API C MySQL

Aplicaciones C/C++ con bases de datos

Salvador Pozo

http://conclase.net

Prólogo

MySQL *es una marca registrada por* ***MySQL AB****. Parte del material que se expone aquí, concretamente las referencias de funciones del API de* ***MySQL*** *y de la sintaxis de SQL, son traducciones del manual original de* ***MySQL*** *que se puede encontrar en inglés en* www.mysql.com*.*

Este sitio está desarrollado exclusivamente por los componentes de ***Con Clase****, cualquier error en la presente documentación es sólo culpa nuestra.*

Por otra parte, MySQL AB fue adquirida por Sun Microsystems en 2008, y ésta a su vez fue comprada por Oracle Corporation en 2010, sigue siendo de código abierto, y tiene licencia dual, pero posee derechos de copia para la mayor parte de su código.

Existe otro motor de base de datos, 

, derivado de MySQL con licencia GPL. Toda la documentación que aparece en Con Clase es válida para ambos motores, y puedes usar cualquiera de ellos indistintamente.

Introducción

El presente manual trata sobre la integración de bases de datos **MySQL** dentro de aplicaciones C y C++. Para ello nos limitaremos a explicar algunas de las funciones y estructuras del API de **MySQL** y al modo en que estas se combinan para crear programas que trabajen con bases de datos.

Se supone que el lector de este documento ya tiene conocimientos sobre diseño de bases de datos y sobre el lenguaje de consulta SQL. De todos modos, en **Con Clase** también se incluye un curso sobre estos temas, al que se puede acceder desde MySQL con Clase.

También supondremos que trabajamos en un sistema que tiene instalado el servidor **MySQL**. Se puede consultar la documentación en la página de **MySQL** para obtener detalles sobre el modo de instalar este servidor en distintos sistemas operativos.

Instalación de bibliotecas para Dev-C++

Lo que sí necesitamos es, por supuesto, un compilador de C/C++. Los ejemplos que aparecen en este curso están escritos en C++, pero usaremos el API C, de modo que las estructuras de los programas deben ser fácilmente adaptables a código C.

Nota:

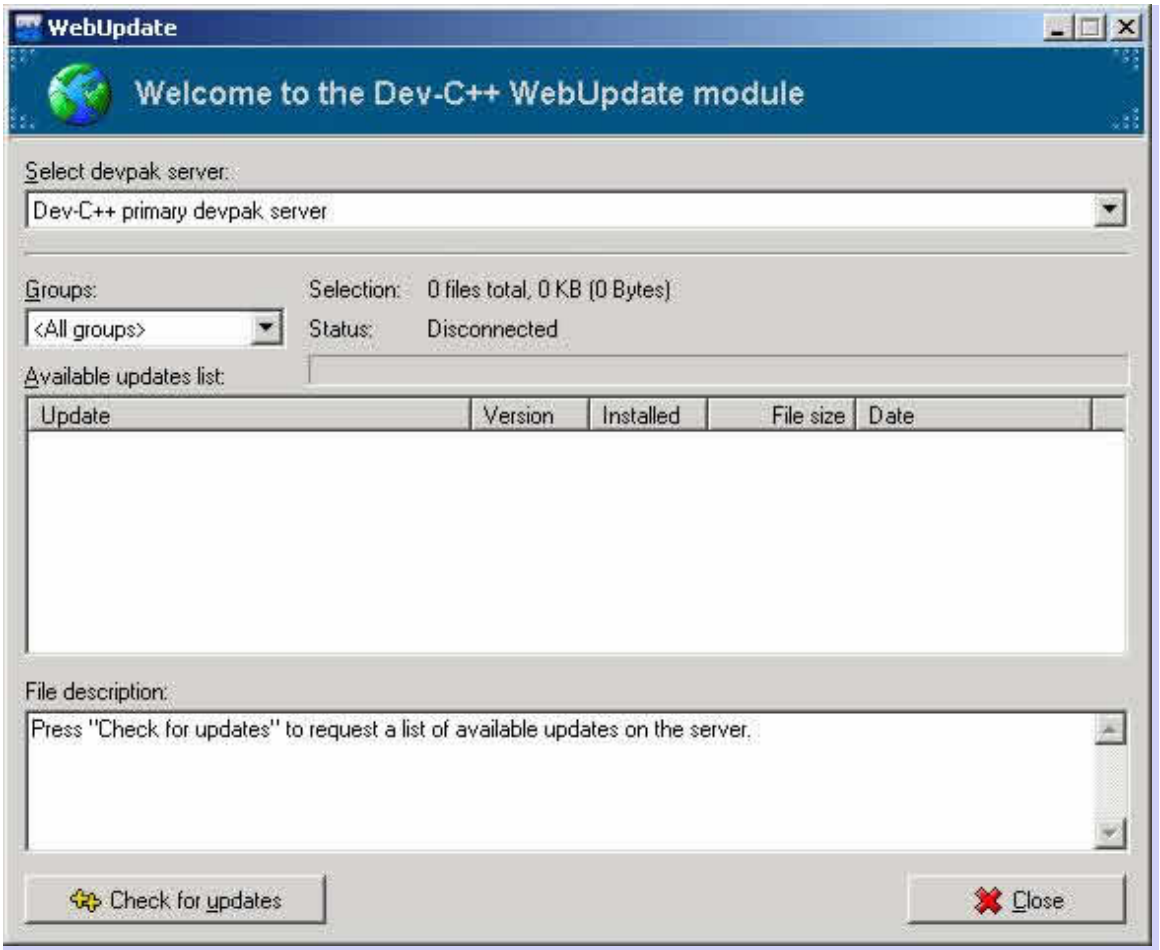
Aunque existe un API C++, que encapsula el API C de **MySQL**, de momento no se incluye documentación sobre él en este curso.

Siguiendo lo que ya es una tradición en **Con Clase** usaremos el compilador **Mingw** y el entorno de desarrollo **Dev-C++** para crear nuestros programas de ejemplo. En este entorno se puede descargar un paquete para usar el API de **MySQL**. Veamos cómo hacerlo paso a paso:

