result. (More preferable yet might be to add `CLIENT_MULTI_RESULTS` to the flags argument to `mysql_real_connect()`, because that makes it explicit that you're enabling the option.)

Two functions form the basis for checking the current status of result retrieval when you're processing multiple result sets:

- `mysql_more_results()` returns nonzero if more results are available and zero otherwise.
- `mysql_next_result()` returns a status and also initiates retrieval of the next set if more results are available. The status is zero if more results are available, –1 if not, and a value greater than zero if an error occurred.

You can use these functions by putting your result-retrieval code inside a loop. After retrieving a result with your usual code, check whether there are any results yet to be retrieved. If so, perform another iteration of the loop. If not, exit the loop. Depending on how you structure your loop, you may not need to call `mysql_more_results()` at all because you can also tell from the return value of `mysql_next_result()` whether more results are available.

In Section 7.4.3, "A General-Purpose Statement Handler," we wrote a function, `process_statement()`, that executes a statement and retrieves the result or displays the number of rows affected. By placing the result-retrieval code into a loop and incorporating `mysql_next_result()`, we can write a similar function, `process_multi_statement()`, that can retrieve multiple result sets:

```
void
process_multi_statement (MYSQL *conn, char *stmt_str)
{
MYSQL_RES *res_set;
int       status;
int       keep_going = 1;

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

  /* the statement(s) succeeded; enter result-retrieval loop */
  do {
    /* determine whether current statement returned data */
    res_set = mysql_store_result (conn);
    if (res_set)      /* a result set was returned */
    {
      /* process rows and free the result set */
      process_result_set (conn, res_set);
      mysql_free_result (res_set);
    }
    else              /* no result set was returned */
    {
      /*
       * does the lack of a result set mean that the statement didn't
       * return one, or that it should have but an error occurred?
       */
      if (mysql_field_count (conn) == 0)
      {
        /*
         * statement generated no result set (it was not a SELECT,
         * SHOW, DESCRIBE, etc.); just report rows-affected value.
         */
        printf ("Number of rows affected: %lu\n",
                (unsigned long) mysql_affected_rows (conn));
      }
      else  /* an error occurred */
      {
```

```
        print_error (conn, "Could not retrieve result set");
        keep_going = 0;
      }
    }
    /* determine whether more results exist */
    /* 0 = yes, -1 = no, >0 = error */
    status = mysql_next_result (conn);
    if (status != 0)    /* no more results, or an error occurred */
    {
      keep_going = 0;
      if (status > 0)   /* error */
        print_error (conn, "Could not execute statement");
    }
  } while (keep_going);
}
```

If you like, just test whether the result of `mysql_next_result()` is zero, and exit the loop if not. The disadvantage of this simpler strategy is that if there are no more results, you don't know whether you've reached the end normally or an error occurred. In other words, you don't know whether to print an error message.

7.8 Using Server-Side Prepared Statements

In the earlier parts of this chapter, the code for SQL statement processing is based on the set of functions provided by the MySQL client library that send and retrieve all information in string form. This section discusses how to use the binary client/server protocol. The binary protocol supports server-side prepared statements and enables transmission of data values in native format.

Not all statements can be prepared. The initial implementation of prepared statements supported only the following statements: `CREATE TABLE`, `DELETE`, `DO`, `INSERT`, `REPLACE`, `SELECT`, `SET`, `UPDATE`, and most variations of `SHOW`. The list of supported statements has expanded since. See the MySQL Reference Manual for the current list.

To use the binary protocol, you must create a statement handler. With this handler, send a statement to the server to be "prepared," or preprocessed. The server analyzes the statement, remembers it, and sends back information about it that the client library stores in the statement handler. Further processing for the statement uses this handler.

