Paul DuBois

Fourth Edition

MySQL®

Developer's Library

MySQL®

Fourth Edition

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

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

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

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

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

> **`load_defaults()` and Security**
>
> On multiple-user systems, utilities such as the `ps` program can display argument lists 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`. This is one reason why it's not a good idea to specify passwords that way. One precaution a program can take to help reduce the risk is to remove the password from the argument list as soon as it starts executing. 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 now 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-`, 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, and 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, and then prints the resulting vector.
4. Calls the option processing routine `handle_options()` to process the argument vector, and 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);
}
```

> **Note**
>
> The source code for `show_opt.c` and several other programs later in this chapter uses the `uchar**` type in MySQL-related data structures. Before MySQL 5.1.20, you'll find that the MySQL header files use `gptr*`, which results in warnings when you compile the programs. You can ignore these warnings.

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

The `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 */
  uchar      **value;             /* pointer to variable to store value in */
  uchar      **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 */
};
```