1. Usar la opción *Ayuda->Sobre...*, se mostrará este cuadro de diálogo:

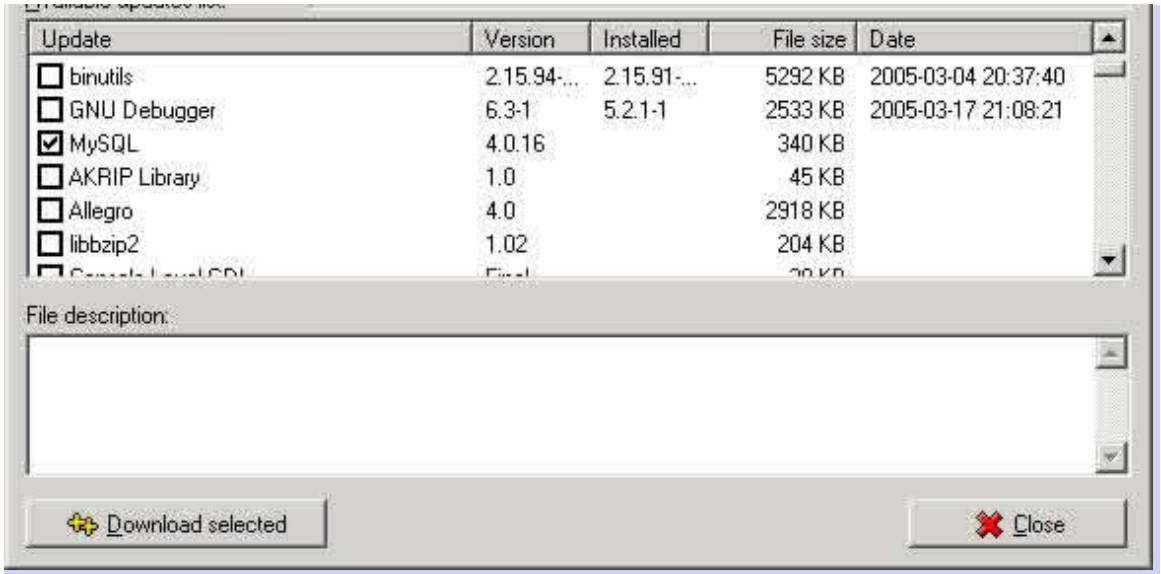
Diálogo sobre

2. Pulsar el botón "Buscar actualizaciones", se mostrará el cuadro de diálogo de actualizaciones:



Diálogo de actualización

3. Pulsar el botón "Check for updates". Se leerá una lista de los paquetes disponibles.
4. Seleccionar el paquete **MySQL** y pulsar el botón "Download selected":



Activar opción MySQL

5. Se descargará el paquete desde Internet, y aparecerá este mensaje: "The updates you selected have been downloaded. Now they will be installed.", es decir, que se han descargado las actualizaciones y se procederá a instalarlas. Pulsamos "Ok".
6. Se abrirá otra ventana, con el "Installation Wizard", pulsamos "Install":

Installation Wizard

7. Cuando termine la instalación pulsamos "Finish", y se cerrará el "Installation Wizard".

Ahora tendremos un subdirectorio *MySQL* bajo el directorio *include* de *Dev-C++*, que contendrá los ficheros de cabecera ".h". En el directorio *lib* de **Dev-C++** se habrán copiado los ficheros "libmySQL.a" y "libmySQL.def". Y en el directorio de *examples* de **Dev-C++** habrá una carpeta con un proyecto de ejemplo *MySQLClientTest*, (que no funcionará :-), aunque esto no debe preocuparnos, ya que tampoco lo vamos a necesitar).

Usar ficheros incluidos con MySQL

Pero en realidad, aunque esto funcione, lo cierto es que junto con la instalación del servidor **MySQL** ya existen los ficheros de cabecera y las bibliotecas necesarias para usar **MySQL** en nuestros programas *C/C++*.

El método que vamos a explicar tiene la ventaja de que usaremos las bibliotecas adecuadas para la versión de **MySQL** que tenemos instalada. Si usamos el método del paquete de **Dev-C++** probablemente estaremos usando bibliotecas de una versión anterior. Además, funciona con cualquier IDE que use como compilador **MinGW**, como por ejemplo, **Code::Blocks**.

Los ficheros de cabecera están en el directorio *include* dentro del directorio de instalación de **MySQL**, que dependerá de dónde lo hayas instalado. Esos ficheros pueden ser copiados al directorio a un directorio *mysql* dentro del directorio *include* de **Dev-C++**, o en su caso, en la carpeta *include* de **MinGW**.

Además, en el directorio *lib\opt* de **MySQL** existen varios ficheros de biblioteca, y una biblioteca de enlace dinámico *libmysql.dll*. Copiaremos esta biblioteca a un directorio donde sea accesible, por ejemplo al directorio *system32* de **Windows**.

De los ficheros de bibliotecas estáticas sólo nos interesa uno: *libmysql.lib*.

Pero **Mingw** no puede usar esa biblioteca directamente. Este compilador usa otro formato de bibliotecas estáticas, que además de tener la extensión ".a", tienen un formato interno diferente. Deberemos, por lo tanto, convertir la biblioteca al formato adecuado.

Para poder hacerlo necesitamos los siguientes ficheros:

- El fichero de biblioteca dinámica "dll": *libmysql.dll*.
- El fichero de biblioteca estática "lib": *libmysql.lib*.
- El fichero de definición de biblioteca "def": *libmysql.def*.

Y estas tres utilidades:

- Utilidad *reimp.exe*.
- Utilidad *dlltool.exe*.
- Utilidad *as.exe*.

Los dos primeros ficheros ya los tenemos, los encontraremos en el directorio *lib\opt* de modo que para trabajar más fácilmente, los copiaremos a un directorio temporal de trabajo, por ejemplo, *C:\mysqltmp*.

Para obtener el tercero necesitamos una utilidad de **Mingw** llamada *reimp*. Esta utilidad sirve para extraer ficheros de definición de bibliotecas "lib".

Para conseguir la herramienta *reimp* hay que descargar el fichero Utilidades mingw, que contiene algunas utilidades, pero del que sólo nos interesa el fichero *reimp.exe*.

Nota:

Por lo que he podido comprobar, las últimas versiones de MinGW incluyen el programa reimp.exe en la carpeta "bin". Por otra parte, las últimas versiones de MySQL, también contienen el fichero "libmysql.def" en la carpeta "include", pero no se

puede utilizar porque el formato interno no es adecuado para nuestros propósitos.

Copiaremos el fichero *reimp.exe* al directorio de trabajo temporal, y ejecutamos esta sentencia desde la línea de comandos:

```
C:\mysqltmp> reimp -d libmysql.lib
```

Esto genera el fichero *libmysql.def*. El siguiente paso es crear el fichero de biblioteca estática *libmysql.a*. Para ello necesitamos las otras dos utilidades, *dlltool.exe* y *as.exe* que están incluidan con el compilador *Mingw*, y que podemos encontrar en el subdirectorio *mingw32\bin* del directorio donde esté instalado **Dev-C++** o **Code::Blocks**. Para comodidad, copiaremos estas utilidades al directorio temporal de trabajo, y ejecutaremos la siguiente sentencia desde la línea de comandos:

```
C:\mysqltmp>dlltool -d libmysql.def -D libmysql.dll -k -l
libmysql.a -S ./as.exe
```

La utilidad *as.exe* se invoca automáticamente desde *dlltool*. Ahora ya tenemos nuestro fichero de biblioteca estática *libmysql.a*, y lo copiaremos al directorio *lib* de **Dev-C++**.