A statement to be prepared can be parameterized by including '`?`' characters to indicate where data values appear that you will supply later when you execute the statement. For example, you might prepare a statement that looks like this:

```
INSERT INTO score (event_id,student_id,score) VALUES(?,?,?)
```

This statement includes three '`?`' characters that act as parameter markers or placeholders. Later, you can supply data values to be bound to the placeholders. These complete the statement when you execute it. By parameterizing a statement, you make it reusable: The same statement can be executed multiple times, each time with a new set of data values. This means that you send the text of the statement only once, and that each time you execute the statement, you send only the data values. For repeated statement execution, this provides a performance boost:

- The server needs to analyze the statement only once, not each time it is executed.
- Network overhead is reduced, because you send only the data values for each execution, not an entire statement.
- Data values are sent without conversion to string form, which reduces execution overhead. For example, the three columns named in the preceding `INSERT` statement all are `INT` columns. Were you to use `mysql_query()` or `mysql_real_query()` to execute a similar `INSERT` statement, it would be necessary to convert the data values to strings for inclusion in the text of the statement. With the prepared statement interface, you send the data values separately in binary format.
- No conversion is needed for retrieving results, either. In result sets returned by prepared statements, nonstring values are returned in binary format without conversion to string form.

The binary protocol does have some disadvantages, compared to the original nonbinary protocol:

- It is more difficult to use because more setup is necessary for transmitting and receiving data values.
- The binary protocol does not support all statements. For example, `USE` statements don't work.
- For interactive programs, you may as well use the original protocol. In that case, each statement received from the user is executed only once. There is little benefit to using prepared statements, which provide the greatest efficiency gain for statements that you execute repeatedly.

The general procedure for using a prepared statement involves these steps:

1. Allocate a statement handler by calling `mysql_stmt_init()`. This function returns a pointer to the handler, which you use for the following steps.
2. Call `mysql_stmt_prepare()` to send a statement to the server to be prepared and associated with the statement handler. The server determines certain characteristics of the statement, such as what kind of statement it is, how many parameter markers it contains, and whether it produces a result set when executed.
3. If the statement contains any placeholders, you must provide data for each of them before you can execute it. To do this, set up a `MYSQL_BIND` structure for each parameter. Each structure indicates one parameter's data type, its value, whether it is `NULL`, and so on. Then bind these structures to the statement by calling `mysql_stmt_bind_param()`.
4. Invoke `mysql_stmt_execute()` to execute the statement.

5. If the statement modifies data rather than producing a result set (for example, if it is an `INSERT` or `UPDATE`), call `mysql_stmt_affected_rows()` to determine the number of rows affected by the statement.
6. If the statement produces a result set, call `mysql_stmt_result_metadata()` if you want to obtain metadata about the result set. To fetch the rows, you use `MYSQL_BIND` structures again, but this time they serve as receptacles for data returned from the server rather than a source of data to send to the server. You must set up one `MYSQL_BIND` structure for each column in the result set. They contain information about the values you expect to receive from the server in each row. Bind the structures to the statement handler by calling `mysql_stmt_bind_result()`, then invoke `mysql_stmt_fetch()` repeatedly to get each row. After each fetch, you can access the column values for the current row.

   Before calling `mysql_stmt_fetch()`, an optional action is to call `mysql_stmt_store_result()`. If you do this, the result set rows are fetched all at once from the server and buffered in memory on the client side. Also, the number of rows in the result set can be determined by calling `mysql_stmt_num_rows()`, which otherwise returns zero.

   After fetching the result set, call `mysql_stmt_free_result()` to release memory associated with it.
7. To re-execute the statement, return to step 3 to specify new parameter values.
8. To prepare a different statement using the handler, return to step 2.
9. When you're done with the statement handler, dispose of it by calling `mysql_stmt_close()`. If the client connection closes while the server still has prepared statements associated with the connection, the server disposes of them automatically.

A client application can prepare multiple statements, then execute each in the order appropriate to the application.

The following discussion describes how to write a simple program that inserts rows into a table and then retrieves them. The part of the program that processes an `INSERT` statement illustrates how to use placeholders in a statement and transmit data values to the server to be bound to the prepared statement when it is executed. The part that processes a `SELECT` statement shows how to retrieve a result set produced by a prepared statement. You can find the source for this program in the `prepared.c` and `process_prepared_statement.c` files in the `capi` directory of the `sampdb` distribution. I won't show the code for setting up the connection because it is similar to that for earlier programs.