Por último, podemos borrar el directorio de trabajo temporal. Y ya podemos usar bases de datos desde programas C/C++.

Este proceso es válido para convertir cualquier biblioteca en formato "lib" a su equivalente en formato "a".

Usar bibliotecas desde otros compiladores

No lo he verificado, pero es de suponer que otros compiladores usarán, o bien las bibliotecas estáticas con extensión "a" si se basan en **Mingw**, o las que tienen extensión "lib", como los compiladores de **Borland** o **Microsoft**.

En cualquier caso, si quieres compartir tus experiencias en estos compiladores con nosotros, podremos ampliar este apartado.

1 Conectar y desconectar

Siempre, cada vez que queramos usar **MySQL** en una de nuestras aplicaciones, deberemos realizar algunas tareas antes y después de acceder al servidor. Esto es bastante corriente cuando se trabaja con "motores", ya sean gráficos, de bases de datos, o de lo que sea.

Primero hay que verificar que el motor está presente y en ejecución. Después hay que establecer una conexión o canal que sirva para mantener un diálogo con el servidor. A partir de ahí podremos entablar comunicación con ese servidor, y cuando hayamos terminado, cerrar la conexión.

Iniciar y conectar con el servidor MySQL

Lo primero es iniciar un objeto del tipo MYSQL. Este objeto se usa por el API para mantener las variables de la conexión con el motor de **MySQL**.

Para iniciar uno de estos objetos usamos la función mysql_init(). Esta función sólo necesita un parámetro, que es un puntero a un objeto de tipo MYSQL. Si ya disponemos de un objeto de ese tipo podemos pasar su dirección como parámetro, y la función lo iniciará. Si no disponemos de tal objeto, pasaremos un puntero nulo y la función lo creará dinámicamente y nos devolverá un puntero al objeto.

Por ejemplo, usando un objeto dinámico:

```
MYSQL *myData;

// Intentar iniciar MySQL:
if(!(myData = mysql_init(0))) {
   // Imposible crear el objeto myData
```

```
    return 1;
}
```

Y usando un objeto estático:

```
MYSQL myData;

 // Iniciar MySQL:
 mysql_init((MYSQL*)&myData));
```

Generalmente trabajaremos con un objeto MYSQL dinámico, al menos en los ejemplos de este curso.

Establecer una conexión

Una vez hemos inicializado el objeto MYSQL, intentaremos establecer una conexión con el servidor. Para ello usaremos la función mysql_real_connect().

Esta función necesita muchos parámetros, concretamente ocho. Veamos qué significa cada uno de ellos:

1. Un puntero a un objeto MYSQL, que previamente tendremos que haber iniciado con la función mysql_init().
2. El nombre del ordenador donde se está ejecutando el servidor. Puede ser también una dirección IP. Si el servidor está ejecutándose en la misma máquina que la aplicación, este nombre puede ser "localhost", o simplemente, un puntero nulo.
3. El nombre del usuario. En sistemas **Unix** puede ser un puntero nulo, para indicar el usuario actual. En **Windows ODBC** debe especificarse.
4. La contraseña del usuario seleccionado.
5. Base de datos por defecto. Puede ser NULL si no queremos usar una base de datos determinada.
6. Número de puerto. Generalmente usaremos la constante **MYSQL_PORT**.
7. Socket **Unix**. Usaremos NULL para conexiones locales.

8. Opciones de cliente, normalmente 0, pero se puede usar una combinación de ciertos valores para activar algunas opciones. Ver sintaxis de mysql_real_connect().

Si no se puede establecer una conexión, mysql_real_connect() devuelve el valor NULL.

Al menos al principio usaremos conexiones locales para acceder a las bases de datos, aunque más tarde veremos que no hay problema para hacer conexiones a través de una red local o a través de Internet.

Crearemos un usuario para nuestros experimentos, que sólo tenga acceso total a la base de datos "prueba":

```
mysql> GRANT ALL ON prueba.* TO curso IDENTIFIED BY 'clave';
Query OK, 0 rows affected (0.16 sec)
```

Esto hace que los parámetros a usar sean más sencillos. Una llamada típica para acceso local puede ser:

```
   if(!mysql_real_connect(myData, NULL, "curso", "clave",
"prueba", MYSQL_PORT, NULL, 0)) {
      // No se puede conectar con el servidor en el puerto
especificado.
      cout << "Imposible conectar con servidor mysql en el
puerto "
           << MYSQL_PORT << endl;
      mysql_close(myData);
      return 1;
   }
```

Cerrar

Cuando hayamos terminado de trabajar con las bases de datos cerramos la conexión con el motor de bases de datos, para eso usamos la función mysql_close().

```
// Cerrar la conexión
mysql_close(myData);
```

Cerrar la conexión tiene un doble propósito. Por una parte, se cierra la conexión abierta con el servidor mediante la función mysql_real_connect(). Por otra, se libera la memoria correspondiente al objeto MYSQL, si la función mysql_init() fue invocada con un puntero nulo.

Ahora ya estamos en condiciones de empezar a trabajar con bases de datos.

Reconexiones

Un problema frecuente cuando se trabaja con **MySQL** es que las conexiones abiertas que permanecen mucho tiempo inactivas pueden cerrarse de forma automática. Esto no es un defecto, sino algo habitual con conexiones de red. Las conexiones "débiles" (con poco tráfico) se cierran de forma automática pasado un tiempo determinado, para ahorrar recursos.

Esto pasa también con la consola de **MySQL**, pero al intentar nuevas consultas se realiza una reconexión automática. Esto no será así con nuestras aplicaciones, ya que si la conexión se cierra recibiremos un mensaje de error, y la conexión no se restablecerá.

En el API existe una función para verificar si la conexión sigue abierta y reconectar en caso necesario.

La función es mysql_ping(), y devuelve un valor cero si la conexión está activa.

```
if(mysql_ping(&myData)) {
   cout << "Error: conexión imposible" << endl;
   mysql_close(&myData);
}
```

Es recomendable hacer uso de esta función si existe la posibilidad de que la conexión haya estado inactiva durante mucho tiempo, por ejemplo, en aplicaciones **Windows** que pueden estar en segundo plano durante un tiempo indeterminado.

2 Consultas

Ahora que ya sabemos cómo establecer una conexión con el servidor **MySQL** y cómo cerrar esa conexión, veamos cómo podemos hacer consultas SQL desde nuestros programas C/C++.

Seleccionar una base de datos

Para acceder a la información almacenada en una base de datos, debemos hacer referencia a la base de datos mediante su nombre. En el cliente **MySQL** usamos la orden USE para seleccionar una base de datos por defecto.