The main part of the program that sets up to use prepared statements looks like this:

```
void
process_prepared_statements (MYSQL *conn)
{
MYSQL_STMT *stmt;
char       *use_stmt = "USE sampdb";
char       *drop_stmt = "DROP TABLE IF EXISTS t";
char       *create_stmt =
```

```
  "CREATE TABLE t (i INT, f FLOAT, c CHAR(24), dt DATETIME)";

  /* select database and create test table */

  if (mysql_query (conn, use_stmt) != 0
    || mysql_query (conn, drop_stmt) != 0
    || mysql_query (conn, create_stmt) != 0)
  {
    print_error (conn, "Could not set up test table");
    return;
  }

  stmt = mysql_stmt_init (conn);  /* allocate statement handler */
  if (stmt == NULL)
  {
    print_error (conn, "Could not initialize statement handler");
    return;
  }

  /* insert and retrieve some records */
  insert_rows (stmt);
  select_rows (stmt);

  mysql_stmt_close (stmt);       /* deallocate statement handler */
}
```

First, we select a database and create a test table. The table contains four columns of varying data types: `INT`, `FLOAT`, `CHAR`, and `DATETIME`. These different data types must be handled in slightly different ways, as will become evident.

After creating the table, we invoke `mysql_stmt_init()` to allocate a prepared statement handler, insert and retrieve some rows, and deallocate the handler. All the real work takes place in the `insert_rows()` and `select_rows()` functions, which we will get to shortly. For error handling, the program also uses a function, `print_stmt_error()`, that is similar to the `print_error()` function used in earlier programs but invokes the error functions that are specific to prepared statements:

```
static void
print_stmt_error (MYSQL_STMT *stmt, char *message)
{
  fprintf (stderr, "%s\n", message);
  if (stmt != NULL)
  {
    fprintf (stderr, "Error %u (%s): %s\n",
             mysql_stmt_errno (stmt),
             mysql_stmt_sqlstate (stmt),
             mysql_stmt_error (stmt));
  }
}
```

The `insert_rows()` function takes care of adding new rows to the test table:

```
static void
insert_rows (MYSQL_STMT *stmt)
{
char          *stmt_str = "INSERT INTO t (i,f,c,dt) VALUES(?,?,?,?)";
MYSQL_BIND    param[4];
int           my_int;
float         my_float;
char          my_str[26]; /* ctime() returns 26-character string */
MYSQL_TIME    my_datetime;
unsigned long my_str_length;
time_t        clock;
struct tm     *cur_time;
int           i;

  printf ("Inserting records...\n");

  if (mysql_stmt_prepare (stmt, stmt_str, strlen (stmt_str)) != 0)
  {
    print_stmt_error (stmt, "Could not prepare INSERT statement");
    return;
  }

  /*
   * zero the parameter structures, then perform all parameter
   * initialization that is constant and does not change for each row
   */

  memset ((void *) param, 0, sizeof (param));

  /* set up INT parameter */

  param[0].buffer_type = MYSQL_TYPE_LONG;
  param[0].buffer = (void *) &my_int;
  param[0].is_unsigned = 0;
  param[0].is_null = 0;
  /* buffer_length, length need not be set */

  /* set up FLOAT parameter */

  param[1].buffer_type = MYSQL_TYPE_FLOAT;
  param[1].buffer = (void *) &my_float;
  param[1].is_null = 0;
  /* is_unsigned, buffer_length, length need not be set */

  /* set up CHAR parameter */
```

```
param[2].buffer_type = MYSQL_TYPE_STRING;
param[2].buffer = (void *) my_str;
param[2].buffer_length = sizeof (my_str);
param[2].is_null = 0;
/* is_unsigned need not be set, length is set later */

/* set up DATETIME parameter */

param[3].buffer_type = MYSQL_TYPE_DATETIME;
param[3].buffer = (void *) &my_datetime;
param[3].is_null = 0;
/* is_unsigned, buffer_length, length need not be set */

if (mysql_stmt_bind_param (stmt, param) != 0)
{
  print_stmt_error (stmt, "Could not bind parameters for INSERT");
  return;
}

for (i = 1; i <= 5; i++)
{
  printf ("Inserting record %d...\n", i);

  (void) time (&clock); /* get current time */

  /* set the variables that are associated with each parameter */

  /* param[0]: set my_int value */
  my_int = i;

  /* param[1]: set my_float value */
  my_float = (float) i;

  /* param[2]: set my_str to current ctime() string value */
  /* and set length to point to var that indicates my_str length */
  (void) strcpy (my_str, ctime (&clock));
  my_str[24] = '\0';  /* chop off trailing newline */
  my_str_length = strlen (my_str);
  param[2].length = &my_str_length;

  /* param[3]: set my_datetime to current date and time components */
  cur_time = localtime (&clock);
  my_datetime.year = cur_time->tm_year + 1900;
  my_datetime.month = cur_time->tm_mon + 1;
  my_datetime.day = cur_time->tm_mday;
  my_datetime.hour = cur_time->tm_hour;
  my_datetime.minute = cur_time->tm_min;
```

```
    my_datetime.second = cur_time->tm_sec;
    my_datetime.second_part = 0;
    my_datetime.neg = 0;

    if (mysql_stmt_execute (stmt) != 0)
    {
      print_stmt_error (stmt, "Could not execute statement");
      return;
    }

    sleep (1);  /* pause briefly (to let the time change) */
  }
}
```

The overall purpose of `insert_rows()` is to insert into the test table five rows that contain these values:

- An `INT` value from 1 to 5.
- A `FLOAT` value from 1.0 to 5.0.
- A `CHAR` value. To generate these values, we call the `ctime()` system function to get the value of "now" as a string. `ctime()` returns values that have this format:

  ```
  Sun Sep 19 16:47:23 CDT 2004
  ```

- A `DATETIME` value. This also has the value of "now," but stored in a `MYSQL_TIME` structure. The binary protocol uses `MYSQL_TIME` structures to transmit `DATETIME`, `TIMESTAMP`, `DATE`, and `TIME` values.

The first thing we do in `insert_rows()` is prepare an `INSERT` statement by passing it to `mysql_stmt_prepare()`. The statement looks like this:

```
INSERT INTO t (i,f,c,dt) VALUES(?,?,?,?)
```

The statement contains four placeholders, so it's necessary to supply four data values each time the statement is executed. Placeholders typically represent data values in `VALUES()` lists or in `WHERE` clauses. But there are places in which they cannot be used:

- As identifiers such as table or column names. This statement is illegal:

  ```
  SELECT * FROM ?
  ```

- You can use placeholders on one side of an operator, but not on both sides. This statement is legal:

  ```
  SELECT * FROM student WHERE student_id = ?
  ```

  However, this statement is illegal:

  ```
  SELECT * FROM student WHERE ? = ?
  ```

  This restriction is necessary so that the server can determine parameter data types.

The next step is to set up an array of MYSQL_BIND structures, one for each placeholder. As demonstrated in insert_rows(), setting these up involves two stages:

1. Initialize all parts of the structures that will be the same for each row inserted.
2. Perform a row-insertion loop that, for each row, initializes the parts of the structures that vary for each row.

You could actually perform all initialization within the loop, but that would be less efficient.

The first initialization stage begins by zeroing the contents of the param array containing the MYSQL_BIND structures. The program uses memset(), but you can use bzero() if your system doesn't have memset(). These two statements are equivalent:

```
memset ((void *) param, 0, sizeof (param));
bzero ((void *) param, sizeof (param));
```

Clearing the param array implicitly sets all structure members to zero. Code that follows sets some members to zero to make it explicit what's going on, but that is not strictly necessary. In practice, you need not assign zero to any structure members after clearing the structures.