Usando el API hemos visto que podemos seleccionar la base de datos en la llamada a la función mysql_real_connect(). Pero si queremos cambiar la base de datos por defecto durante la ejecución no será necesario cerrar la conexión y abrir una nueva, podemos usar en su lugar la función mysql_select_db():

```
if(mysql_select_db(myData, "prueba")) {
   // Error al seleccionar base de datos.
   cout << "ERROR: " << mysql_error(myData) << endl;
   mysql_close(myData);
   rewind(stdin);
   getchar();
   return 2;
}
```

La función mysql_select_db() hace que la base de datos especificada como segundo parámetro sea considerada la base de datos por defecto en las siguientes consultas. En rigor esto no es imprescindible, ya que podemos acceder a una tabla mediante el especificador de base de datos y el nombre de la tabla, pero si todas

las consultas se van a hacer sobre la misma base de datos, esto nos ahorrará mucho trabajo (de escritura).

Si el valor de retorno de la función es distinto de cero, indica que se ha producido un error. En este ejemplo hemos usardo la función mysql_error para mostrar un mensaje de error que indique el motivo.

Esta función puede fallar por varios motivos, pero de momento sólo nos preocupa si el error es provocado porque la base de datos no existe, en ese caso obtendremos el error ER_BAD_DB_ERROR.

Seleccionar datos de una tabla

La primera operación que intentaremos sobre la base de datos es leer algunos registros de una tabla. Empezaremos por una consulta sencilla, leyendo todos los registros de la tabla 'gente'.

Nota:

Supondremos que tal tabla ya existe y que contiene datos, pero si no es así en tu caso, puedes crearla y añadir datos desde la consola siguiendo los pasos del curso de **MySQL**: crear base de datos, crear una tabla, insertar filas.

La consulta que vamos a hacer es:

```
SELECT * FROM gente;
```

Necesitamos usar una sentencia SQL, en este caso una sentencia SELECT.

Para incluir consultas dentro de programas C/C++ se usan dos funciones del API C de **MySQL**, mysql_query o mysql_real_query. En ambos casos, primer parámetro es, como siempre, un manipulador de un objeto MYSQL iniciado; el segundo contiene la cadena con la consulta o instrucción SQL a ejecutar.

La función mysql_real_query usa un tercer parámetro, que el la longitud de la cadena que contiene la consulta. Esta función permite usar consultas binarias, es decir, que la cadena de la consulta puede contener caracteres nulos.

```
mysql_query(myData, "SELECT * FROM gente");
```

Veamos un ejemplo de cómo lanzar esta consulta:

```
   // Hacer una consulta con el comando "SELECT * FROM
gente":
   if(mysql_query(myData, "SELECT * FROM gente")) {
      // Error al realizar la consulta:
      cout << "ERROR: " << mysql_error(myData) << endl;
      mysql_close(myData);
      rewind(stdin);
      getchar();
      return 2;
   }
```

Recuperar datos de una base de datos

Ya hemos hecho una consulta mediante SELECT, ahora mostraresmos el proceso para recuperar los resultados de esa consulta. Por cierto, otras consultas también generan conjuntos de resultados, como SHOW_TABLES o SHOW DATABASES, y el tratamiento de esos resultados es el mismo.

Lo primero es recuperar el conjunto de resultados procedente del servidor **MySQL**, para ello se usa la función mysql_store_result.

La función mysql_store_result requiere como parámetro un objeto MYSQL, con la conexión actual y devuelve un puntero a una estructura MYSQL_RES.

Del mismo modo, una vez procesados los resultados, se deben liberar mediante una llamada a la función mysql_free_result. Esto

liberará la memoria usada por el conjunto de resultados, es decir, por la estructura MYSQL_RES.

```
// Almacenar el resultado de la consulta, lo necesitaremos
después:
if((res = mysql_store_result(myData))) {
   // Procesar resultados
   ...
   // Liberar el resultado de la consulta:
   mysql_free_result(res);
}
```

Una vez obtenido el conjunto de resultados, disponemos de varias funciones que podemos aplicar sobre él.

Procesar un conjunto de resultados

Hay un conjunto de funciones que se aplican a un conjunto de resultados para obtener información sobre él.

Número de elementos de un conjunto de resultados

Una de las primeras cosas que puede interesarnos es saber cuantos elementos contiene el conjunto de resultados. Para obtener ese valor se puede usar la función mysql_num_rows.

Número de columnas por fila de un conjunto de resultados

Otro parámetro interesante es el número de columnas por fila, para obtenerlo se puede usar la función mysql_num_fields.

Ambas funciones requieren como parámetro un puntero a una estructura MYSQL_RES, y ambas retornan un entero.

```
   // Obtener el número de registros seleccionados:
   i = (int) mysql_num_rows(res);
```

```
    j = (int) mysql_num_fields(res);
    // Mostrar el número de registros seleccionados:
    cout << "Consulta:  SELECT * FROM gente" << endl;
    cout << "Numero de filas encontradas:  " << i << endl;
    cout << "Numero de columnas por fila:  " << j << endl;
```

Información sobre columnas

También podemos obtener información sobre cada una de las columnas del conjunto de resultados. Para ello disponemos de las funciones mysql_fetch_field, mysql_fetch_fields y mysql_fetch_field_direct.

La primera obtiene información sobre una de las columnas, en una estructura MYSQL_FIELD. La primera vez que se use devuelve la información sobre la primera columna, la segunda vez sobre la siguiente, etc.

```
    for(l = 0; l < j; l++) {
       columna = mysql_fetch_field(res);
       cout << "Nombre: " << columna->name << endl;
       cout << "Longitud: " << columna->length << endl;
       cout << "Valor por defecto: " << (columna->def ?
columna->def : "NULL") << endl;
    }
```

La función mysql_fetch_fields devuelve la información sobre todas las columnas a la vez, en un array de estructuras MYSQL_FIELD.

```
    columna = mysql_fetch_fields(res);
    for(l = 0; l < j; l++) {
       cout << "Nombre: " << columna[l].name << endl;
       cout << "Longitud: " << columna[l].length << endl;
       cout << "Valor por defecto: " << (columna[l].def ?
columna[l].def : "NULL") << endl;
    }
```

La tercera función, mysql_fetch_field_direct, devuelve la información sobre la columna indicada por el número especificado como segundo parámetro:

```
    // Información sobre columnas n:
    cout << endl << "Informacion sobre columna 1:" << endl;
    columna = mysql_fetch_field_direct(res, 1);
    cout << "Nombre: " << columna->name << endl;
    cout << "Longitud: " << columna->length << endl;
    cout << "Valor por defecto: " << (columna->def ? columna-
>def : "NULL") << endl;
    cout << endl;
```

Contenido de las filas de un conjunto de resultados

Por supuesto, también nos interesa obtener cada una de las filas contenidas en el conjunto de resultados. Para esa tarea se usa se la función mysql_fetch_row. Esta función también requiere como parámetro una estructura MYSQL_RES, y da como salida una estructura MYSQL_ROW.

```
    for(l = 0; l < i; l++) {
       row = mysql_fetch_row(res);
       cout << "Registro no. " << l+1 << endl;
       // Mostrar cada campo:
       for(k = 0 ; k < j ; k++)
          cout << ((row[k]==NULL) ? "NULL" : row[k]) << endl;
    }
```

También podemos usar como condición el valor de retorno de la función mysql_fetch_row, ya que tal valor es *NULL* si no quedan filas por recuperar.

El tipo MYSQL_ROW no es más que un array de cadenas, (un char**), que contiene los valores de todas las columnas en la forma de un array de cadenas. Si alguno de los valores de un atributo es *NULL*, el puntero correspondiente a esa columna será *NULL*.

Obtener longitudes de columnas

Mediante la función mysql_fetch_lengths podemos obtener las longitudes de todas las columnas de la fila actual de un conjunto de resultados.

Estos valores se pueden usar para copiar esos valores sin necesidad de calcular sus longitudes, o cuando algunas de las columnas contengan valores binarios, en cuyo caso no podremos usar *strlen* para calcular esas longitudes.

```
// Leer registro a registro y mostrar:
l=1;
for(l = 0; l < i; l++) {
   row = mysql_fetch_row(res);
   lon = mysql_fetch_lengths(res);
   cout << "Registro no. " << l+1 << endl;
   // Mostrar cada campo y su longitud:
   for(k = 0 ; k < j ; k++) {
      cout << ((row[k]==NULL) ? "NULL" : row[k]);
      cout << " longitud: " << lon[k] << endl;
   }
}
```

Acceder a filas del conjunto de forma aleatoria

Mediante mysql_fetch_row accedemos a las filas del conjunto de resultados de forma secuencial. Sin embargo, todas las filas del conjunto de resultados están en memoria, de modo que podemos acceder a ellas en cualquier orden, o acceder sólo a algunas de ellas.

Para acceder a una fila arbitraria se usa la función mysql_data_seek. Esta función precisa dos parámetros. El primero es el conjunto de resultados y el segundo un desplazamiento entre 0 y mysql_num_rows(result)-1.

```
// 3ª fila:
mysql_data_seek(res, 2);
```

```
row = mysql_fetch_row(res);
cout << "Tercera fila" << endl;
for(k = 0 ; k < j ; k++) {
   cout << ((row[k]==NULL) ? "NULL" : row[k]) << endl;
}
```

También podemos usar la función mysql_row_tell para obtener el valor de desplazamiento de la fila actual.

Ese valor se puede usar como argumento en llamadas a la función mysql_row_seek, pero el valor usado por estas dos funciones no es un número de fila, por lo tanto, no se puede usar en la función mysql_data_seek.

```
MYSQL_ROW_OFFSET pos;
pos = mysql_row_tell(res);
// lecturas de filas
mysql_row_seek(res, pos);
row = mysql_fetch_row(res);
cout << "Fila guardada" << endl;
for(k = 0 ; k < j ; k++) {
   cout << ((row[k]==NULL) ? "NULL" : row[k]) << endl;
}
```

Trabajar con conjuntos de resultados muy grandes

Si el conjunto de resultados contiene muchas filas puede ser poco práctico usar la función mysql_store_result, ya que se requerirá mucha memoria para almacenar el conjunto completo, posiblemente más de la disponible.

Cuando eso suceda, o cuando preveamos que esa situación puede suceder, podemos usar la función mysql_use_result en su lugar.

Esta función no carga el conjunto de resultados en memoria, y se usa la función mysql_fetch_row para recuperar las filas una por una.

MySQL recomienda no realizar procesos muy largos con cada fila, ya que la tabla está bloqueada para otros usuarios hasta que se llame a mysql_free_result.

```
   // El mismo proceso usando mysql_use_result:
   // Hacer una consulta con el comando "SELECT * FROM
gente":
   if(mysql_query(myData, "SELECT * FROM gente")) {
      // Error al realizar la consulta:
      cout << "ERROR: " << mysql_error(myData) << endl;
      mysql_close(myData);
      return 2;
   }
   if((res = mysql_use_result(myData))) {
      j = (int) mysql_num_fields(res);
      while(row = mysql_fetch_row(res)) {
         for(k = 0 ; k < j ; k++)
            cout << ((row[k]==NULL) ? "NULL" : row[k]) <<
endl;
      }

      // Liberar el resultado de la consulta:
      mysql_free_result(res);
   }
```

En este ejemplo hacemos uso del hecho de que el valor de retorno de mysql_fetch_row es NULL si no quedan filas por leer. No tenemos otra opción, ya que la función mysql_num_rows no funcionará correctamente hasta que se terminen de procesar todas las filas.

Las funciones mysql_data_seek, mysql_row_seek y mysql_row_tell no se pueden usar cuando se recupera un conjunto de resultados con la función mysql_use_result.