The next step is to assign the proper information to each parameter in the MYSQL_BIND array. For each parameter, the structure members that need to be set depend on the type of value you're transmitting:

- The buffer_type member always must be set; it indicates the data type of the value. Appendix G, "C API Reference," contains a table that lists each of the permitted type codes and shows the SQL and C types that correspond to each code.
- The buffer member should be set to the address of the variable that contains the data value. insert_rows() declares four variables to hold row values: my_int, my_float, my_str, and my_datetime. Each param[i].buffer value is set to point to the appropriate variable. When it comes time to insert a row, we set these four variables to the table column values and they are used to create the new row.
- The is_unsigned member applies only to integer data types. It should be set to true (nonzero) or false (zero) to indicate whether the parameter corresponds to an UNSIGNED integer type. Our table contains a signed INT column, so we set is_unsigned to zero. Were the column an INT UNSIGNED, we would set is_unsigned to 1, and would also declare my_int as unsigned int rather than as int.
- The is_null member indicates whether you're transmitting a NULL value. In the general case, you set this member to the address of a my_bool variable. Then, before inserting any given row, you set the variable true or false to specify whether the value to be inserted is NULL. If no NULL values are to be sent (as is the case here), you can set is_null to zero and no my_bool variable is needed.
- For character string values or binary data (BLOB values), two more MYSQL_BIND members come into play. These indicate the size of the buffer in which the value is stored and the actual size of the current value being transmitted. In many cases these might be the same, but they will differ if you're using a fixed-size buffer and sending values that vary

in length from row to row. buffer_length indicates the size of the buffer. length is a pointer; it should be set to the address of an unsigned long variable that contains the actual length of the value to be sent.

For numeric and temporal data types, buffer_length and length need not be set. The size of each of these types is fixed and can be determined from the buffer_type value. For example, MYSQL_TYPE_LONG and MYSQL_TYPE_FLOAT indicate 4-byte and 8-byte values.

After the initial setup of the MYSQL_BIND array has been done, we bind the array to the prepared statement by passing the array to mysql_stmt_bind_param(). Then it's time to assign values to the variables that the MYSQL_BIND structures point to and execute the statement. This takes place in a loop that executes five times. Each iteration of the loop assigns values to the statement parameters:

- For the integer and floating-point parameters, it's necessary only to assign values to the associated int and float variables.
- For the string parameter, we assign the current time in string format to the associated char buffer. This value is obtained by calling ctime(), and then chopping off the newline character.
- The datetime parameter also is assigned the current time, but this is done by assigning the component parts of the time to the individual members of the associated MYSQL_TIME structure.

With the parameter values set, we execute the statement by invoking mysql_stmt_execute(). This function transmits the current values to the server, which incorporates them into the prepared statement and executes it.