Ejemplo

Veamos un ejemplo completo en el que se aplican estas funciones.

```
/*
  Name: MySQL Ejemplo 1
  Author: Salvador Pozo Coronado, salvador@conclase.net
  Date: 07/05/2005
  Description: Ejemplo para mostrar contenidos de bases de
datos
  usando MySQL.
  Nota: incluir la opción "-lmysql" en las opciones del
linker
  dentro de las opciones de proyecto.
*/

// Includes...
#include <iostream>
#include <windows.h>
#include <mysql/mysql.h>
#include <mysql/mysqld_error.h>
#include <cstring>
#include <cstdio>

using namespace std;

// Programa principal
int main()
{
   // Variables
   MYSQL           *myData;
   MYSQL_RES       *res;
   MYSQL_ROW        row;
   MYSQL_FIELD     *columna;
   int              i, j, k, l;
   unsigned long   *lon;
   MYSQL_ROW_OFFSET pos;

   // Intentar iniciar MySQL:
   if(!(myData = mysql_init(0))) {
      // Imposible crear el objeto myData
      cout << "ERROR: imposible crear el objeto myData." <<
endl;
      rewind(stdin);
      getchar();
      return 1;
   }

   // Debe existir un usuario "curso" con clave de acceso
"clave" y
   // con al menos el privilegio "SELECT" sobre la tabla
```

```
   // "prueba.gente"
   if(!mysql_real_connect(myData, NULL, "curso", "clave",
"prueba", MYSQL_PORT, NULL, 0)) {
      // No se puede conectar con el servidor en el puerto
especificado.
      cout << "Imposible conectar con servidor mysql en el
puerto "
           << MYSQL_PORT << " Error: " <<
mysql_error(myData) << endl;
      mysql_close(myData);
      rewind(stdin);
      getchar();
      return 1;
   }

   // Conectar a base de datos.
   if(mysql_select_db(myData, "prueba")) {
      // Imposible seleccionar la base de datos,
posiblemente no existe.
      cout << "ERROR: " << mysql_error(myData) << endl;
      mysql_close(myData);
      rewind(stdin);
      getchar();
      return 2;
   }

   // Hacer una consulta con el comando "SELECT * FROM
gente":
   if(mysql_query(myData, "SELECT * FROM gente")) {
      // Error al realizar la consulta:
      cout << "ERROR: " << mysql_error(myData) << endl;
      mysql_close(myData);
      rewind(stdin);
      getchar();
      return 2;
   }

   // Almacenar el resultado de la consulta:
   if((res = mysql_store_result(myData))) {
      // Obtener el número de registros seleccionados:
      i = (int) mysql_num_rows(res);
      // Obtener el número de columnsa por fila:
      j = (int) mysql_num_fields(res);

      // Mostrar el número de registros seleccionados:
      cout << "Consulta:  SELECT * FROM gente" << endl;
      cout << "Numero de filas encontradas:  " << i << endl;
      cout << "Numero de columnas por fila:  " << j << endl;
```

```
      // Información sobre columnas usando
mysql_fetch_field:
      cout << endl << "Informacion sobre columnas:" << endl;
      for(l = 0; l < j; l++) {
         columna = mysql_fetch_field(res);
         cout << "Nombre: " << columna->name << endl;
         cout << "Longitud: " << columna->length << endl;
         cout << "Valor por defecto: " << (columna->def ?
columna->def : "NULL") << endl;
      }
      cout << endl;

      // Información sobre columnas usando
mysql_fetch_fields:
      cout << endl << "Informacion sobre columnas:" << endl;
      columna = mysql_fetch_fields(res);
      for(l = 0; l < j; l++) {
         cout << "Nombre: " << columna[l].name << endl;
         cout << "Longitud: " << columna[l].length << endl;
         cout << "Valor por defecto: " << (columna[l].def ?
columna[l].def : "NULL") << endl;
      }
      cout << endl;

      // Información sobre columnas n, usando
mysql_fetch_field_direct:
      cout << endl << "Informacion sobre columna 1:" <<
endl;
      columna = mysql_fetch_field_direct(res, 1);
      cout << "Nombre: " << columna->name << endl;
      cout << "Longitud: " << columna->length << endl;
      cout << "Valor por defecto: " << (columna->def ?
columna->def : "NULL") << endl;
      cout << endl;

      // Leer registro a registro y mostrar:
      l=1;
      for(l = 0; l < i; l++) {
         row = mysql_fetch_row(res);
         lon = mysql_fetch_lengths(res);
         cout << "Registro no. " << l+1 << endl;
         // Mostrar cada campo y su longitud:
         for(k = 0 ; k < j ; k++) {
            cout << ((row[k]==NULL) ? "NULL" : row[k]);
            cout << " longitud: " << lon[k] << endl;
         }
      }
```

```
      cout << endl;

      // Mostrar sólo 3ª fila:
      mysql_data_seek(res, 2);
      row = mysql_fetch_row(res);
      cout << "Tercera fila" << endl;
      for(k = 0 ; k < j ; k++) {
         cout << ((row[k]==NULL) ? "NULL" : row[k]) << endl;
      }

      // Almacenar posición de la fila actual (la 4ª):
      pos = mysql_row_tell(res);
      // Lectura de filas hasta el final:
      while(mysql_fetch_row(res));
      // Recuperar la posición guardada:
      mysql_row_seek(res, pos);
      row = mysql_fetch_row(res);
      cout << "Cuarta fila" << endl;
      for(k = 0 ; k < j ; k++) {
         cout << ((row[k]==NULL) ? "NULL" : row[k]) << endl;
      }

      cout << endl;

      // Liberar el resultado de la consulta:
      mysql_free_result(res);
   }

   // El mismo proceso usando mysql_use_result:
   // Hacer una consulta con el comando "SELECT * FROM
gente":
   if(mysql_query(myData, "SELECT * FROM gente")) {
      // Error al realizar la consulta:
      cout << "ERROR: " << mysql_error(myData) << endl;
      mysql_close(myData);
      rewind(stdin);
      getchar();
      return 2;
   }

   // Mostrar todas las filas:
   if((res = mysql_use_result(myData))) {
      j = (int) mysql_num_fields(res);
      while(row = mysql_fetch_row(res)) {
         for(k = 0 ; k < j ; k++)
            cout << ((row[k]==NULL) ? "NULL" : row[k]) <<
endl;
```

```
        }

        // Liberar el resultado de la consulta:
        mysql_free_result(res);
    }
    // Cerrar la conexión
    mysql_close(myData);

    // Esperar a que se pulse una tecla y salir.
    rewind(stdin);
    getchar();
    return 0;
}
```

3 Crear una base de datos

Hemos empezado por lo más simple: leer datos desde una base de datos existente. En este capítulo veremos cómo crear y eliminar bases de datos y tablas.

En este capítulo y el siguiente crearemos una aplicación de ejemplo para gestionar una base de datos simple de contactos, con sólo nombres y números de teléfono.

La base de datos se llamará "agenda", y para poder empezar a trabajar con ella necesitaremos un usuario con acceso a esa base de datos y con todos los privilegios. De modo que empezaremos por abrir una sesión **MySQL** y conceder al usuario "curso" todos los privilegios sobre la tabla "agenda". Recondermos que esto se puede hacer aunque tal base de datos no exista:

```
mysql> GRANT ALL ON agenda.* TO curso;
Query OK, 0 rows affected (0.08 sec)
```

La base de datos "agenda" contendrá dos tablas, una con los nombres de los contactos, y una segunda con los números de teléfono. Ambas tablas estarán interrelacionadas por una clave de indentificación de usuario:

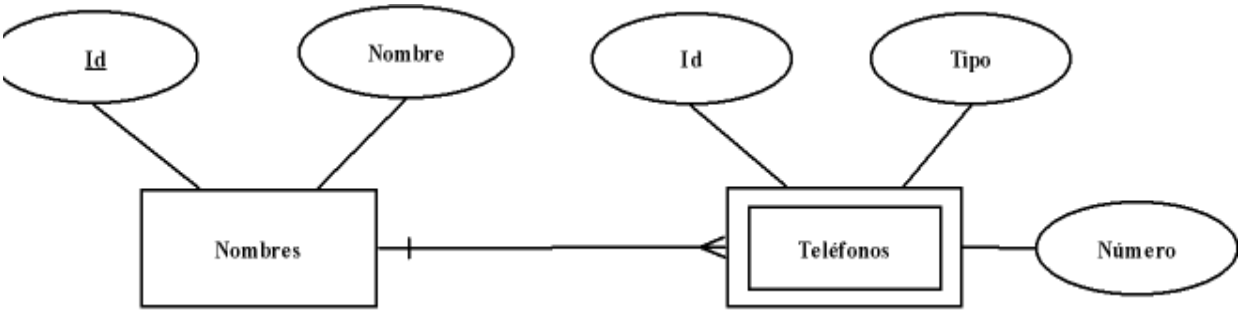


Diagrama de Agenda

La estructura de la tabla de nombres es:

- id: clave identificadora. Usaremos un entero autoincrementado.
- nombre: nombre de contacto. Usaremos una cadena de longitud variable de 40 caracteres.