When insert_rows() returns, the test table has been populated and select_rows() can be called to retrieve them:

```
static void
select_rows (MYSQL_STMT *stmt)
{
char          *stmt_str = "SELECT i, f, c, dt FROM t";
MYSQL_BIND    param[4];
int           my_int;
float         my_float;
char          my_str[24];
unsigned long my_str_length;
MYSQL_TIME    my_datetime;
my_bool       is_null[4];

  printf ("Retrieving records...\n");

  if (mysql_stmt_prepare (stmt, stmt_str, strlen (stmt_str)) != 0)
  {
```

```
    print_stmt_error (stmt, "Could not prepare SELECT statement");
    return;
  }

  if (mysql_stmt_field_count (stmt) != 4)
  {
    print_stmt_error (stmt, "Unexpected column count from SELECT");
    return;
  }

  /*
   * initialize the result column structures
   */

  memset ((void *) param, 0, sizeof (param)); /* zero the structures */

  /* set up INT parameter */

  param[0].buffer_type = MYSQL_TYPE_LONG;
  param[0].buffer = (void *) &my_int;
  param[0].is_unsigned = 0;
  param[0].is_null = &is_null[0];
  /* buffer_length, length need not be set */

  /* set up FLOAT parameter */

  param[1].buffer_type = MYSQL_TYPE_FLOAT;
  param[1].buffer = (void *) &my_float;
  param[1].is_null = &is_null[1];
  /* is_unsigned, buffer_length, length need not be set */

  /* set up CHAR parameter */

  param[2].buffer_type = MYSQL_TYPE_STRING;
  param[2].buffer = (void *) my_str;
  param[2].buffer_length = sizeof (my_str);
  param[2].length = &my_str_length;
  param[2].is_null = &is_null[2];
  /* is_unsigned need not be set */

  /* set up DATETIME parameter */

  param[3].buffer_type = MYSQL_TYPE_DATETIME;
  param[3].buffer = (void *) &my_datetime;
  param[3].is_null = &is_null[3];
  /* is_unsigned, buffer_length, length need not be set */
```

```
  if (mysql_stmt_bind_result (stmt, param) != 0)
  {
    print_stmt_error (stmt, "Could not bind parameters for SELECT");
    return;
  }

  if (mysql_stmt_execute (stmt) != 0)
  {
    print_stmt_error (stmt, "Could not execute SELECT");
    return;
  }

  /*
   * fetch result set into client memory; this is optional, but it
   * enables mysql_stmt_num_rows() to be called to determine the
   * number of rows in the result set.
   */

  if (mysql_stmt_store_result (stmt) != 0)
  {
    print_stmt_error (stmt, "Could not buffer result set");
    return;
  }
  else
  {
    /* mysql_stmt_store_result() makes row count available */
    printf ("Number of rows retrieved: %lu\n",
            (unsigned long) mysql_stmt_num_rows (stmt));
  }

  while (mysql_stmt_fetch (stmt) == 0)  /* fetch each row */
  {
    /* display row values */
    printf ("%d  ", my_int);
    printf ("%.2f  ", my_float);
    printf ("%*.*s  ", (int) my_str_length, (int) my_str_length, my_str);
    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
            my_datetime.year,
            my_datetime.month,
            my_datetime.day,
            my_datetime.hour,
            my_datetime.minute,
            my_datetime.second);
  }

  mysql_stmt_free_result (stmt);      /* deallocate result set */
}
```

`select_rows()` prepares a `SELECT` statement, executes it, and retrieves the result. In this case, the statement contains no placeholders:

```
SELECT i, f, c, dt FROM t
```

That means it's unnecessary to set up any `MYSQL_BIND` structures before executing the statement. But we're not off the hook. The bulk of the work in `select_rows()`, just as in `insert_rows()`, is setting up an array of `MYSQL_BIND` structures. The difference is that they're used to receive data values from the server *after* executing the statement rather than to set up data values to be sent to the server *before* executing the statement.

Nevertheless, the procedure for setting up the `MYSQL_BIND` array is somewhat similar to the corresponding code in `insert_rows()`:

1. Zero the array.
2. Set the `buffer_type` member of each parameter to the appropriate type code.
3. Point the `buffer` member of each parameter to the variable where the corresponding column value should be stored when rows are fetched.
4. Set the `is_unsigned` member for the integer parameter to zero.
5. For the string parameter, set the `buffer_length` value to the maximum number of bytes that should be fetched, and set `length` to the address of an `unsigned long` variable. At fetch time, this variable is set to the actual number of bytes fetched.
6. For every parameter, set the `is_null` member to the address of a `my_bool` variable. At fetch time, these variables are set to indicate whether the fetched values are `NULL`. (Our program ignores these variables after fetching rows because we know that the test table contains no `NULL` values. In the general case, you should check them.)

After setting up the parameters, we bind the array to the statement by calling `mysql_stmt_bind_result()`, then execute the statement.

At this point, you can immediately begin fetching rows by calling `mysql_stmt_fetch()`. Our program demonstrates an optional step that you can do first: It calls `mysql_stmt_store_result()`, which fetches the entire result set and buffers it in client memory. The advantage of doing this is that you can call `mysql_stmt_num_rows()` to find out how many rows are in the result set. The disadvantage is that it uses more memory on the client side.

The row-fetching loop involves calling `mysql_stmt_fetch()` until it returns a nonzero value. After each fetch, the variables associated with the parameter structures contain the column values for the current row.

Once all the rows have been fetched, a call to `mysql_stmt_free_result()` releases any memory associated with the result set.