En cuanto a la tabla de teléfonos, la estructura es:

- id: clave identificadora de contacto. Clave foránea de la tabla de nombres. Crearemos una referencia de modo que las modificaciones de clave se transmitan en cascada y los borrados también.
- tipo: tipo de teléfono. Indicará si es un celular, fijo, particular, etc. Usaremos una cadena de longitud variable de 20 caracteres.
- numero: número de teléfono. Usaremos una cadena de longitud variable de 15 caracteres.

Por supuesto, usaremos tablas **InnoDB**.

Averiguar si existe una base de datos

Antes de empezar a trabajar con una base de datos puede ser interesante saber si tal base de datos existe o no. Esto nos permitirá crear la base de datos si es la primera vez que se ejecuta la aplicación o si se ha eliminado la base de datos desde la última ejecución.

Para averiguar si una base de datos existe intentaremos activarla, y si se produce un error, verificaremos si ese error es debido a que la base de datos no existe. Para ello comprobaremos si el valor de error es ER_BAD_DB_ERROR. Vamos a crear una función C++ para esta tarea:

```
bool ExisteDB(MYSQL *myData, char *db)
{
   // Conectar a base de datos.
   if(mysql_select_db(myData, db)) {
      if(ER_BAD_DB_ERROR == mysql_errno(myData)) return
false;
```

```
    }
    return true;
}
```

Otra alternativa, quizás mucho mejor, es verificar si existe la base de datos usando una sentencia SHOW DATABASES. Si mostramos sólo las bases de datos cuyo nombre coincida con el de la que deseamos verificar, bastará con comprobar el número de filas retornadas. Si ese número es cero significa que la base de datos no existe:

```
bool ExisteDB(MYSQL *myData, char *db)
{
   char *consulta;
   char *plantilla = "SHOW DATABASES LIKE \'%s\'";
   MYSQL_RES *res;
   bool valorret = true;

   consulta = new char[strlen(db)+strlen(plantilla)];
   sprintf(consulta, plantilla, db);

   if(!mysql_query(myData, consulta)) {
      if((res = mysql_store_result(myData))) {
         // Procesar resultados
         if(!mysql_num_rows(res)) valorret = false;
         // Liberar el resultado de la consulta:
         mysql_free_result(res);
      }
   }
   delete[] consulta;
   return valorret;
}
```

Hemos tenido que hacer uso de memoria dinámica para preparar la cadena para hacer la consulta. Esto será muy habitual, sobre todo cuando las consultas se compliquen, y requieran selecciones y proyecciones (seleccionar columnas y filas). Más adelante crearemos una función para hacer este proceso más sencillo.

Crear una base de datos

A pesar de que existe una función en el API C de **MySQL** para crear bases de datos, mysql_create_db, su uso está desaconsejado, y es preferible usar mysql_query para lanzar una consulta CREATE DATABASE.

```
   if(!ExisteDB(myData, "agenda")) {
      cout << "La base de datos \"agenda\" no existe." <<
endl;
      cout << "La creamos..." << endl;
      mysql_query(myData, "CREATE DATABASE agenda");
   }
```

Esto creará la base de datos "agenda" si no existe previamente, pero no creará ninguna tabla.

Comprobar si una tabla existe

Para comprobar si existe una tabla podemos hacer algo parecido a lo que hemos hecho con la base de datos. Intentaremos hacer una consulta sobre la tabla, y si fracasamos, verificaremos si el error es ER_NO_SUCH_TABLE. En tal caso, la tabla no existe.

Crearemos otra función C++ para esta tarea:

```
bool ExisteTabla(MYSQL *myData, char *db, char *tabla)
{
   char *consulta;
   char *plantilla = "SELECT * FROM %s.%s";
   bool valorret = true;

   consulta = new
char[strlen(db)+strlen(tabla)+strlen(plantilla)-1];
   sprintf(consulta, plantilla, db, tabla);
   if(mysql_query(myData, consulta)) {
      if(ER_NO_SUCH_TABLE == mysql_errno(myData)) valorret =
false;
   }
   delete[] consulta;
   return valorret;
}
```

Otra opción, mucho más recomendable, es hacer una consulta usando la sentencia SHOW TABLES, de un modo análogo al usado con las bases de datos:

```
bool ExisteTabla(MYSQL *myData, char *db, char *tabla)
{
   char *consulta;
   char *plantilla = "SHOW TABLES FROM %s LIKE \'%s\'";
   MYSQL_RES *res;
   bool valorret = true;

   consulta = new
char[strlen(db)+strlen(tabla)+strlen(plantilla)-1];
   sprintf(consulta, plantilla, db, tabla);

   if(!mysql_query(myData, consulta)) {
      if((res = mysql_store_result(myData))) {
         // Procesar resultados
         if(!mysql_num_rows(res)) valorret = false;
         // Liberar el resultado de la consulta:
         mysql_free_result(res);
      }
   }
   delete[] consulta;
   return valorret;
}
```

Crear una tabla

Para crear una tabla usaremos la sentencia CREATE TABLE. El proceso se explica con detalle en el capítulo 7 del curso de **MySQL**.

Para crear las tablas que necesitamos para esta base de datos usaremos las siguientes consultas SQL:

```
mysql> CREATE TABLE agenda.nombres (
    -> id INT NOT NULL AUTO_INCREMENT,
    -> nombre VARCHAR(40),
    -> PRIMARY KEY (id))
    -> ENGINE=InnoDB;
Query OK, 0 rows affected (0.61 sec)
```

```
mysql> CREATE TABLE agenda.telefonos (
    -> id INT NOT NULL,
    -> tipo VARCHAR(20),
    -> numero VARCHAR(15),
    -> FOREIGN KEY (id) REFERENCES nombres(id)
    -> ON DELETE CASCADE
    -> ON UPDATE CASCADE)
    -> ENGINE=InnoDB;
Query OK, 0 rows affected (0.14 sec)

mysql>
```

En nuestro programa C++ crearemos las tablas que no existan usando este código:

```
   if(!ExisteTabla(myData, "agenda", "nombres")) {
      cout << "La tabla \"Nombres\" no existe." << endl;
      cout << "La creamos..." << endl;
      mysql_query(myData, "CREATE TABLE agenda.nombres ("
         "id INT NOT NULL AUTO_INCREMENT, "
         ";nombre VARCHAR(40), "
         ";PRIMARY KEY (id)) "
         "ENGINE=InnoDB");
   }

   if(!ExisteTabla(myData, "agenda", "telefonos")) {
      cout << "La tabla \"Telefonos\" no existe." << endl;
      cout << "La creamos..." << endl;
      mysql_query(myData, "CREATE TABLE agenda.telefonos ("
         "id INT NOT NULL, "
         "tipo VARCHAR(20), "
         "numero VARCHAR(15), "
         "FOREIGN KEY (id) REFERENCES nombres(id) "
         "ON DELETE CASCADE "
         "ON UPDATE CASCADE) "
         "ENGINE=InnoDB");
   }
```

Eliminar una tabla

Análogamente podemos eliminar tablas usando consultas con la sentencia DROP TABLE:

```
    cout << "Eliminar tabla de nombres" << endl;
    mysql_query(myData, "DROP TABLE agenda.nombres");
```

Eliminar una base de datos

Lo mismo sirve para eliminar bases de datos, usando la sentencia DROP DATABASE:

```
    cout << "Eliminar base de datos" << endl;
    mysql_query(myData, "DROP DATABASE agenda");
```

4 Añadir, modificar y eliminar datos

En este capítulo veremos cómo manipular datos contenidos en una base de datos, añadir, modificar o eliminar datos de tablas.

Insertar datos en una tabla

Evidentemente, la forma más simple de introducir datos en una tabla es usar una sentencia INSERT:

```
    mysql_query(myData, "INSERT INTO nombres (nombre) VALUES
"
       "('Carlos'), "
       "('Felipe'), "
       "('Irene')");
```

Pero esto no es muy práctico, ya que generalmente tendremos que introducir datos suministrados por el usuario de la aplicación, y eso significa que debemos leer valores de atributos y crear consultas con valores variables.

En una consulta INSERT necesitaremos, generalmente, dos listas de valores. La primera es una lista de atributos, y la segunda una lista de valores para esos atributos. En el ejemplo usamos una lista de listas de valores de atributos, pero podemos simplificar el proceso haciendo una consulta para cada inserción en la tabla.

En nuestra base de datos "agenda", la tabla de nombres tiene dos atributos, pero el primero de ellos es un campo *AUTO_INCREMENT*, por lo que no será necesario especificar un valor cuando se inserten datos, ese valor se calcula de forma

automática. Por lo tanto, la lista de atributos contiene un único valor: 'nombre'.

Veamos un ejemplo:

```
    // Insertar datos:
    cout << "introducir datos" << endl;
    char nombre[41];
    char *consulta;
    char resp;
    char patronnombre[] = "INSERT INTO nombres (nombre)
VALUES(\'%s\')";
    do {
       // Leer contacto:
       cout << "Nombre de contacto: " << flush;
       cin.getline(nombre, 41);
       consulta = new
char[strlen(patronnombre)+strlen(nombre)-1];
       sprintf(consulta, patronnombre, nombre);
       mysql_query(myData, consulta);
       delete[] consulta;
       cout << "Otro contacto?: " << flush;
       cin >> resp;
       cin.ignore();
    } while (toupper(resp) == 'S');
```

Obtener el último valor autoincrementado insertado

Frecuentemente, como sucede en nuestra base de datos de ejemplo, tenemos que insertar datos en una tabla interrelacionada con otra. Para hacerlo necesitamos el valor del atributo referenciado. En nuestro ejemplo necesitamos conocer el último valor de 'id' generado para insertar los registros de teléfonos usando el mismo valor.

Para conseguir ese valor hay dos posibilidades. Una consiste en hacer una consulta para obtener el valor de la función LAST_INSERT_ID(). La otra, que nos resultará mucho más práctica,

consiste en usar la función del API mysql_insert_id, que hace lo mismo, pero nos evita procesar un conjunto de resultados:

```
   // Insertar datos:
   cout << "introducir datos" << endl;
   char nombre[41];
   char tipo[21];
   char numero[15];
   char *consulta;
   char resp;
   my_ulonglong id;
   char patronnombres[] = "INSERT INTO nombres (nombre)
VALUES(\'%s\')";
   char patrontelefonos[] = "INSERT INTO telefonos (id,
tipo, numero) "
      "VALUES(\'%11d\', \'%s\', \'%s\')";
   do {
      // Leer contacto:
      cout << "Nombre de contacto: " << flush;
      cin.getline(nombre, 41);
      consulta = new
char[strlen(patronnombres)+strlen(nombre)-1];
      sprintf(consulta, patronnombres, nombre);
      mysql_query(myData, consulta);
      delete[] consulta;
      id = mysql_insert_id(myData);
      // Leer teléfonos:
      do {
         cout << "Tipo de telefono: " << flush;
         cin.getline(tipo, 21);
         cout << "Numero de telefono: " << flush;
         cin.getline(numero, 15);
         consulta = new
char[strlen(patrontelefonos)+11+strlen(tipo)+
            strlen(numero)-7];
         sprintf(consulta, patrontelefonos, static_cast<int>
(id),
            tipo, numero);
         mysql_query(myData, consulta);
         delete[] consulta;
         cout << "Otro telefono?: " << flush;
         cin >> resp;
         cin.ignore();
      } while (toupper(resp) == 'S');
      cout << "Otro contacto?: " << flush;
      cin >> resp;
```

```
      cin.ignore();
   } while (toupper(resp) == 'S');
```

Eliminar datos de una tabla

Eliminar filas de una tabla es igualmente simple. Nos limitaremos a hacer una consulta DELETE con una cláusula *WHERE* de modo que sólo eliminemos las filas que queramos.

En nuestro ejemplo "agenda", las filas de teléfonos están relacionadas mediante una clave foránea a las filas de nombres, de modo que si se elimina una fila de nombres, automáticamente se eliminan todas las filas de teléfonos con el mismo valor de 'id'. Esto nos facilita la tarea de eliminar contactos completamente:

```
   // Eliminar un contacto:
   char patrondelete[] = "DELETE FROM nombres WHERE nombre =
\"%s\"";
   cout << "Nombre a eliminar de agenda: " << flush;
   cin.getline(nombre, 41);
   consulta = new
char[strlen(patrondelete)+strlen(nombre)-1];
   sprintf(consulta, patrondelete, nombre);
   mysql_query(myData, consulta);
   delete[] consulta;
```

Puede ser interesante conocer el número de filas afectadas por la sentencia DELETE. Se puede obtener una información análoga a la presentada en el cliente **MySQL** usando la función del API mysql_affected_rows:

```
mysql> DELETE FROM nombres WHERE nombre="Pepito";
Query OK, 1 row affected (0.01 sec)
```

Por ejemplo:

```
   cout << "Filas eliminadas: " <<
(int)mysql_affected_rows(myData) << endl;
```

Actualizar filas de una tabla

El proceso para modificar filas se explica con detalle en el capítulo 8 del curso de **MySQL**. Para hacerlo desde un programa C/C++ bastará con hacer las consultas UPDATE o REPLACE adecuadas.

Tabla de contenido

- Obtener el último valor autoincrementado insertado
- Eliminar datos de una tabla
- Actualizar filas de una tabla