At this point, `select_rows()` returns to the caller, which invokes `mysql_stmt_close()` to dispose of the prepared statement handler.

The preceding discussion provides a broad overview of the prepared statement interface and some of its key functions. The client library includes several other related functions; for more information, consult Appendix G, "C API Reference."

7.9 Using Prepared CALL Support

Prepared statement support got a boost in MySQL 5.5 with improved handling of prepared CALL statements to invoke stored procedures, including the capability of accessing the returned values of OUT and INOUT procedure parameters. Previously, prepared CALL statements could not produce multiple result sets and callers could not access returned parameter values.

For demonstration purposes, assume that the following stored procedure exists:

```
CREATE PROCEDURE grade_event_stats
  (IN p_event_id INT, OUT p_min INT, OUT p_max INT)
BEGIN
  -- display scores for event
  SELECT student_id, score
    FROM score
    WHERE event_id = p_event_id
    ORDER BY student_id;
  -- store min/max event scores in OUT parameters
  SELECT MIN(score), MAX(score)
    FROM score
    WHERE event_id = p_event_id
    INTO p_min, p_max;
END;
```

The procedure takes one IN parameter and two OUT parameters. Given a grade event ID as the its parameter, the procedure displays the scores for that event, and returns the minimum and maximum scores for the event in its two OUT parameters. If we were to call grade_event_stats() from the mysql client, the statements might look like this:

```
mysql> SET @p_min = NULL, @p_max = NULL;
mysql> CALL grade_event_stats(4, @p_min, @p_max);
+------------+-------+
| student_id | score |
+------------+-------+
|          2 |     7 |
|          3 |    17 |
|          4 |    16 |
|          5 |    20 |
...
mysql> SELECT @p_min, @p_max;
+--------+--------+
| @p_min | @p_max |
+--------+--------+
```

```
|      7 |     20 |
+--------+--------+
```

The following discussion shows how to use the C API to do the same thing. The source for this program is the `prepared_call.c` file in the `capi` directory of the `sampdb` distribution. To create the procedure, use `prepared_call_setup.sql` in that same directory:

```
% mysql sampdb < prepared_call_setup.sql
```

Prepared `CALL` support requires MySQL 5.5.3 or higher. To verify after connecting to a server that it has this capability, `prepared_call` checks the server version:

```
if (mysql_get_server_version (conn) < 50503)
{
  print_error (NULL, "Prepared CALL requires MySQL 5.5.3 or higher");
  mysql_close (conn);
  exit (1);
}
```

The next part of the program initializes a prepared statement handler and uses it to execute a prepared `CALL` statement. If that succeeds, it processes the procedure result:

```
stmt = mysql_stmt_init (conn);
if (!stmt)
  print_error (NULL, "Could not initialize statement handler");
else
{
  if (exec_prepared_call (stmt) == 0)
    process_call_result (conn, stmt);
  mysql_stmt_close (stmt);
}
```

Preparing a `CALL` statement is much like preparing other statements. Pass the statement string to `mysql_stmt_prepare()`, with data values represented by '?' placeholder characters. For `CALL`, the placeholders represent parameter values passed to the procedure. Parameter setup is necessarily specific to the number and types of parameters a procedure takes. For simplicity, all `grade_event_stats()` parameters are integers.

```
static int exec_prepared_call (MYSQL_STMT *stmt)
{
MYSQL_BIND params[3];   /* parameter buffers */
int        int_data[3]; /* parameter values */
int        i;

  /* prepare CALL statement */
  if (mysql_stmt_prepare (stmt, "CALL grade_event_stats(?, ?, ?)", 31))
  {
    print_stmt_error (stmt, "Cannot prepare statement");
    return (1);
```

```
  }

  /* initialize parameter structures and bind to statement */
  memset (params, 0, sizeof (params));

  for (i = 0; i < 3; ++i)
  {
    params[i].buffer_type = MYSQL_TYPE_LONG;
    params[i].buffer = (char *) &int_data[i];
    params[i].length = 0;
    params[i].is_null = 0;
  }

  if (mysql_stmt_bind_param (stmt, params))
  {
    print_stmt_error (stmt, "Cannot bind parameters");
    return (1);
  }

  /* assign parameter values and execute statement */
  int_data[0]= 4;  /* p_event_id */
  int_data[1]= 0;  /* p_min (OUT param; initial value ignored by procedure */
  int_data[2]= 0;  /* p_min (OUT param; initial value ignored by procedure */

  if (mysql_stmt_execute (stmt))
  {
    print_stmt_error (stmt, "Cannot execute statement");
    return (1);
  }
  return (0);
}
```

After executing the CALL statement, process its results, which can have several parts:

1. Statements executed within the procedure can each produce a result set. This includes statements such as SELECT, SHOW, and so forth.
2. If the procedure has any OUT or INOUT parameters, there is an additional single-row result set containing the final parameter values, in the order they appear in the procedure definition.
3. A final status packet. There is no result set associated with this, so you can distinguish it from the statement or parameter result sets because it has a result column count of zero.

All parts of the CALL result that are present must be processed. The statement and parameter result sets are optional, depending on how the procedure is written. The procedure always returns the final status packet, regardless of whether there are any preceding result sets.

The `process_call_result()` function shows a general-purpose loop that retrieves procedure results. The code makes no assumptions about whether the procedure produces any statement or parameter result sets. The retrieval loop has this logic:

1. Get the column count of the next result to determine whether it contains a result set or is the final status packet. A count of zero signifies the status packet, which requires no further processing.
2. A positive column count signifies a result set with that many columns. The result set could either be produced by a statement or contain the returned procedure parameter values. The connection handler has a status member containing a flag you can use to distinguish the two cases. (`process_call_result()` uses the flag only for information purposes, to announce for each result set how it was produced.) In either case, fetch the result set.
3. Check whether there are more results by calling `mysql_stmt_next_result()`. This function returns zero if more results are available, –1 if not, and a value greater than zero if an error occurred. Therefore, if the result is zero, return to step 1 to get the next result.

```
static void process_call_result (MYSQL *conn, MYSQL_STMT *stmt)
{
int status;
int num_cols;

  /*
   * For each result, check number of columns.  If none, the result is
   * the final status packet and there is nothing to do. Otherwise,
   * fetch the result set.
   */
  do {
    if ((num_cols = mysql_stmt_field_count (stmt)) > 0)
    {
      /* announce whether result set contains parameters or data set */
      if (conn->server_status & SERVER_PS_OUT_PARAMS)
        printf ("OUT/INOUT parameter values:\n");
      else
        printf ("Statement result set values:\n");

      if (process_result_set (stmt, num_cols))
        break; /* some error occurred */
    }

    /* status is -1 = done, 0 = more results, >0 = error */
    status = mysql_stmt_next_result (stmt);
    if (status > 0)
      print_stmt_error (stmt, "Error checking for next result");
  } while (status == 0);
}
```

To fetch the statement and parameter result sets, the loop calls `process_result_set()`, not shown. Check the `prepared_call.c` source to see what it does. For simplicity, it assumes that any retrieved values are integers. You'd modify that to be more general for programs that handle other data types.

Here is the output from running the `prepared_call` program:

```
% ./prepared_call sampdb
Statement result set values:
 val[1] = 2; val[2] = 7;
 val[1] = 3; val[2] = 17;
 val[1] = 4; val[2] = 16;
 val[1] = 5; val[2] = 20;
...
OUT/INOUT parameter values:
 val[1] = 7; val[2] = 20;
```

Take care to account for any differences between the positions of parameters in the procedure definition and their positions in the parameter result set. Observe that the preceding output says the returned parameter values have positions 1 and 2, although they are actually parameters 2 and 3 to procedure `grade_event_stats()`. Why is this? Because the final result set that returns the parameter values includes only `OUT` and `INOUT` parameters, not `IN` parameters. Any changes to `IN` parameters within the procedure are not seen by the caller. Therefore, they have the same values after the call as before and need not be returned in the procedure results.